

Министерство образования и науки Российской Федерации

Вологодский государственный технический университет

А. П. Ерёменко

Аппаратное обеспечение ПК

Часть I

Введение в базовые принципы организации ЭВМ

*Утверждено редакционно-издательским советом ВоГТУ
в качестве учебного пособия*

Вологда
2013

УДК 004.382.7(075.8)
ББК 32.973.26
Е 70

Рецензенты:

Канд. экон. наук, доцент кафедры общеобразовательных специальных дисциплин
Санкт-Петербургского государственного университета в г. Вологде *Ю.А. Жаравина*;
канд. пед. наук, доцент, зав. кафедрой информационных технологий и МПИ
Вологодского государственного педагогического университета *О.Б. Голубев*

Ерёменко, А.П.

Е 70 Аппаратное обеспечение ПК. Часть I. Введение в базовые принципы организации ЭВМ:
учебное пособие / А.П. Ерёменко. – Вологда: ВоГТУ, 2013. – 251 с.

ISBN 978–5–87851–501–6

В учебном пособии рассматриваются математические и физические основания архитектуры ЭВМ, а также базовые вопросы информатики, имеющие отношение к архитектуре компьютеров.

Пособие подготовлено для бакалавров направления 230400.62, не имеющих подготовки в области hardware, изучающих дисциплины «Аппаратное обеспечение ПК», «Архитектура ЭВМ и систем», и может быть полезно начинающему специалисту в области аппаратной части компьютеров.

УДК 004.382.7(075.8)
ББК 32.973.26

ISBN 978–5–87851–501–6

© ВоГТУ, 2013
© Ерёменко А.П., 2013

Введение

За последние десять лет в области разработки ЭВМ произошло много примечательных событий. Закон Мура может являться самоисполняющимся пророчеством, но, несмотря на это, сложно отрицать колоссальный прогресс вычислительной техники, коснувшийся в последние годы самых разных областей: от супервычислений (развитие кластерных технологий, распространение GPGPU и FPGA) и до планшетного бума, смартфонов, облачных хранилищ. Написать хорошее пособие по архитектуре вычислительных систем в таких условиях чрезвычайно сложно, и тому есть несколько причин.

Во-первых, и так было почти всегда, hardware устаревает уже в процессе его описания и, к моменту выхода книги, многие архитектуры, которые казались важными автору в процессе работы над ней, превращаются практически в антиквариат, интересный только коллекционерам.

Во-вторых, предсказать направление развития ЭВМ более или менее достоверно можно лишь в самых общих чертах. Как показывает опыт, все действительно прорывные открытия практически никто не ждал и не готовился к ним – начиная от изобретения мыши и заканчивая вторжением мобильной электроники на рынок ПК и ноутбуков. Попытки написать что-нибудь фундаментальное, скорее всего, приведут к повторению ситуации с такими известными высказываниями, как «640 кб ОЗУ должно быть достаточно для каждого компьютера»¹, и «ни у кого не может возникнуть необходимость иметь компьютер в своем доме»².

В-третьих, и это уже только наша проблема, положение дел с разработкой архитектуры ЭВМ в России, скажем честно, печально. Несмотря на то, что многие отечественные программисты заслуженно пользуются успехом на Западе и на наличие когда-то действительно мощной научной школы и разработок мирового уровня («Сетунь», серии МИР, БЭСМ, «Минск» и т.п.), развитие микроэлектроники в СССР практически остановилось после внедрения ЕС ЭВМ. В отрасли, где зачастую прогресс измеряется даже не месяцами, а неделями, многолетняя стагнация привела к невообразимым последствиям. Ситуация 90-х г. не улучшила положение, и сейчас Россия совершенно вытеснена с мировой арены разработки архитектуры компьютеров. Это означает не только отсутствие разработки и производства собственных ЭВМ, но и всего, что этому предшествует – конкурентоспособных научных школ, современных учебников, непрерывной традиции преподавания на новом практическом материале, активного участия студентов в разработке и стартапах и т.п.

При этом мы вполне способны вести фундаментальные исследования в области перспективных компьютерных технологий, хотя и на западные гранты и/или не у нас (открытие графена, например), и писать хорошо продаваемый код, но промежуточный уровень – собственно разработка hardware закончился для России в 1971 г. после клонирования IBM S/360. Печальную тенденцию можно оценить по тому, что последняя ЕС-1220 была выпущена в 1998 (клон мейнфрейма IBM S/390 1990 г., причем в разработке участвовала сама IBM, и она же предоставила программное обеспечение)³.

Многие не согласятся с этим – а как же «Эльбрус», «Мультиклет» и вообще, разве Сколково – не русская Силиконовая долина? Увы, рекламные обещания оказались весьма далеки от реальности, и далее мы подробно рассмотрим почему, а Сколково заслуженно запомнилось лишь нанолампочками А. Чубайса.

Добавим к этому сложности с получением формального образования в этой области – ведь программисту достаточно, в самом худшем случае, ручки и бумаги, чтобы создать действительно хороший алгоритм, тогда как обучение проектированию ЭВМ и работе с аппаратной частью требует значительных финансовых вложений в оборудование, непосильных сейчас для большинства вузов.

Поэтому большая часть учебников по курсу «Архитектура ЭВМ», за несколькими замечательными исключениями, которые можно найти в библиографии, представляет достаточно печальную картину. Там классификация Флинна или принципы фон Неймана, устаревшие чуть ли не сразу после создания, возводятся в ранг абсолютного знания, а описание работы процессора рассматривается на примере реального режима 8086 под DOS с теми самыми «640 кб ОЗУ».

¹ Приписывается Биллу Гейтсу, 1981 г. Отметим, что в то время 640 кб было не просто много, а очень много – отечественные клоны PDP-11 1970-х годов – СМ ЭВМ имели ОЗУ от 64 до 512 кб, и этого хватало на работу в многотерминальном режиме. Intel 8080 и Z80 из-за ограничений разрядности адресовали не более 64 кб, а знаменитый «Спектрум» имел вообще 16 кб. Это еще раз напоминает нам, что, даже досконально зная ситуацию сегодня, невозможно угадать, что произойдет с архитектурой ЭВМ завтра.

² Кен Олсон, основатель DEC, 1977 г. Опять же в те годы сложно было найти более справедливое высказывание, учитывая размеры и назначение ЭВМ тех лет.

³ Далее будут подробно разобраны многие отечественные и зарубежные архитектуры, так что читатель сам получит возможность оценить ситуацию с развитием ЭВМ в России.

Но, к сожалению, данная дисциплина – не теория алгоритмов, которой можно без проблем обучиться по классическим книгам Кнута еще 1970-х годов. Необходимо наличие максимально возможной новой информации, поэтому большинство вещей, не касающихся истории, в книге даются на год ее издания.

Это пособие призвано служить кратким введением в математические и физические принципы архитектуры компьютера. Почему была выбрана такая, достаточно необычная тема? Дело в том, что в новых учебных планах для бакалавриата классическая дисциплина «Архитектура ЭВМ», читаемая обычно на 3 курсе, заменена на «Аппаратную часть ПК», читаемую в самом начале обучения. Очевидно, что в таких условиях сложно сохранить приемлемый уровень подачи материала, т.к. в ходе лекций постоянно приходится отвлекаться на разъяснение вещей, уже очевидных для старших курсов. В результате, в ходе чтения лекций студентам в 2007-2013 гг. специальности 230400 «Информационные системы и технологии» кафедры ИСиТ ФГБОУ ВПО ВоГТУ у автора родилась идея написать общую вводную части в архитектуру ЭВМ.

В таком справочнике какие-то вещи ученикам напоминаются (например, основы полупроводниковой техники), какие-то излагаются для них впервые (например, теория кодирования). Вместе с этим в пособии как иллюстрация упоминаемых принципов рассматриваются достаточно необычные архитектуры ЭВМ разных времен – стекковые и теговые машины, троичные компьютеры и т.п. Автор надеется, что формальные математические выкладки будут немного разбавлены интересными историями о проектировании и разработке компьютеров, и уровень изложения, оставаясь достаточно строгим, не вызовет отторжения у неподготовленного читателя.

Для чтения математической части пособия достаточно базовых теоретико-множественных понятий – таких, как множество, отображение и т.п.; для части, посвященной информатике и физике, надеемся, хватит школьного курса. Книга разделена на три основные части:

В математической части излагаются основы систем счисления, математической логики, теории информации и теории абстрактных автоматов;

В физической части напоминает понятие полупроводимости и устройство полупроводниковых приборов, а также рассказывается о цифровых и аналоговых сигналах и проектировании схем.

В части, посвященной информатике, речь идет о типах данных, используемых ЭВМ, машинных словах, даются основы языка ассемблера.

Надеемся, что это небольшое пособие сможет выполнить свою задачу и поможет продуктивно осваивать такую сложную и интересную дисциплину, как архитектура ЭВМ, как бы она не называлась в дальнейшем.

Автор благодарен всем уважаемым коллегам с кафедры ИСиТ за непрерывное участие и поддержку во всех начинаниях, связанных с нелегкой судьбой этой дисциплины, а также своим студентам, без наличия искреннего интереса у которых, это пособие не имело бы смысла.

I. МАТЕМАТИКА

1.1. Системы счисления

В основе использования любых цифровых устройств, за редким исключением, лежит двоичная система счисления. Напомним кратко ее основные принципы. Системой счисления называется какой-либо формальный способ представления (не обязательно в письменном виде!) множества чисел. В качестве представляемого поля обычно рассматривается $\mathbb{N}$, но, при потребности, принципы записи естественным образом расширяются для представления $\mathbb{Z}$ и далее.

Система счисления обязана предоставлять каждому обозначаемому элементу однозначную стандартную запись (снова, не обязательно на бумаге, более того, для нас будет важно, как число записывается в более абстрактном виде – скажем, в памяти компьютера) и строится на основе знаний алгебраической и арифметической структуры представляемого поля.

Стандартное представление числа, целого или вещественного, как правило, является базовым элементом типизации данных в программировании и конструировании ЭВМ; на основе разработанной системы счисления можно построить и более сложные структуры для хранения в памяти векторов, матриц и прочих типов. Так же выбор системы счисления является одним из базовых элементов разработки аппаратной части ЭВМ, он зависит от доступной элементной базы и удобства пользователя, тесно связанного со сложностью архитектуры. Очевидно, что использовать в качестве базовой десятичную систему (как ENIAC) проще, с точки зрения низкоуровневого программирования, но ее аппаратная реализация значительно сложнее двоичной (как следующий за ним EDSAC и современные машины). Также, пользуясь методами теории информации, несложно показать, что оптимальной, с точки зрения размера кода и быстродействия, является троичная система, однако ее физическая реализация сопряжена со значительными неудобствами.

Системы счисления подразделяются на позиционные, непозиционные и смешанные (иногда считаемые обобщением позиционных), при этом, с точки зрения разработки ЭВМ, интерес представляют лишь первая и последняя. В позиционных системах счисления один и тот же знак (цифра) в записи числа имеет различные значения в зависимости от того места (разряда), где он расположен. Позиционная b – ричная система счисления полностью определяется числом b , называемым основанием системы. В подавляющем большинстве случаев $b \in \mathbb{N}$, случаи произвольного b (или нескольких различных оснований) относят к смешанным системам.

Произвольное беззнаковое число x в такой системе представляется в виде конечной линейной комбинации степеней основания:

$$x = \sum_{k=0}^{n-1} a_k b^k,$$

где a_k – набор базовых чисел (цифр системы счисления), удовлетворяющих неравенству $0 \leq a_k \leq b - 1$. Степени b^k в такой записи называются весовыми коэффициентами, а $a_k b^k$ – разрядом. Старшинство разрядов и, соответственно, порядок чтения числа определяются значением k – номером разряда. Как правило, в ненулевых x все левые нули опускаются, и представляемое число записывается просто его последовательностью b – ричных цифр, перечисляемых по убыванию старшинства разрядов слева направо: $x = a_{n-1} a_n \dots a_0$.

Вот некоторые ценные свойства позиционной системы счисления:

- Основание системы счисления в ней самой всегда записывается как 10; например, в двоичной системе счисления 10 означает число 2. Данное утверждение неприменимо к унарной системе счисления, в которой используется только одна цифра.

- Для записи числа x в b – ричной системе требуется $\lceil \log_b x \rceil + 1$ цифр.

- Естественный порядок на натуральных числах соответствует лексикографическому порядку на их представлениях в позиционной системе счисления. Поэтому сравнивать их представления можно поразрядно, начиная со старшего разряда, до тех пор, пока цифра в одном числе не будет больше соответствующей цифры в другом.

- Позиционная система счисления позволяет без труда выполнять сложение, вычитание, умножение, деление и деление с остатком чисел, зная только таблицу сложения однозначных чисел, а для трёх последних операций ещё и таблицу умножения в соответствующей системе.

Очевидно, что в позиционных системах чем больше основание, тем меньшее количество разрядов (то есть записываемых цифр) требуется при записи числа, при этом количество требуемых разнообразных цифр возрастает. Если вести речь о проектировании ЭВМ – это означает, что при малых b нам потребуется много простых однотипных элементов для хранения числа, при больших же b – незначительное количество сложных разнотипных элементов, имеющих много состояний, соответствующих разным цифрам. Так, в совре-

менной типовой ОЗУ используются микроконденсаторы, имеющие лишь два состояния – разряжен и заряжен, для хранения одного двоичного бита, ENIAC же нуждался в десяти защелках⁴ из 2-х триодов каждая, для хранения одного десятиразрядного числа, для полного 20-значного регистра, таким образом, требовалось 400 триодов.

1.1.1. Двоичная система

Наиболее распространенные позиционные системы счисления – это, конечно, двоичная, троичная, восьмеричная, десятичная и шестнадцатеричная. Двоичная система имеет богатую историю – бинарную систему кодирования применяли еще индийские математики в 200 г. до н.э. а также инки около 1000 г. н.э. Европу с двоичной системой познакомил, по-видимому, Френсис Бэкон, описав первую систему бинарного кодирования в классической книге об основах криптографии (1605). Эту идею развил Лейбниц в XVII веке в трактате «*Explication de l'Arithmétique Binaire*», где использовал уже современную запись двоичного числа в виде 0 и 1. В логику эти обозначения ввел в 1854 г. Джордж Буль, а первые теоретические разработки ее применения к цифровым машинам появились лишь в 1937 г. в диссертации Клода Шеннона – основателя теории информации.

Двоичная арифметика полностью определяется следующими таблицами:

Таблица сложения:

+	0	1
0	0	1
1	1	10*

Таблица вычитания:

-	0	1
0	0	1
1	1	0

Таблица умножения:

×	0	1
0	0	1
1	0	1

* - с переносом 1 в следующий разряд

Преобразование десятичных чисел в двоичные происходит с помощью последовательного деления и выписывания остатков, например:

$$\begin{aligned}
 19/2 &= 9, & \text{остаток } & \mathbf{1} \\
 9/2 &= 4, & \text{остаток } & \mathbf{1} \\
 4/2 &= 2, & \text{остаток } & \mathbf{0} \\
 2/2 &= 1, & \text{остаток } & \mathbf{0} \\
 1/2 &= 0, & \text{остаток } & \mathbf{1} \\
 19_{10} &= 10011_2
 \end{aligned}$$

Для перевода двоичного числа в десятичное, например 110001, необходимо записать его справа налево по разрядам:

$$1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 = 49$$

Можно также воспользоваться специальной таблицей степеней двойки:

1024	512	256	128	64	32	16	8	4	2	1
					1	1	0	0	0	1
					32	16	0	0	0	1

=49

Для перевода, например дроби 0.101, необходимо использовать следующую схему:

$$1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$$

Обратное преобразование легко понять, также обратившись к примеру. Переведем число 0.116 в двоичную дробь:

$$\begin{aligned}
 0.116 \times 2 &= \mathbf{0.232} \\
 0.232 \times 2 &= \mathbf{0.464} \\
 0.464 \times 2 &= \mathbf{0.928} \\
 0.928 \times 2 &= \mathbf{1.858} \\
 0.858 \times 2 &= \mathbf{1.712} \\
 0.712 \times 2 &= \mathbf{1.424} \\
 &\dots \\
 0.116_{10} &= 0.000111\dots_2
 \end{aligned}$$

⁴ О защелках и памяти ЭВМ подробнее будет рассказано в следующей части.

Двоичная система является наиболее экономичной из позиционных в том смысле, что для представления значащей цифры достаточно тривиального устройства, имеющего всего 2 состояния – это может быть триггер, электромеханическое реле, транзистор и т.п. Как показала практика, дешевле и надежнее создавать устройства, состоящие из минимально простых элементов, чем из меньшего, пусть даже на порядок, количества более сложных, так что, начиная с 1950-х г., двоичная система не имеет конкурентов в вычислительной технике.

1.1.2. Троичная система

Троичная система счисления существует в двух вариантах – симметричная и несимметричная. Симметричная система использует цифры $\{-1, 0, 1\}$, несимметричная, соответственно, $\{0, 1, 2\}$. Троичные цифры могут кодироваться различными способами.

Наиболее естественное представление (если у нас, конечно, имеется устройство, способное устойчиво принимать 3 фиксированных состояния) носит название трехуровневое кодирование (3-Level Coded Ternary, 3L CT, оно же т.н. однопроводное, так как для передачи сигнала нужен лишь один провод). В представлении 3L CT используются классические троичные цифры, симметричные или нет, а также троичный триггер, имеющий три состояния.

Конечно, было бы неплохо реализовать такую схему с использованием стандартных двоичных элементов, и тут на помощь приходят двухуровневые системы – т.н. двухбитные двоичнокодированные троичные цифры (2-Bit BinaryCodedTernary, 2B BCT или двухпроводное, т.к. для передачи троичной цифры нужно уже два провода).

В этом случае троичная цифра кодируется двумя разрядами двоичного числа, например так:

$$00 - 0, 01 - 1, 11 - 2$$

или так

$$00 - 0, 01 - 1, 10 - 2$$

Вообще троичные компьютеры обладают рядом значительных преимуществ по сравнению с двоичными. Например, при сложении тритов (аналог бита, только принимающий три значения) в троичных сумматорах количество сложений в 1,5 раза меньше, чем при сложении битов в двоичных сумматорах, и, следовательно, быстродействие приблизительно в 1,5 раза больше.

Также при применении симметричной троичной системы счисления сложение и вычитание производится в одних и тех же двухоперандных сумматорах-вычитателях без преобразования отрицательных чисел в дополнительные коды, т.е. ещё немного быстрее, чем в сумматорах с преобразованием отрицательных чисел в дополнительные коды.

Помимо этого, троичная система обладает большей емкостью, следовательно, код, записываемый в ней, короче аналогичного двоичного. Если рассматривать проблему экономичности системы счисления, т.е. возможность представления как можно большего количества чисел с использованием как можно меньшего общего количества знаков, то, при количестве базовых триггеров r общее количество записываемых ими знаков $m = r \cdot b$, а количество представимых ими чисел найдется как $b^{\frac{m}{b}}$.

Как функция от b , это выражение достигает максимума при $b = e \approx 2.718281 \dots$ Конечно, использовать трансцендентное число в качестве основания в случае реального компьютера невозможно, но из целочисленных значений ближе всего к нему именно число 3. Таким образом, теоретически, троичная система является наивыгоднейшей с точки зрения емкости кода и его длины.

Троичная логика целиком включает в себя двоичную логику как центральное подмножество, поэтому троичные компьютеры могут выполнять любые двоичные операции плюс операции троичной логики, упрощающие вычисления.

Позиционная целочисленная симметричная троичная система счисления была предложена итальянским математиком Фибоначчи (Леонардо Пизанский) для решения т.н. задачи о гирях, однако на Востоке она была известна на 200 лет раньше. Также задачу о наилучшей системе гирь рассматривал Лука Пачоли в XV в., помимо этого ею интересовались Баше, Эйлер и Менделеев, поэтому в отечественной литературе симметричную троичную систему счисления, доставляющую ее решение, также называют системой Баше-Менделеева.

Благодаря тому, что основание 3 нечётно, в троичной системе возможно симметричное относительно нуля расположение цифр: $-1, 0, 1$, с которым связано шесть ценных свойств:

- Наличие положительной и отрицательной цифр позволяет непосредственно представлять как положительные, так и отрицательные числа. При этом нет необходимости в специальном разряде знака и не надо вводить дополнительный (или обратный) код для выполнения арифметических операций с отрицательными числами. Все действия над числами, представленными в троичной системе счисления с цифрами 0, 1, -1 , выполняются естественно с учётом знаков чисел. Знак числа определяется знаком старшей значащей цифры числа: если она положительна, то и число положительно, если отрицательна, то и число отрицательно;
- Отсутствие проблемы округления (для округления достаточно просто отбросить ненужные цифры);

- Таблица умножения в этой системе, как отметил еще Коши, примерно в четыре раза короче;
 - Для изменения знака у представляемого числа нужно всего лишь инвертировать знаки у всех его цифр (т.н. инверсия Лукашевича);
 - При суммировании большого количества чисел значение для переноса в следующий разряд растёт с увеличением количества слагаемых не линейно, а пропорционально квадратному корню числа слагаемых;
 - По затратам числа знаков на представление чисел она равна троичной несимметричной системе;
- Как правило, в записи для удобства используют вместо -1 специальное обозначение $\bar{1}$. При этом положительные и отрицательные числа представляются элементарно, например:

$$\bar{1}01 = 9 - 1 = 8$$

$$10\bar{1} = -9 + 1 = -8$$

Непосредственное представление команд троичным кодом при программировании и при вводе в машину неудобно и неэкономно, поэтому в машинных кодах ЭВМ «Сетунь» применялась специальная вспомогательная девятиричная форма представления команд (подобно вспомогательным 8 и 16-ричным числам в двоичных машинах), где девятиричные цифры $\bar{4}, \bar{3}, \bar{2}, \bar{1}, 1, 2, 3, 4$ сопоставляются парам троичных и заменяются буквами:

Девятиричная цифра	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$	0	1	2	3	4
Троичная цифра	$\bar{1}1$	$\bar{1}0$	$\bar{1}\bar{1}$	$0\bar{1}$	0	1	$\bar{1}\bar{1}$	0	1
Латинский алфавит	W	X	Y	Z	—	—	—	—	—
Русский алфавит	Ж	Х	У	Ц	—	—	—	—	—

Существует большое количество возможных реализаций троичных элементов в компьютере. Наиболее распространены импульсные элементы, такие как ферритодиодные троичные элементы Н. П. Брусенцова, аналогичные двоичным элементам ЛЭМ-1 Л. И. Гутенмахера (они же магнитные усилители) – использовались в машине «Сетунь», а также потенциальные.

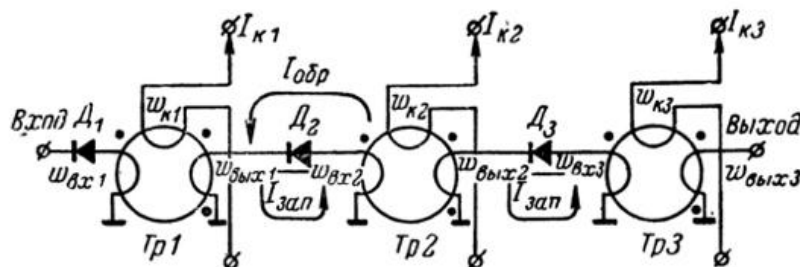
Они, в свою очередь, делятся на:

- Трёхуровневые 3L СТ, в которых трём устойчивым состояниям соответствуют три уровня напряжения (положительное, нулевое, отрицательное), (высокое, среднее, низкое). Объём передаваемых данных увеличивается в 1,5 раза на один разряд. Амплитуда сигнала помехи до 25% $U_{\text{п}}$;
- Двухуровневые 2L ВСТ, в которых логические элементы (инверторы) имеют два устойчивых состояния с двумя уровнями напряжения (высокое, низкое), а троичность работы достигается системой обратных связей. Амплитуда сигнала помехи до 50% $U_{\text{п}}$. Они делятся на:
 - Двухуровневые двухбитные 2L 2В ВСТ, по скорости приблизительно на 1/3 уступают троичным двухуровневым трёхбитным триггерам, по сравнению с обычными двоичными триггерами в 1,5 раза увеличивают аппаратные затраты;
 - Двухуровневые трёхбитные 2L 3В ВСТ, по сравнению с обычными двоичными RS-триггерами увеличивают объём хранимых и передаваемых данных в 1,5 раза на один разряд. Наиболее экономичны – уменьшают прямые аппаратные затраты приблизительно на 5,7 % по сравнению с аппаратными затратами на обычных двоичных триггерах;
- Смешанные, в которых вход данных трёхуровневый по одной линии и земле, а выход данных двухуровневый по трём линиям и земле.

Первую вычислительную машину с троичной системой счисления разработал английский изобретатель-самоучка Томас Фоулер в 1840 г. Его машина была механической и полностью деревянной и представляла собой устройство для табулирования функций – конкурента знаменитой дифференциальной машины Беббиджа. Фоулер работал банковским служащим, для ускорения работы он сделал таблицы для счета степенями двойки и тройки, а позже опубликовал эти таблицы в виде брошюры. Затем он полностью автоматизировал расчеты по таблицам, изготовив табулятор. Английская патентная система того времени была несовершенна, предыдущее изобретение Фоулера (термосифон для систем парового отопления) было скопировано с минимальными изменениями и запатентовано множеством недобросовестных изобретателей, поэтому, опасаясь, что его идею снова могут украсть, он решил изготовить машину в единственном экземпляре и из дерева. В силу низкого качества материала для обеспечения достаточной точности вычислений Фоулеру пришлось сделать машину весьма громоздкой, около 2 м в длину. Впрочем, как писал сам изобретатель в сопроводительной записке, отправляя устройство в Лондонский королевский колледж, «если бы ее можно было изготовить из металла, она бы оказалась не больше пишущей машинки». К сожалению, замечательное изобретение так и осталось незамеченным, оригинал машины также не сохранился до наших дней.

Более ста лет спустя, в 1954 году в МГУ должны были передать электронно-вычислительную машину М-2, но не сложилось, и кто-то предложил построить свою. Руководителем группы, осуществлявшей проектирование и изготовление машины, был назначен Николай Петрович Брусенцов. Задача была сделать маши-

ну предельно простой и недорогой, потому что никакого специального финансирования у проекта не было, поэтому из соображений экономичности и простоты архитектуры пришли к решению, что она будет троичной. К концу 1958 г. был закончен первый экземпляр машины, которой дали имя «Сетунь». Она была относительно невелика для вычислительных машин того поколения и занимала площадь 25–30 м². Благодаря изящной архитектуре она была способна выполнять 2000–4500 операций в секунду, обладала оперативной памятью в 162 девятитритных ячейки и запоминающим устройством на магнитном барабане емкостью 36–72 страницы по 54 ячейки каждая.



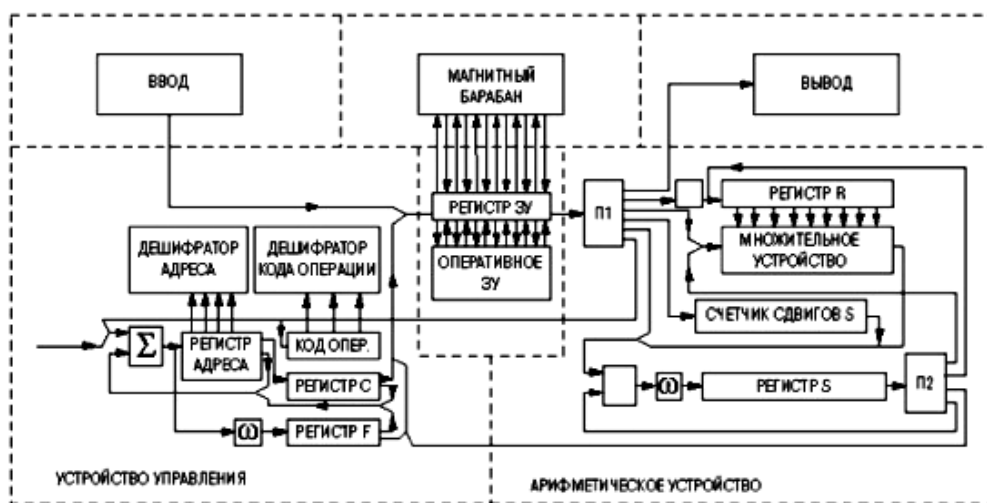
Ферритодиодный троичный элемент Н. П. Брусенцова

Машинных команд было всего 27 (причем три так и остались невостребованными), благодаря чему программный код получался весьма экономным; программирование непосредственно в машинных кодах было настолько простым, что для «Сетуни» даже не разрабатывали свой ассемблер. Данные вводили в машину с перфоленты, результаты выводились на телетайп (причем, что любопытно, отрицательные цифры печатались как обычные, но в перевернутом виде).

При эксплуатации машина показывала 95–98% полезного времени (расходуемого на решение задач, а не на поиск неисправностей и устранение неполадок), а в те времена очень хорошим результатом считалось, если машина могла дать хотя бы 60%. На межведомственных испытаниях 1960 года машину признали пригодной для массового использования, последовало распоряжение о серийном выпуске «Сетуни» на Казанском заводе математических машин. С 1961 по 1965 год было построено 47 экземпляров, которые работали по всей стране. Затем производство свернули, в качестве возможных причин указывали косность мышления бюрократов и, самое главное, компьютер оказался слишком дешевым в производстве и потому невыгодным для завода и покупателей. Как видим, традиционная российская беда распилов и откатов имеет долгую историю.



Сетунь, 1958 г., машинный зал МГУ



Блок-схема машины «Сетунь»

Впоследствии Николай Брусенцов и Евгений Жоголев разработали более современную версию машины, используя те же принципы троичности, – «Сетунь-70», но она так и не пошла в серийное производство. Единственный опытный образец работал в МГУ до 1987 года, после чего был принудительно по решению руководства отправлен в металлолом, ученым удалось спасти только пульт (сейчас выставляется в Политехническом Музее в Москве) и немного документации.

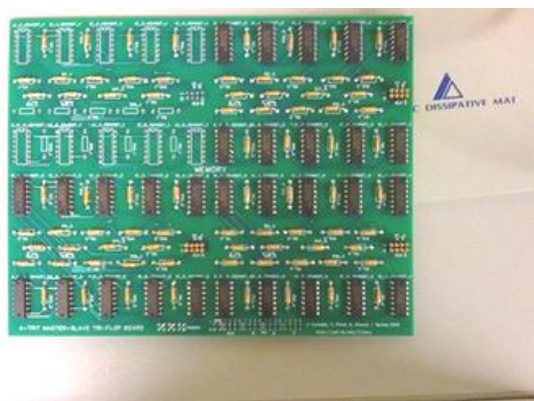
Следующим троичным проектом стал TERNAC – эмулятор троичной арифметики на двоичном компьютере, написанный на диалекте FORTRAN для Burroughs B1700 Джорджем Фридером (G. Frieder) в 1973 г. в Нью-Йоркском Университете. TERNAC поддерживал два формата чисел: с фиксированной точкой и с плавающей точкой. Слова с фиксированной точкой были длиной 24 трита, слова с плавающей точкой были длиной 48 тритов, 42 трита для мантиссы и 6 тритов для экспоненты.

Реализация эмулятора TERNAC была предпринята для того, чтобы узнать, возможна ли реализация не двоичной структуры на двоичном компьютере, и каков расход памяти и времени в такой реализации. Как проверка возможности, эта попытка была успешной, но скорость работы и расход памяти были того же порядка, как и при двоичных вычислениях.

В настоящее время троичных компьютеров, воплощенных в железе, не существует, несмотря на то, что некоторые компании, например Hurgres и даже IBM, публиковали пресс-релизы, посвященные таким разработкам. Из работ энтузиастов широко известны лишь две: проект троичной компьютерной архитектуры TCA2 команды Джеффа Коннели (Jeff Connelly, <http://jeff.tk:81/wiki/Trinary>) и проект Shaos – симулятор троичного компьютера на двоичном микроконтроллере 3niti Alpha Simu1⁵.

Команда Коннели работала в двух направлениях: создание трёхуровневых троичных логических элементов с использованием транзисторных КМОП-структур, входящих в микросхемы серии CD40xx, и уже на базе разработанных логических элементов создание компьютерной системы. TCA2 – это очень большое и сложное устройство, но если попытаться охарактеризовать его по параметрам процессорных возможностей, как объём выполняемого кода, максимальная длина программ и т.п., то видно, что это достаточно скромный компьютер. Проект был закончен в 2008 году, были некоторые упоминания о TCA3, но пока подробностей нет.

У проекта Shaos всё с точностью до наоборот – сам по себе симулятор в плане архитектуры предоставляет больше возможностей и требует небольших затрат, но это не настоящий троичный процессор, а аппаратно - программный симулятор на двоичном микроконтроллере.



Джефф Коннели и его плата TCA2

Почему же троичные компьютеры не вытеснили привычные нам системы и вряд ли уже вытеснят? На это есть несколько причин и самая главная из них – отсутствие необходимой элементной базы. В настоящее время не существует ни одного полупроводникового элемента, который совмещал бы в себе поддержку троичного сигнала и был бы столь же мал, прост, надежен и дешев, как обычный транзистор. Использование ферритодиодных элементов, обладающих естественной троичностью, сейчас невозможно – их габариты были хороши лишь для 1950-х гг., а уменьшить их невозможно. Использование же 2L ВСТ логики приводит к 1.5-2-х кратному усложнению схемы, что не оправдывает рост производительности в те же 1.5-2 раза, так как кроме повышенной сложности в недостатках мы имеем повышенное тепловыделение, снижение надежности и огромные проблемы с совместимостью. К сожалению троичные компьютеры остались в прошлом, и само по себе изменение системы счисления не является сейчас способом радикально увеличить производительность ЭВМ.

1.1.3. Восьми- и шестнадцатеричная система

Восьмеричная система часто используется в областях, связанных с цифровыми устройствами. Характеризуется лёгким переводом восьмеричных чисел в двоичные и обратно, путём замены восьмеричных чисел на триплеты двоичных. Ранее широко использовалась в программировании и вообще компьютерной документации, однако в настоящее время почти полностью вытеснена шестнадцатеричной.

Восьмеричная система активно применялась в системах PDP-8, ICL 1900 и мейнфреймах IBM, так как их байт был длиной 12, 24 или 36 бит. В результате восьмеричные числа были идеальным сокращением для этих платформ, так как размер их машинного слова был кратен трем, а восьмеричная цифра представляет три

⁵ <http://ternary.info>

двоичных. В результате 4, 8 или 12 восьмеричных цифр кратко кодировали все команды. Также восьмеричная система была удобна для первых дисплеев с семисегментными цифровыми индикаторами и калькуляторов на операторских консолях – двоичные были очень неудобны для человека, десятичные потребовали специального преобразования, а шестнадцатеричные были тогда излишни.

Все современные платформы используют машинное слово, кратное 16, поэтому восьмеричная система вышла из употребления, хотя некоторые платформы еще допускают удобное восьмеричное представление, такие как эмуляторы PDP-11 или старые процессоры Motorola 68k.

Пожалуй, наиболее известным современным применением восьмеричной системы является управление доступом в Unix-системах. Программа `chmod`, предназначенная для этого, работает из командной строки и понимает следующие значения:

двоичная	восьмеричная	символьная	права на файл	права на директорию
000	0	---	нет	нет
001	1	--x	выполнение	чтение файлов и свойств
010	2	-w-	запись	создание, удаление и переименование
011	3	-wx	запись и выполнение	все, кроме чтения списка файлов
100	4	r--	чтение	чтение имен файлов
101	5	r-x	чтение и выполнение	чтение всего
110	6	rw-	чтение и запись	не определено
111	7	rwx	полный доступ	полный доступ

При этом права назначаются отдельно пользователю, группе и остальным.

Например, знаменитая команда, означающая разрешение всем делать все, что угодно с файлом, записывается так:

`chmod 777 %filename`

где `%filename` – полное имя файла.

В соглашении Motorola восьмеричные цифры записываются с префиксом `@`, в соглашении Intel с постфиксом `Q`. DR-DOS DEBUG использует `\` для префикса. Вообще существовало огромное количество

стандартов, например, восьмеричное число 73 в зависимости от диалекта и соглашения можно записать как 073, Q73, 0o73, \ 73, 73o, @73 и даже как Ø73.

Шестнадцатеричная система широко используется в низкоуровневом программировании и компьютерной документации, поскольку в современных компьютерах минимальной единицей памяти является 8-битный байт, значения которого удобно записывать двумя шестнадцатеричными цифрами. Такое использование началось с системы IBM/360, где вся документация использовала шестнадцатеричную систему, в то время как в документации других компьютерных систем того времени (даже с 8-битными символами, как, например, PDP-11 или БЭСМ-6) использовали восьмеричную систему.

В стандарте Unicode номер символа принято записывать в шестнадцатеричном виде, используя не менее 4 цифр (при необходимости — с ведущими нулями). В разных языках программирования для записи шестнадцатеричных чисел используют различный синтаксис:

- В Ада и VHDL такие числа указывают так: «16#5A3#».
 - В Си и языках схожего синтаксиса, например, в Java, используют префикс «0x». Например, «0x5A3».
 - В некоторых ассемблерах используют букву «h», которую ставят после числа. Например, «5A3h».
- При этом если число начинается не с десятичной цифры, то для отличия от имён идентификаторов (например, констант) впереди ставится «0» (ноль): «0FFh» (255₁₀).
- Другие ассемблеры (AT&T, Motorola), а также Pascal и некоторые версии BASIC используют префикс «\$». Например, «\$5A3».
 - Некоторые иные платформы, например ZX Spectrum, в своих ассемблерах (MASM, TASM, ALASM, GENS и т. д.) использовали запись #5A3, обычно выровненную до одного или двух байт: #05A3.
 - Turbo Basic использует для указания шестнадцатеричных цифр сочетание «&h». Например, «&h5A3».
 - В Unix-подобных операционных системах (и многих языках программирования, имеющих корни в Unix/Linux) непечатаемые символы при выводе/вводе кодируются как 0xCC, где CC — шестнадцатеричный код символа.

Сложение в шестнадцатеричной системе определяется так:

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Преобразование шестнадцатеричного числа в десятичное:

Для перевода произвольного числа, например, 0x5A3, необходимо предпринять те же действия, что и в случае с двоичным:

$$3 \times 16^0 + 10 \times 16^1 + 5 \times 16^2 = 1443$$

Для перевода двоичного числа в шестнадцатеричное, достаточно разбить его на тетрады и заменить каждую соответствующей цифрой, например:

$$010110100011_2 = 0101 \quad 1010 \quad 0011 = 5A3_{16}$$

Перевод десятичного числа в шестнадцатеричное, аналогичен переводу в двоичную систему:

$$\begin{aligned} 3728/16 &= 233, & \text{остаток} & \quad \mathbf{0} \\ 233/16 &= 14, & \text{остаток} & \quad \mathbf{9} \\ 14 < 16 & & \text{остаток} & \quad \mathbf{14_{10} = E_{16}} \\ 3728_{10} &= 0xE90_{16} \end{aligned}$$

Преимущества шестнадцатеричной системы счисления:

- Наибольшую популярность получили платформы с 8-битным байтом. В таких платформах байт может принимать $2^8=256$ значений в диапазоне от 0 до 255. Число 256_{10} , записанное в шестнадцатеричной системе, является круглым трёхзначным числом – 100_{16} .

- То, что это число является трёхзначным, означает, что для представления любого числа в диапазоне 0—255 требуется не более 2 разрядов. 16 – минимальное основание системы счисления, при котором для записи достаточно двух разрядов. Системы с меньшим основанием потребовали бы уже три (и более) разряда, а возможность уложиться в один разряд обеспечила бы только система счисления с основанием 256, которая совершенно неприменима на практике из-за большого числа цифр в ней.

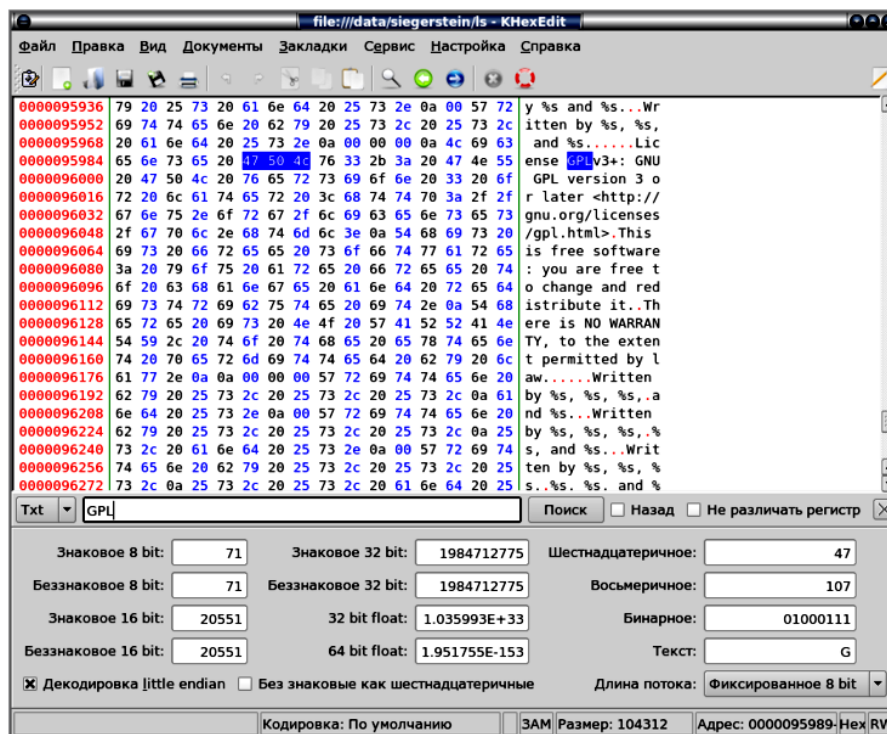
- То, что это число является круглым, означает, что для представления будут использованы все возможные комбинации цифр, и не останется неиспользованных (например, в случае использования десятичной системы, неиспользованными остаются комбинации, соответствующие числам от 256 до 999).

- При использовании шестнадцатеричной системы получается, что правая цифра соответствует младшим четырём битам байта, а левая цифра в записи — старшим 4 битам. Поэтому любые операции над 4-битными блоками, выровненными по границе байта, можно упрощённо производить как операции над соответствующими цифрами.

Очевидно, что, например, для платформ с 9-битовым байтом использовались бы трёхзначные восьмеричные числа.

Для работы с данной системой используют шестнадцатеричные, т.н. Нех-редакторы.

Нех-редактор отображает данные в виде матрицы, каждая ячейка которой соответствует одному байту, записанному в шестнадцатеричной системе счисления в виде двухзначного числа (с ведущим нулём, если он требуется). Количество столбцов матрицы является степенью двойки, чаще всего используются 16 или 8 колонок, иногда 4. Число строк зависит от количества байтов, которые требуется отобразить/отредактировать. В случае использования 16 колонок одна строка соответствует одному параграфу.



Свободный hex-редактор KHexEdit, входящий в состав KDE (kdeutils)

Кроме этого, часто используются дополнительные элементы:

- Слева от матрицы отображается линейка из чисел: каждой строчке соответствует число, означающее адрес/смещение первого байта этой строчки. Шаг адресов при этом равен количеству колонок.
- Сверху от матрицы отображается другая линейка из чисел: над каждой колонкой отображается смещение байта, стоящего в этой колонке, относительно первого байта соответствующей строчки. Сумма числа, соответствующего i -той строке, и числа, соответствующего j -той колонке, является адресом/смещением байта (i, j) , стоящего на пересечении взятой строки и взятого столбца.
- Справа от матрицы могут отображаться те же данные, но в другой интерпретации. Наиболее часто используется альтернативное отображение данных как текста в кодировке ASCII, при этом байты, значения которых соответствуют непечатаемым символам, отображаются как точки. Многие редакторы позволяют выбрать произвольную кодировку для режима отображения текста. Вариант с отображением данных как текста встречается в Нех-редакторах общего назначения. В редакторах, являющихся частью какой-либо специальной программы, например, отладчика, обычно доступны другие режимы, такие как листинг дизассемблирования, подсказки об адресах, и другие специфические для приложения режимы.

Большинство редакторов позволяют совершать редактирование как в левой половине, так и в правой (при её наличии). При этом совершаемые изменения когерентны. Простые редакторы позволяют только менять значение выбранных байтов по отдельности. Продвинутые – редактировать совокупности байтов как цельные блоки, выполнять поиск по последовательностям или шаблонам последовательностей. Нех-редакторы для редактирования образов дисков могут включать в себя функции по восстановлению повреждённой файловой системы или случайно удалённых файлов.

Нех-редакторы используются для редактирования нетекстовых данных, когда специализированной программы для редактирования имеющихся данных либо нет, либо её применение неоправданно или нежелательно. Примером может служить внесение изменений в скомпилированную программу без перекомпиляции последней. Так, с помощью Нех-редактора можно изменить значения жестко вшитых в код программы строковых и числовых констант, если правильно определить их расположение.

Если Нех-редактор имеет функции дизассемблера или дизассемблирование целевой программы уже было проведено с помощью другого инструмента заранее, и известно смещение, можно внести изменение непосредственно в код программы и изменить её поведение.

1.1.4. Смешанные системы счисления

Смешанная система счисления является обобщением b – ричной. Основанием подобной системы является возрастающая последовательность чисел $\{b_k\}_{k=0}^{\infty}$. Числа представляются аналогичными линейными комбинациями. Смешанные системы могут быть степенными (это, собственно, классическая позиционная система), показательными и даже гипероператорными. Наиболее известны:

Факториальная система, где $b_k = k!$, используется при декодировании перестановок списками инверсий. Имея номер перестановки, можно воспроизвести её следующим образом: число, на единицу меньшее номера (нумерация начинается с нуля), записывается в факториальной системе счисления. При этом коэффициент при числе $i!$ будет обозначать число инверсий для элемента $i+1$ в том множестве, в котором производятся перестановки (число элементов меньших $i+1$, но стоящих правее его в искомой перестановке).

Система Фибоначчи, где любое натуральное n однозначно разлагается в сумму

$$\sum_k f_k F_k,$$

где $f_k \in \{0,1\}$, а F_k это k – е число Фибоначчи, при этом в коэффициентах f_k число единиц конечно и не встречаются две единицы подряд. Это утверждение носит название теоремы Цекендорфа (1939).

На основе фибоначчевой системы счисления строится код Фибоначчи – универсальный код для натуральных чисел (1, 2, 3...), использующий последовательности битов. Поскольку комбинация 11 запрещена в фибоначчевой системе, её можно использовать как маркер конца записи. Для составления кода Фибоначчи по записи числа в фибоначчевой системе счисления следует переписать цифры в обратном порядке (так, что старшая единица оказывается последним символом) и приписать в конце ещё раз 1. Вот представление некоторых чисел в этой системе:

Десятичная система	0	1	2	3	4	5	6	7
Фибоначчьева система	0	1	10	100	101	1000	1001	1010
Код Фибоначчи	--	11	011	0011	1011	00011	10011	01011

Существуют системы счисления с отрицательным основанием, называемые негативно-позиционные, или просто нега-позиционные, системы счисления с дробным, иррациональным, трансцендентным и даже комплексным основанием, но их применение в компьютерной технике очень и очень ограничено. В качестве редкого исключения система с $b = 1/3$ позволила бы производить операции умножения и деления на троичных регистрах сдвига. Естественно, такие идеи никогда не были воплощены в металле, так как сложность их чрезмерна и не пропорциональна полученному выигрышу.

Однако в 1960-е годы в Польше несколько инженеров решили разработать нега-позиционный компьютер. Особенностью негативных систем является отсутствие знака перед отрицательными числами и, следовательно, отсутствие правил знаков. Всякое число любой из нега-позиционных систем, отличное от 0, с нечётным числом цифр — положительно, а с чётным числом цифр – отрицательно. Обычно название нега-позиционной системы состоит из префикса нега- и названия соответствующей системы счисления с положительным основанием.

Нега-позиционные системы счисления были впервые предложены Витторио Грюнвальдом в его работе «Giornale di Matematiche di Battaglini», опубликованной в 1885 г. Впоследствии, самостоятельно заново, они были изучены А. Кемпнером (1936), З. Павляком и А. Вакуличем (1959). Это привело к тому, что нега-двоичная система счисления была реализована в польском компьютере BINEG, построенном в 1957-59 гг. и основанном на идеях З. Павляка и А. Лазаркевича из Математического института в Варшаве.

Память BINEG на магнитном барабане содержала 512 слов 36-битной длины. Машина имела очень простую структуру шины. Все регистры, кроме регистров управления, были помещены в барабан. Общее число электронных ламп было около 350. Компьютер использовал оригинальную нега-двоичную арифметику, и 34-битное число x представлялось как

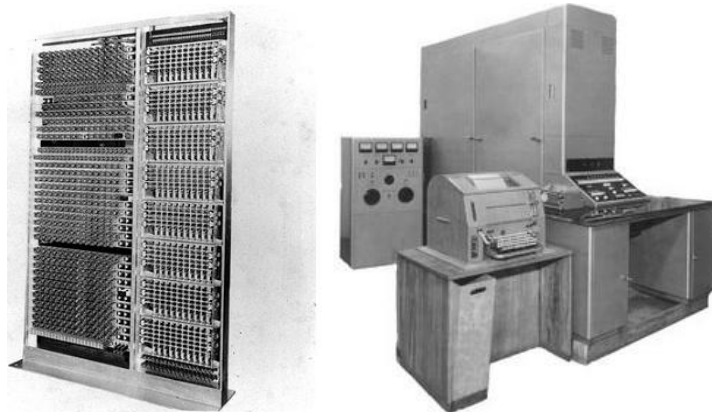
$$x = \sum_{k=0}^{33} a_k (-2)^k,$$

Ассемблер его был очень прост и состоял всего из 20 команд. BINEG какое-то время использовался для обучения студентов в Варшавском Техническом Университете, дальнейшая его судьба неизвестна.

Известно, однако, что Здислав Павляк на этом не остановился и в 1962 и 1964 построил с помощью компании ELWRO еще 2 нега-двоичных компьютера – ламповый UMC-1 и транзисторный UMC-10, названные так по сокращению ЭВМ на польском (Uniwersalna Maszyna Cyfrowa). UMC-10 был достаточно удачным, обладал приличным для тех лет быстродействием, был построен в количестве 25 штук и в силу простоты программирования и дешевизны активно использовался вузами и НИИ для работы.

Нельзя сказать, что нега-двоичная система так уж подходит для создания ЭВМ – какие-то операции в ней выполняются проще, какие-то сложнее⁶, но машины на ее основе чрезвычайно просты и дешевы. Тем более, польская математическая школа всегда славилась небанальными идеями, вспомним хотя бы открытие Лукасевичем многозначных логик или бесскобочную польскую запись. Так и здесь – поляки остроумно и виртуозно решили проблему, создав по тем временам выдающийся и очень необычный компьютер. Нельзя сказать, насколько такая машина была бы актуальна сейчас, разве что для развивающихся стран, но получить за малые деньги неплохую производительность было бы можно.

Поляков смог переплюнуть лишь легендарный Джордж Штибиц (George Stibitz), построивший в 1940 г. в Bell Labs Complex Number Calculator, который оперировал комплексными числами, причем мнимая и действительная части хранились в машине отдельно. CNC был изготовлен из 450 телефонных реле и мог умножать и делить восьмизначные числа. Последний из серии – CNC Mk VI был построен в 1950 г. и еще десять лет работал в Технологическом институте в Бихаре (Индия).



Машина Штибица (слева) и UMC-1 (справа)

1.1.5. Машина Фибоначчи

Не все попытки создать экзотический компьютер были успешны, более того, некоторые из них больше походили на техническую шизофрению. В качестве назидания полезно рассмотреть один такой случай. В 1957 г. Бергман предложил очень сложную систему счисления с иррациональным основанием в виде константы золотого сечения:

$$b = \varphi = \frac{\sqrt{5} + 1}{2} \approx 1.618033 \dots$$

Несложно показать, что любое рациональное $x \in [0,1]$ допускает однозначное разложение в ряд через отрицательные степени φ :

$$x = \sum_{k=-1}^{\infty} \varepsilon_k \varphi^k,$$

где $\varepsilon_k \in \{0,1\}$ и обладает фибоначчиевым свойством отсутствия соседних единиц.

Таким образом, системы Бергмана и Фибоначчи являются в каком-то смысле обобщением двоичной системы и около 1970-х г. некоторые инженеры задумались – нельзя ли разработать соответствующую модель в железе и не принесет ли это прирост производительности.

Преимущество системы Бергмана заключается в том, что определенная группа иррациональных чисел представляется в этой системе в виде конечных комбинаций нулей и единиц, не требуя округления. Также коды Фибоначчи и коды Бергмана обладают естественной избыточностью, которая может быть использована для целей контроля преобразований. Эта избыточность проявляет себя в свойстве неоднозначности представлений числа, например, 19 в коде Фибоначчи представляется так:

$$19 = 1001101 = 1010001 = 1010010 = 0111101.$$

При этом различные кодовые представления одного и того же числа могут быть получены одно из другого с помощью специальных фибоначчиевых операций – свертки ($011 \circledast 100$) и развертки ($100 \circledast 011$). Если над кодом выполнить все возможные свертки, мы приходим к специальному представлению, называемому минимальной формой, в которой не встречается рядом двух единиц, такое представление уже однозначно по теореме Цекендорфа. Если, наоборот, проделать все операции развертки, мы приходим к симметричному представлению – максимальной форме, в которой не встречается двух нулей подряд.

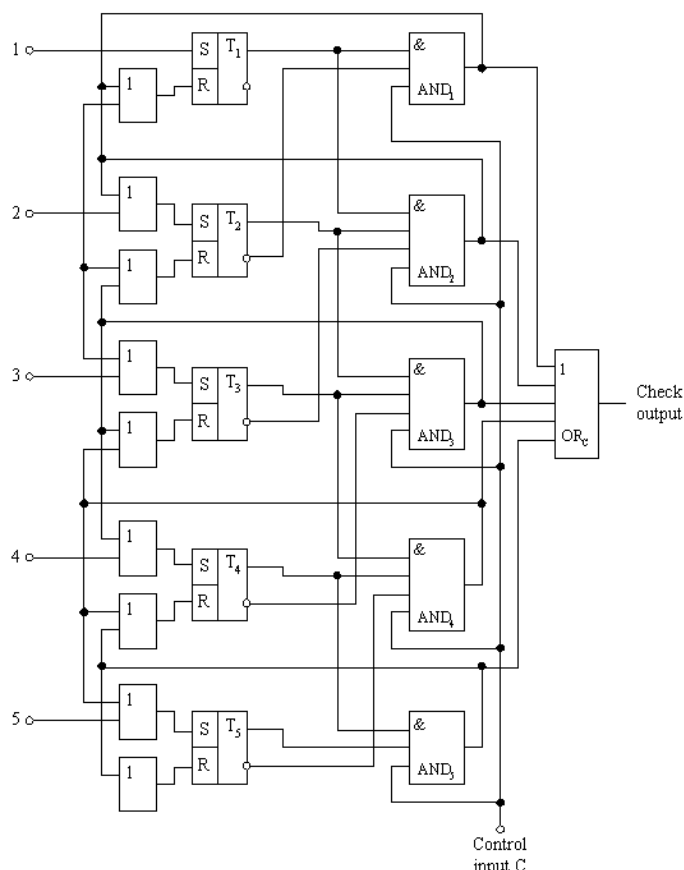
В 1986 г. в Виннице (Украина) А. П. Стахов, бывший заведующий кафедрой информационно-измерительной техники Таганрогского радиотехнического института, был назначен директором конструкторско-технологического бюро «Модуль», входящего в состав Винницкого политеха. А. Стахов много лет

⁶ Детали можете изучить в «Г. Уоррен, мл., Алгоритмические трюки для программистов, Вильямс, 2007 г.» или «Кнут Д.Э Искусство программирования. Том 2. Получисленные алгоритмы, 2000 г.»

увлекался золотым сечением и всем, что с ним связано, и предложил разработать на новых принципах фибоначчьев процессор. На разработки было выделено около 15 млн. рублей – значительная для того времени сумма, вдвое превышавшая годовой объем финансирования научно-исследовательских работ всего Винницкого политехнического института.

Заказчиком, в качестве которого выступала головная компьютерная организация Министерства общего машиностроения, было сформулировано три основных направления научных и опытно-конструкторских разработок: проектирование самоконтролирующегося процессора Фибоначчи для специальных применений; проектирование самокорректирующихся фибоначчьевых ЦАП-АЦП высокой точности и метрологической стабильности; проектирование самосинхронизирующихся систем цифровой магнитной записи и волоконно-оптических систем передачи информации. К этим разработкам в ВПИ было подключено около 200 сотрудников.

Естественно, для начала понадобилось разработать элементную базу. В качестве таковой послужили обычные двоичные триггеры, на которых был собран регистр, выполняющий свертку⁷:



Этот Ф-регистр, по сути, остался единственным изобретением бюро «Модуль». Середина 1980-х была уже относительно свободным временем, так что на него было получено много патентов, в т.ч. и зарубежных, миллионы утекали в неизвестном направлении, а полноценный Ф-компьютер так и не появился.

Если схема регистра кажется вам весьма усложненной, по сравнению с каким-нибудь тривиальным компаратором кодов или шифратором – не переживайте, вы не ошиблись, это действительно так. Ф-регистр устройство нетривиальное, но это только начало, ведь именно из таких регистров планировалось собирать Ф-сумматоры.

Почему все так ужасно, спросите вы? Давайте рассмотрим в качестве примера, процедуру типичного Ф-сложения. Пусть у нас имеется два числа в виде Ф-кодов: 31 = 10011011 и 22 = 01011010.

Первый шаг состоит в приведении кодов к минимальной форме:

$$31 = 100\underline{11} \underline{011} = 10100100$$

$$22 = 01\underline{011}010 = \underline{011}00010 = 10000010$$

Второй шаг состоит в формировании многоразрядной промежуточной суммы

S_1 и многоразрядного переноса C_1 в соответствии с таблицей Ф-сложения:

$$31 = \underline{1}0100100 +$$

$$22 = \underline{1}0000010 =$$

$$S_1 = \underline{11}100110$$

$$C_1 = 00\underline{1}00000$$

Заметим, что в этом случае ситуация 1 + 1 возникает для старших разрядов Ф-представлений. Эти разряды и возникающие при этом преобразования разрядов в промежуточной сумме S_1 и промежуточном переносе C_1 подчеркнуты. Третий шаг состоит в приведении промежуточной суммы S_1 к минимальной форме: $S_1 = \underline{011100110} = 100101000$

Четвертый шаг состоит в сложении S_1 и C_1 :

$$S_1 = 100\underline{101000} +$$

$$C_1 = 000\underline{100000} =$$

$$S_2 = 100\underline{111000}$$

$$C_2 = 00000\underline{1000}$$

⁷ Смысл этой схемы читатель легко уяснит, ознакомившись ниже с основами комбинаторной логики. Конечно, такой порядок изложения может показаться странным, но не привести здесь это примечательное устройство было бы совершенно неуместным, и далее станет ясно, почему.

Разряды 1 + 1 и возникающие в результате их сложения преобразования разрядов в S_2 и C_2 подчеркнуты. Пятый шаг состоит в приведении промежуточной суммы S_2 к минимальной форме:

$$S_2 = 100\underline{11}1000 = 101001000.$$

Шестой шаг состоит в сложении S_2 и C_2 :

$$S_2 = 10100\underline{1}000 +$$

$$C_2 = 00000\underline{1}000 =$$

$$S_3 = 10100\underline{11}00$$

$$S_3 = 10100\underline{11}00$$

Седьмой шаг есть приведение промежуточной суммы S_3 к минимальной форме:

$$S_3 = 10100\underline{11}00 = 101010000.$$

Восьмой шаг состоит в сложении S_3 и C_3 :

$$S_3 = 101010000 +$$

$$C_3 = 000000010 =$$

$$S_4 = 101010010$$

$$C_4 = 000000000$$

Сложение закончено, поскольку на восьмом шаге возникает нулевой перенос. Искомый результат $S_4 = 101010010$. Зачем мы привели здесь эту схему? Исключительно с целью показать, что мир устроен таким образом, что нельзя выиграть все и сразу – невозможно создать систему кодов, которая одновременно исправляла бы ошибки, была минимальной и при этом удобной в использовании. Весь вопрос в том, чем мы жертвуем и, в случае Ф-процессора, жертвы оказались слишком велики – ведь мы не дошли даже до умножения!

Дальнейшая история Ф-процессора печальна и поучительна. Разработка полной микросхемы такого уровня сложности представляла бы серьезные проблемы даже для Intel. Их процессор 80386, первый, который в СССР не смогли даже скопировать, не то что разработать самим (для серии ЕС-1863 даже пришлось закупить оригинальные процессоры на Западе), был одним из самых сложных на тот период, но Ф-машина была бы на порядок сложнее. Пятнадцать миллионов на разработку разлетелись незаметно, А. Стахов попытался запросить еще, причем в валюте, для закупки хотя бы простейшей САПР из-за рубежа, так как все наши системы не были рассчитаны на такой уровень проектирования, но тут настал 1990, и проект умер вместе с Союзом.

Судьба А. Стахова, однако, сложилась намного лучше – для начала он попытался заинтересовать своим процессором известные компании, но, несмотря на «65 международных патентов США, Японии и Германии на Ф-регистр», этого оказалось недостаточно⁸. Традиционные процессоры уже тогда были на порядок дешевле и лишь незначительно медленнее этого монстра. Не найдя работы ни в Европе, ни в США, ни в России, Стахов уехал в Ливию, в Университет Триполи, затем в Мозамбик, но и там покупателей на его регистр не нашлось. От огорчения он ударился в мистику, стал активным членом лженаучной «Академии Тринитаризма», пропагандирующей «мистические свойства Золотого Сечения и путь через него к Математике Гармонии». Затем он эмигрировал в Канаду, где благополучно пребывает и сейчас, участвует в конференциях искателей «Мировой Золотой Гармонии», регулярно получает от них всякие почетные медали и дипломы, в т.ч. звание «Доктор Божественной Геометрии в Математике», и сетует, какой проект загубило глупое руководство. Он утверждает до сих пор, что если бы кто-нибудь выделил ему еще миллионов 20 долларов, то Ф-процессор доделали бы.

Таким образом, не все, что потенциально, вообще может быть воплощено в железе и стоит такового воплощения, этот урок предлагается читателю в целях предостережения против своеобразной технической шизофрении в которую, иногда, склонны впадать люди. Для справедливости стоит отметить, что сами коды Фибоначчи являются удобным решением для цифровой обработки сигналов в реальном времени, и программные пакеты, включающие такие возможности, стандартно поставляются уже много лет со многими DSP, например, Analog Devices ADSP-2189M. При этом эмуляция Ф-кодов на обычной аппаратной части не вызывает ни проблем, ни нареканий по производительности.

1.1.6. Двоично-десятичное кодирование

Далеко не таким маргинальным, но достаточно экзотическим является т.н. двоично-десятичное кодирование (binary-coded decimal, BCD). Это класс бинарных кодов, в котором каждый разряд целого десятичного числа записывается в виде его двоичного кода, обычно четырехбитного, но существовали и шести- и восьмибитные коды.

Главным достоинством BCD является отсутствие потерь точности на округление при переводе двоичных дробей в десятичные, и наоборот, упрощение операций умножения, деления и округления, а также легкость ввода с клавиатуры и вывода на индикацию. По сравнению с классической двоичной системой, основ-

⁸ Может быть дело в том, что теоретическая возможность построения такого устройства впервые была показана еще в 1976 П. Монтейро и Р. Ньюкомбом (Университет Мэриленд) в статье «Minimal and maximal Fibonacci Representations: Boolean Generation», The Fibonacci Quarterly (№ 1, 1976), однако, авторы рассматривали это лишь как забавный технический курьез.

ными недостатками являются небольшое увеличение сложности схемы, необходимое для аппаратной поддержки основной арифметики, и немного менее плотное хранение информации.

BCD была использована во многих ранних десятичных компьютерах, таких, как ENIAC. Хотя сейчас ее применение не столь широко, но BCD-коды до сих пор играют важную роль в финансовых и промышленных вычислениях, для минимизации ошибок округления, правда, работа с ними уже происходит только на программном, а не на аппаратном уровне.

Вот как происходит BCD-преобразование:

Десятичное число	4-х битный BCD-код, разложение через 8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Таким образом, для перевода числа из BCD в десятичную систему достаточно сделать следующее. Сначала нужно разбить произвольное число на цифры, например, 1001011101100010 на блоки по 4 бита: 1001 0111 0110 0010, затем найти каждую цифру, умножив вес соответствующей позиции на 0 или 1 и сложив результат:

$$1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 9$$

$$0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 7$$

$$0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 6$$

$$0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1$$

Так получаем число 9761.

Одна из разновидностей 6-битный BCD-код (alphanumeric BCD, ABCD - буквенно-цифровой двоично-десятичный код, BCD Interchange Code, BCDIC - код обмена) использовался как в десятичных (IBM 1620, IBM 1400), так и в недесятичных (IBM 700/7000) мейнфреймах IBM, в суперкомпьютерах CDC 1604 и огромном количестве других машин 1950-1970 гг. - Burroughs, Bull, General Electric, NCR, Siemens, Sperry-UNIVAC и др. Такие знаменитые машины, как IBM S/360 и DEC VAX-11, также его поддерживали.

Совсем экзотическим видом было десятично-пятеричное кодирование (Bi-quinary coded decimal, BQCD) с основанием 5. IBM 650 выводил на своей передней панели информацию в виде BQCD- индикации. В настоящее время BCD использует BIOS многих персональных компьютеров для хранения даты, эта традиция идет от оригинальной микросхемы часов MC6818 компьютеров IBM PC/AT. Семейство 8-битных компьютеров Atari на MIPS-процессорах использовало BCD для повышения точности вычислений с плавающей запятой, хранение даты в таком формате применялось даже в ранних версиях Sony PlayStation 3. Сейчас разработаны многие более эффективные версии BCD, такие как Chen-Ho coding и Densely coding. Из отечественных оригинальных разработок можно отметить серию машин «МИР» единственную, использовавшую BCD.

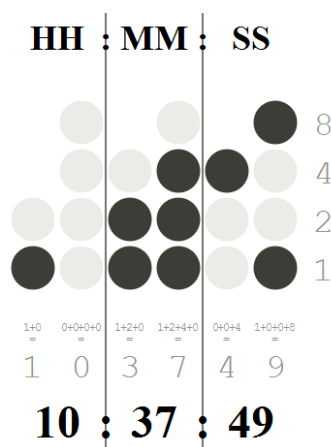


Передняя панель IBM 650

В 60-е годы в СССР также пользовались широкой популярностью идеи т. н. машинной арифметики в остаточных классах. Идея такой системы заключается в том, что числа в ней представляются остатками от

деления на выбранную систему оснований – взаимно простых чисел. Толчком к исследованиям в этой области послужили опубликованные в 1955-1957 гг. работы чешских ученых М. Валаха и А. Свободы. Концепция была подхвачена советскими учеными И.Я. Акушским и Д.И. Юдицким, опубликовавшим в 1968 г. книгу «Машинная арифметика в остаточных классах».

На вопрос о побудительных мотивах создания применения СОК в компьютерах И.Я. Акушский и Д.И. Юдицкий ответили просто: «Следует отметить, что позиционные системы счисления, в которых представляется и обрабатывается информация в современных вычислительных машинах, обладает существенным недостатком – наличием межразрядных связей, которые накладывают свой отпечаток на способы реализации арифметических операций, усложняют аппаратуру и ограничивают быстродействие. Поэтому естественно изыскание возможностей такой арифметики, в которой бы поразрядные связи отсутствовали. Оказалось, что такая арифметика может быть построена на базе непозиционной системы счисления, в частности системы остаточных классов». И. Акушский и Д. Юдицкий были



Один из вариантов часов с BCD-индикатором для настоящего энтузиаста компьютеров

не только теоретиками, но и принимали активное участие в реализации компьютера на основе СОК еще в 1957 г.

Однако такая система оказалась трудной в использовании – она была ограничена лишь полем целых положительных чисел, детектирование переполнения в ней представляет изрядную проблему, вообще любые операции, требующие знания обо всем числе сразу – такие как деление и сравнение выполняются по весьма ресурсоемким алгоритмам. И хотя авторам книги «Машинная арифметика в остаточных классах» удалось продвинуться в преодолении этих трудностей, но полностью разрешить их не удалось и революции на основе СОК не получилось. Среди реализованных полностью или частично проектов, естественно для военных и других целей, в СССР производительных компьютеров не было вплоть до 1980-х г.: из машин на основе СОК можно отметить 5Э53, проект КБ «Алмаз», ЭВМ 41-50 «Лидер», Т-340А и К-340А. В настоящее время разработки СОК ведутся достаточно вяло, дальше нескольких диссертаций дело не пошло. Причина, в общем та же – внедрять новую, достаточно сложную систему ради сомнительной выгоды слишком дорого, тем более что и традиционные системы работают не хуже.

1.2. Теория информации

1.2.1. Информация и энтропия

Чтобы разобраться с тем, что измеряется в битах и байтах, мы должны, прежде всего, определить само понятие «информация». Ее хранят, накапливают, передают и обрабатывают, для чего, собственно, и были построены компьютеры, однако очень мало кто серьезно задумывается – что же это такое?

В сороковых годах информатики задумались над разработкой способов хранения данных, при котором пространство, предназначенное для хранения (в то время – очень дорогостоящая память первых ЭВМ), будет расходоваться оптимально. Клод Шеннон, изучая нюансы различий между семантикой (что некая сущность значит) и синтаксисом (как некая сущность выражается), разработал большинство базовых понятий математической теории коммуникации, позже названной теорией информации.

Понимание того, что одно и то же значение (семантика) может быть реализовано различными способами (синтаксис), приводит к закономерному вопросу: «Какой способ выражения чего-либо является оптимальным, в смысле наиболее экономичным?» Поиск ответа на этот вопрос привел Шеннона к мысли об энтропии, которая соотносится с количеством содержащейся в файле полезной информации.

Первое, над чем задумался Шеннон, что такое информация/данные с математической точки зрения. До этого, да и теперь еще, на этот вопрос пытались ответить разнообразные науки. Приведем несколько «канонических определений»:

- В РФ правовое определение понятия «информация» дано в федеральном законе от 27 июля 2006 года № 149-ФЗ «Об информации, информационных технологиях и о защите информации» (Статья 2): «информация – сведения (сообщения, данные) независимо от формы их представления».

- С точки зрения философии информация существует вне нашего сознания, и может иметь отражение в нашем восприятии только как результат взаимодействия: отражения, чтения, получения в виде сигнала, стимула. Информация не материальна, как и все свойства материи. С материальной же точки зрения информация – это порядок следования объектов материального мира.

- С точки зрения Wikipedia объективная информация – свойство материальных объектов и явлений (процессов) порождать многообразие состояний, которые посредством взаимодействий (фундаментальные взаимодействия) передаются другим объектам и запечатлеваются в их структуре.

- Даже сам отец кибернетики Норберт Винер отвечал на этот вопрос так: «Информация – это не материя и не энергия, информация – это информация». Однако основное определение информации, которое он дал в книге «Кибернетика, или управление и связь в животном и машине» и повторил в «Кибернетика и общество», более развернуто: «Информация – это обозначение содержания, полученное нами из внешнего мира в процессе приспособления к нему нас и наших чувств».

Что объединяет для математика все эти понятия? Конечно их полная и абсолютная непригодность. Как и многие юридические формулировки, первая является полнейшей глупостью, определяющей вообще ничего – информация это сведения, сведения это информация. Второе определение, по здравом размышлении, также не годится, ибо вместо ответа на вопрос «Что такое информация» отвечает на вопрос «Каковы ее свойства», что касается порядка следования объектов, то неплохо для начала определить, что такое порядок и что считать порядком, а что нет. Попытки уточнить это приводят определение номер два вслед за предыдущим. Третье немного ближе к сути, но настолько запутано и зубодробительно, что мало пригодно для наших целей.

Давайте пойдем проверенным путем, и для начала определим, чем информация точно не является. Определенно можно утверждать, что чем сложнее поведение чего-либо, тем с меньшей вероятностью мы можем это поведение предугадать. Несложно догадаться, какой стороной может упасть монета, но оценить, из-за какой именно сгоревшей лампы завис ENIAC, если их 17.000, несколько труднее.

Впервые над математической мерой нашего невежества задумался Клод Шеннон в середине 1930-х гг. Результатом его изысканий стала характеристика нашего незнания, пропорциональная числу возможных вариантов угадывания различных событий – чем их больше, тем меньше мы знаем, какое из них произойдет. Эта характеристика получила название энтропия. Математически ее можно описать так. Допустим, у нас есть кубик, который может равновероятно упасть на одну из шести граней. Значит, пока мы не посмотрели на кубик, мы не можем сказать точно, какой стороной он упал, и у нас на выбор шесть вариантов. Мы можем утверждать, в определенном смысле слова, что неопределенность данной системы равна 6. Так мы, очевидно, сможем сравнить неопределенности монетки (2) и кубика (6) и выяснить, что последний – более хаотичен. Значит, когда нам сообщают, какой стороной упал кубик, и какой – монетка, мы, в случае кубика избавляемся от большей неопределенности.

Итак, у нас есть первый кандидат на математическое определение неопределенности: неопределенность данного события равна количеству возможных его исходов. Тогда, например, неопределенность алфавита из 33 букв равна 33. Но что-то здесь все равно не так, и вот что.

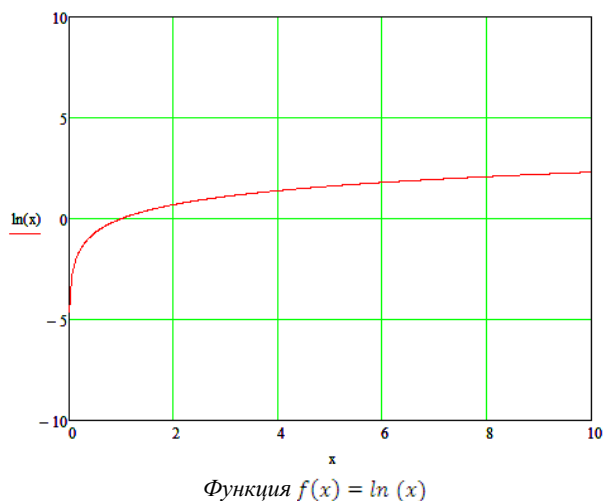
Первая проблема – как учесть в определении, что, например, в реальном неопределенном сообщении вероятность встретить конкретную букву неодинакова. Например, на этой особенности построены игры в слова, когда подсказка распространенной буквы – например «а» или «о» несет меньше информации о загаданном слове, чем наличие в нем редкой буквы, типа «ф» или «ц».

Далее, определение дает нам возможность ввести минимальную единицу неопределенности – логично, что минимально не определен выбор всего из двух вариантов, но как выразить в элементарных бинарных выборах все остальные? Скольким например бинарным неопределенностям соответствует выбор из трех вариантов, десяти, тридцати?

Нам поможет тот факт, что логичным было бы потребовать, чтобы для бросания k независимых m -гранных костей неопределенность была в k раз больше (т. к. общее число возможностей для них m^k), а при бросании, допустим, n -гранной и m -гранной костей вместе суммарная неопределенность нашлась, как $n + m$ (т. к. число возможностей для них $n + m$). Отсюда вопрос, какая функция $f(x)$ удовлетворяет свойствам:

- $f(x \cdot y) = f(x) + f(y)$
- $f(m^k) = k \cdot f(x)$

Конечно, эта функция – логарифм по произвольному основанию.



Величина $H = C \cdot \log_n m$, где m - число состояний системы, а C - произвольная константа и называется энтропия или мера хаотичности системы, как это и предложил Шеннон в статье «A Mathematical Theory of Communication» (1948). Основанием логарифма может являться произвольное число n , в зависимости от этого приняты различные единицы измерения неопределенности. Шеннон не был первым, кто задумывался о сущности неопределенности информации и определении ее количества. В 1865 г. немецкий физик Рудольф Клаузиус ввел в статистическую физику понятие энтропии или меры уравниваемости системы. В 1921 г. основатель большей части математической статистики англичанин Роналд Фишер впервые ввел термин «информация» в математику, но полученные им формулы носят очень специальный характер. Первый шаг на пути к современному пониманию энтропии сделал в 1928 г. Хартли. Основной полученный им результат можно сформулировать примерно так: если в заданном множестве, содержащем N элементов, выделен некоторый элемент x , о котором известно лишь, что он принадлежит этому множеству, то, чтобы найти x , необходимо получить количество информации, равное $\log_2 N$. Эту формулу обычно называют формулой Хартли.

Изменим немного терминологию, как то принято в теории информации. Вместо монеты или кости источником неопределенности и информации будем считать алфавит – набор символов, от минимального, в один знак, до сколь угодно большого. Количество символов алфавита, как и количество граней кости, определяет базовую неопределенность одного символа. Вместо последовательности бросков рассмотрим последовательность выборов символов из алфавита – случайную или, как в реальности, не очень, такую последовательность будем называть сообщением. В дальнейшем нам необходимо научиться находить энтропию сообщения, его информацию и выбирать такое представление сообщения, при котором его объем минимален, а информация максимальна, что и является основной задачей теории информации.

Формула Хартли является частным случаем более общей формулы Шеннона, позволяющей найти количество информации в случайном сообщении фиксированного алфавита. Выбор основания логарифма оп-

ределяет минимальную систему, которую мы будем использовать для измерения хаотичности. Система с числом состояний 0 не существует, с числом состояний 1 (постоянное) не подходит, ведь она полностью детерминирована (измениться в ней ничто не может), естественным минимумом будет 2. Естественно, нецелые значения, $1 \leq n \leq 2$, также не подходят, ибо не существует системы с нецелым числом состояний.

К. Шеннон в том же 1948 г. предложил использовать слово bit для обозначения наименьшей единицы неопределенности – системы с двумя состояниями. Термин энтропия использовался Шенноном по совету легендарного Джона фон Неймана, отметившего, что полученные Шенноном для теории связи формулы для ее расчета совпали с соответствующими формулами статистической физики, а также то, что «точно никто не знает», что же такое энтропия.

Клод Шеннон вывел, что энтропия имеет следующие свойства:

- непрерывность; изменение значения величины вероятности на малую величину должно вызывать малое результирующее изменение меры незнания;
- монотонность; в случае, когда все варианты равновероятны, увеличение количества вариантов должно всегда увеличивать значение меры;
- аддитивность; в случае нескольких последовательных уточнений итоговое незнание должно равняться сумме незнаний на каждом шаге.

Пусть у нас имеется x событий, каждое из которых может иметь n исходов с вероятностью p_n соответственно. Тогда энтропия имеет следующие свойства:

- $H(p_1, p_2, \dots, p_n)$ определена и непрерывна для любых $p_1, p_2, \dots, p_n$, причем $\sum p_i = 1$.
- Для целых положительных n

$$H\left(\frac{1}{n}, \dots, \frac{1}{n}\right) < H\left(\frac{1}{n+1}, \dots, \frac{1}{n+1}\right)$$

- Для целых положительных b_i , таких, что $b_1 + \dots + b_k = n$

$$H\left(\frac{1}{n}, \dots, \frac{1}{n}\right) = H\left(\frac{b_1}{n}, \dots, \frac{b_k}{n}\right) + \sum_{i=1}^k \frac{b_i}{n} H\left(\frac{1}{b_i}, \dots, \frac{1}{b_i}\right)$$

Средняя энтропия для набора несвязанных между собой n событий, вероятности наступления которых соответственно равны p_n , найдется как:

$$H = - \sum_{i=1}^n p_i \log_k p_i$$

К биту мы будем приводить меру хаоса любой системы. Минимальная мера хаоса очевидна – минимальная неопределенность возникает в бинарном вопросе, подразумевающим 2 варианта ответа. Поэтому за эталон обычно берут двоичную неопределенность и говорят, что неизвестность из двух вариантов – это минимальная энтропия (обозначается один бит).

Например, для монетки энтропия равна

$$- \sum_{i=1}^2 \frac{1}{2} \log_2 \frac{1}{2} = -2 \frac{1}{2} \log_2 \frac{1}{2} = 1 \text{ бит}$$

Для кубика, очевидно,

$$- \sum_{i=1}^6 \frac{1}{6} \log_2 \frac{1}{6} = -6 \frac{1}{6} \log_2 \frac{1}{6} = 2.585 \text{ бита}$$

Что же такое информация? Очевидно, это энтропия с обратным знаком, т.е. сообщая n бит информации, мы уменьшаем неизвестность на n бит.

Допустим, на экзамене у вас есть шанс 0.1 получить оценку 2, 0.2 получить 3, 0.5 получить 4 и 0.2 получить 5. Тогда сообщив вашу оценку родителям, вы передадите им

$$0.1 * \log_2 0.1 + 0.2 * \log_2 0.2 + 0.5 * \log_2 0.5 + 0.2 * \log_2 0.2 = 1.761 \text{ бита информации.}$$

Отсюда недалеко и до информации сообщения. Алфавит – конечное множество. Буквы – элементы множества. Произвольная последовательность букв – сообщение.

Математическая теория информации изучает следующие вещи:

- Какой алфавит (т.е. система кодирования) обладает наивысшей информационной емкостью и удобством.
- Как оптимально кодировать информацию.
- Как разработать избыточные помехоустойчивые коды.
- Как анализировать количество информации в тексте на естественном и искусственном языке.

Алфавит может иметь вероятностное распределение, далекое от равномерного. Если исходный алфавит содержит n символов, тогда его можно сравнить с эталонным алфавитом, вероятностное распределение которого равномерно. Соотношение энтропии исходного и оптимизированного алфавита – это эффективность исходного алфавита, которая может быть выражена в процентах. Эффективность исходного алфавита с

n символами может быть также определена как его n -арная энтропия, она ограничивает максимально возможное сжатие без потерь, которое может быть реализовано при использовании кодирования – Хаффмана, LZW, арифметического кодирования и т.п.

Если следование символов алфавита не зависит, количество информации, которую несёт последовательность таких символов (а, следовательно, и энтропия), очевидно, меньше. Для учёта таких фактов используется условная энтропия. Условной энтропией первого порядка называется энтропия для алфавита, где известны вероятности появления одной буквы после другой (то есть, вероятности двухбуквенных сочетаний):

$$H_1 = - \sum_{i=1}^n p_i \sum_{j=1}^m p_i(j) \log_k p_i(j)$$

Здесь i – состояние, зависящее от предшествующего символа, $p_i(j)$ – вероятность того, что следующим символом будет j , если i был предыдущим. Например, для русского языка без буквы «ё»:

$$H_0 = 5 \quad H_1 = 4.385 \quad H_2 = 3.52 \quad H_3 = 3.01$$

Единица измерения информации, как мы уже упоминали, зависит от основания логарифма. Минимальной единицей является бит – ответ на вопрос из двух вариантов. Если вариантов три – мы получим трит – который уже встречали в обсуждении троичных ЭВМ. Если основание системы счисления равно 10, то такую единицу называют хартли или децит. Наконец, в случае трансцендентного основания e единица измерения носит название нат. Вот соотношение между этими единицами:

$$-\log_2 \frac{1}{2} = 1 \text{ бит} \approx 0.69 \text{ ната} \approx 0.63 \text{ трита} \approx 0.3 \text{ хартли}$$

Таким образом, одна ячейка памяти ENIAC вмещает примерно в три раза больше информации, нежели ячейка обычной двоичной машины.

1.2.2. Кодирование и теоремы Шеннона

Очевидно, чем больше алфавит, тем больше информации несет один символ, зато количество символов увеличивается. Например, в китайской письменности каждый иероглиф обозначает отдельный слог и отдельную морфему. Общее количество иероглифов превышает 80 тысяч, но большую часть из них можно встретить лишь в памятниках классической китайской литературы.

- Знание 500 наиболее частотных иероглифов достаточно для понимания 80% обычного современного китайского текста.

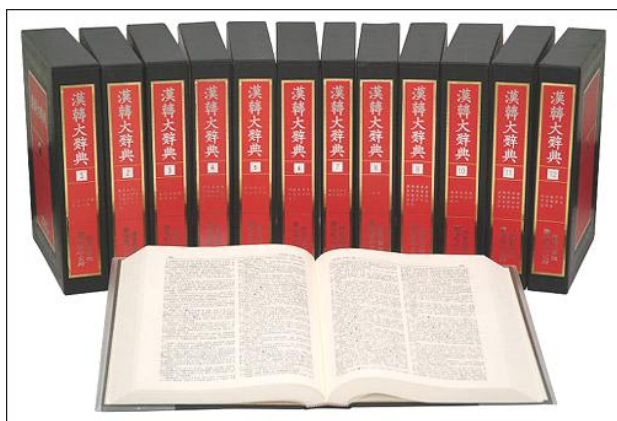
- Знание 1000 знаков позволяет понять 91% текста.

- Знание 2400 знаков позволяет понять 99% текста.

- Знание 3000 иероглифов достаточно для полного чтения неспециализированной литературы.

- Большие однотомные двуязычные словари включают, как правило, 6000-8000 иероглифов. Среди этого объема уже немало весьма редко используемых иероглифов, например используемые в названиях ритуальных предметов древности или медикаментов традиционной китайской медицины.

- Наиболее полный словарь иероглифов Zhōng huá zì hǎi («Море китайских иероглифов») издания 1994 года содержит 85568 иероглифов, а вышедший в 2004 на Тайване Yìtǐ zì diǎn – 106230. Корео-китайский словарь Han-Han Dae Sajeon, завершённый в Южной Корее в 2008 году, содержит 53667 иероглифов, состоит из 16 томов, его создание заняло больше 30 лет и потребовало инвестиций порядка 25.000.000\$ и участия 200.000 человек.



Han-Han Dae Sajeon

Современная молодежь Китая для упрощения набора неофициальных текстов разработала специфическую систему кодирования – цифровые коды – замещение иероглифов цифрами в чате, при написании SMS и в других случаях, в основном посвящены любовной тематике. Например, очень популярен код 520 («я люблю тебя»), который часто встречается в названиях сайтов. Также распространены и нейтральные коды, например, 94 («согласен»). Цифровые коды используют присущую китайскому языку развитую омонимию.

Вот некоторые популярные коды:

01925 «люби меня как прежде»

045692 «ты мой самый любимый»

20110 «буду любить тебя сто миллионов лет»

58 «спокойной ночи»,

... а это отрывок разговора из продвинутого чата:

98!
07868?
57386
0748
39
881!::))

Напротив, самый простой код – бинарный, состоит лишь из 2-х символов 0 и 1. Очевидно, что длина сообщения в такой кодировке максимальна, зато алфавит минимален. Вопрос выбора оптимального соотношения длины алфавита и сообщений и простоты кодирования в общем случае нетривиален.

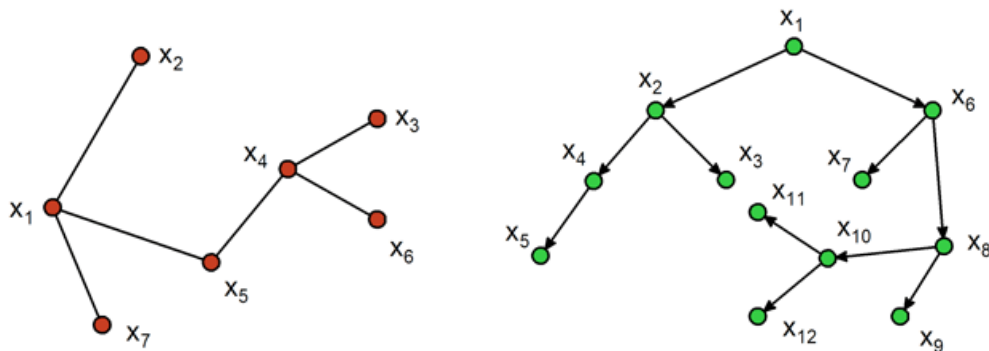
Поскольку, как уже упоминалось, базовый алфавит (называемый кодируемым) присутствует всегда, задача сводится к выбору подходящего кодирующего алфавита, в который будет переведено наше сообщение. При этом стоит учесть – под кодируемым алфавитом можно понимать вообще все возможные символы, например, языка, а можно лишь те, что встречаются в сообщении. Первый подход позволяет придумать код один раз для полного алфавита и использовать его всегда, при этом он, возможно, не будет оптимальным. Второй подход позволяет писать код, оптимальный для конкретного сообщения, но требующий передачи информации об алфавите.

Схемой алфавитного кодирования (или просто алфавитным кодом) называется любое отображение символов кодируемого алфавита в слова кодирующего алфавита, которые называют кодовыми словами. Пользуясь схемой кодирования, каждому слову кодируемого алфавита можно сопоставить его код – конкатенацию кодовых слов, соответствующих каждому символу этого слова. **Конкатенация** (сцепление) – операция склеивания объектов линейной структуры, обычно строк. Например, конкатенация слов «микро» и «мир» даст слово «микромир».

Процесс восстановления сообщения из комбинации символов называется декодированием. Код называется **разделимым** (или **однозначно декодируемым**), если никаким двум словам кодируемого алфавита не может быть сопоставлен один и тот же код.

Отличие кодирования от перевода в том, что кодирование не меняет язык и лексику, а лишь ее представление – например, запись 你愛不愛我 как 02825 – это код, а как «ты любишь меня?» – перевод.

Для наглядного описания кодов используются кодовые деревья. Дерево – частный случай графа – совокупности непустого множества вершин и множества пар вершин.



Кодовые деревья

Теперь немного формальности. Кодирование информации — отображение слов из одного алфавита в другой с целью выбрать оптимальный из них с точки зрения информационной емкости и устойчивости к помехам. Пусть U – исходный алфавит, $u \in U$ – слово алфавита, Z – кодирующий алфавит, превосходящий U по какому-либо параметру. Тогда код – это отображение $C: U \rightarrow Z$ и $z = C(u)$.

Код всегда можно задать т.н. кодовой таблицей для алфавита U . Этим он отличается от языка – кодирование просто формальная замена символов, как транслит, в отличие от перевода, где слова подбираются из контекста. Код называется **однозначно декодируемым** тогда и только тогда, когда $(z_{11}z_{21} \dots z_{n1} = z_{12}z_{22} \dots z_{n2}) \wedge (z_{i1} = z_{i2})$. Коды бывают разные:

- Код фиксированной длины (fixed-length code) — кодирование каждого символа производится с помощью строк одинаковой длины. Также он называется равномерным или блоковым кодом.
- Код переменной длины (variable-length code) — кодирование производится с помощью строк переменной длины. Также называется неравномерным кодом.

Префиксный код (prefix code) – особенно важный вид кода, в котором никакое кодовое слово не является началом другого. Аналогично можно определить постфиксный код – это код, в котором никакое кодовое слово не является концом другого слова. Все вышеперечисленные коды являются **однозначно декодируемыми** или **разделимыми** (англ. uniquely decodable) – для такого кода любое слово, составленное из кодо-

вых слов, можно декодировать только единственным способом (однозначно декодируемый код, так же называемый разделимым). Вот некоторые классические примеры таких кодов:

- ASCII – блочный код.
- Код Хаффмана — префиксный код.
- Азбука Морзе — не является ни блочным, ни префиксным, тем не менее, однозначно декодируемый за счет использования пауз.

Вот классический пример префиксного кода:

Пусть $U = \{a, b, c\}$; $Z = \{0,1\}$; $c(a) = 00$, $c(b) = 01$, $c(c) = 1$. Тогда строка, например, *accbacbbabc*, будет закодирована как *001101001010100011*. Видно, что этот код однозначно разбивается на кодирующие слова. Префиксный код имеет колоссальное преимущество – слова не нужно отделять друг от друга каким-либо маркером, но у него есть и два больших недостатка. Во-первых, префиксные коды являются кодами переменной длины, а данные, в основном, считываются блочно, в результате код приходится считывать побитово, что значительно замедляет скорость считывания. Во-вторых, появление ошибок в кодовой комбинации приводит к неправильному декодированию не только данной, но и всех последующих комбинаций, в отличие от равномерных кодов, где ошибка в кодовой комбинации приводит к неправильному декодированию только ее.

Как уже говорилось, префиксный код всегда однозначно декодируем, обратное же, в общем случае, неверно. Например, пусть $U = \{a, b, c\}$; $Z = \{1,2,3\}$; $c(a) = 1$, $c(b) = 12$, $c(c) = 31$. Закодировав строку *abbca*, мы получим *11212311*. Эта строка однозначно декодируема, так как мы знаем, что слева от двойки и справа от тройки всегда стоит 1, но код префиксным не является.

Необходимое и достаточное существование префиксного кода с указанными свойствами дает неравенство Крафта-Макмиллана (т. Макмиллана, 1965 г.):

Пусть заданы кодируемый и кодирующий алфавиты, состоящие из n и d символов, соответственно, и заданы желаемые длины кодовых слов $L_1, L_2, \dots, L_n$. Тогда необходимым и достаточным условием существования префиксного кода, обладающего словами данной длины, является выполнение неравенства:

$$\sum_{i=1}^n d^{-L_i} \leq 1.$$

Префиксный код называется примитивным, если его невозможно сократить, т.е. при вычеркивании любого знака он перестает быть префиксным. Первая теорема Шеннона (об оптимальном кодировании) гласит:

Всегда существует префиксный код, для которого средняя длина сообщений отличается от нормированной энтропии не более чем на 1:

$$E < \frac{H}{\log_2 D} + 1.$$

Здесь E - средняя длина сообщения, H – энтропия источника информации, D - мощность алфавита.

Первой важной задачей теории информации является разработка методов наилучшего энтропийного кодирования – кодирования последовательности значений с возможностью однозначного восстановления с целью уменьшения объема данных с помощью усреднения вероятностей появления элементов в закодированной последовательности. Предполагается, что до кодирования отдельные элементы последовательности имеют различную вероятность появления. После кодирования в результирующей последовательности вероятности появления отдельных символов практически одинаковы (энтропия на символ максимальна).

Различают несколько вариантов кодов:

- Сопоставление каждому элементу исходной последовательности различного числа элементов результирующей последовательности. Чем больше вероятность появления исходного элемента, тем короче соответствующая результирующая последовательность. Примером могут служить код Шеннона-Фано, код Хаффмана;
- Сопоставление нескольким элементам исходной последовательности фиксированного числа элементов конечной последовательности. Примером является код Танстола;
- Другие структурные коды, основанные на операциях с последовательностью символов. Примером является кодирование длин серий;
- Если приблизительные характеристики энтропии потока данных предварительно известны, может быть полезен более простой статический код, такой как унарное кодирование, гамма-код Элиаса, код Фибоначчи, код Голломба или кодирование Райса.

Согласно теореме Шеннона, существует предел сжатия без потерь, зависящий от энтропии источника. Чем более предсказуемы получаемые данные, тем лучше их можно сжать. Случайная независимая равновероятная последовательность сжатию без потерь не поддается.

Рассмотрим классическое кодирование – метод Шеннона-Фано (SFC). Этот алгоритм использует избыточность сообщения, заключенную в неоднородном распределении частот символов первичного алфавита,

то есть заменяет коды более частых символов короткими двоичными последовательностями, а коды более редких символов — более длинными двоичными последовательностями.

Алгоритм был независимо друг от друга разработан Шенноном (публикация «Математическая теория связи», 1948 год) и позже Фано (опубликовано как технический отчет).

Код Шеннона-Фано строится с помощью дерева. Построение этого дерева начинается от корня. Всё множество кодируемых элементов соответствует корню дерева (вершине первого уровня). Оно разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Эти подмножества соответствуют двум вершинам второго уровня, которые соединяются с корнем. Далее каждое из этих подмножеств разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Им соответствуют вершины третьего уровня. Если подмножество содержит единственный элемент, то ему соответствует конечная вершина кодового дерева; такое подмножество разбиению не подлежит. Подобным образом поступаем до тех пор, пока не получим все концевые вершины. Ветви кодового дерева размечаем символами 1 и 0.

При построении кода Шеннона-Фано разбиение множества элементов может быть произведено, вообще говоря, несколькими способами. Выбор разбиения на уровне n может ухудшить варианты разбиения на следующем уровне ($n + 1$) и привести к неоптимальности кода в целом. Другими словами, оптимальное поведение на каждом шаге пути ещё не гарантирует оптимальности всей совокупности действий. Поэтому код Шеннона-Фано не является оптимальным в общем смысле, хотя и дает оптимальные результаты при многих распределениях вероятностей. Для одного и того же распределения вероятностей можно построить, вообще говоря, несколько кодов Шеннона-Фано, и все они могут дать различные результаты.

Допустим, нам необходимо закодировать текст, для которого известно следующее распределение:

A (частота встречаемости 50)

B (частота встречаемости 39)

C (частота встречаемости 18)

D (частота встречаемости 49)

E (частота встречаемости 35)

F (частота встречаемости 24)

Построим следующее дерево (в скобках — вес вершины, т.е. суммарная частота вхождения букв в слово):

Полученный код: A — 11, B — 101, C — 100, D — 00, E — 011, F — 010. Любому префиксному коду, таким образом, можно однозначно сопоставить кодовое дерево. Если в кодовом дереве нет неиспользуемых ветвей, то его называют завершенным.

Кодирование Шеннона-Фано является частным случаем т.н. арифметического кодирования, однако все эти методы имеют изначально присущий им значительный недостаток. Для эффективной работы они нуждаются в априорном знании о распределении букв в сообщении. Естественно, как правило, эта информация неизвестна. В таком случае на помощь приходит вторая группа кодов, разработанная в 1970-е годы — кодирование Лемпеля-Зива (LZ). Его идея заключается в замене подстроки указателем на ее раннее вхождение в кодируемую последовательность. Например, пусть $U = \{a, b\}$.

Алгоритм LZ можно представить так:

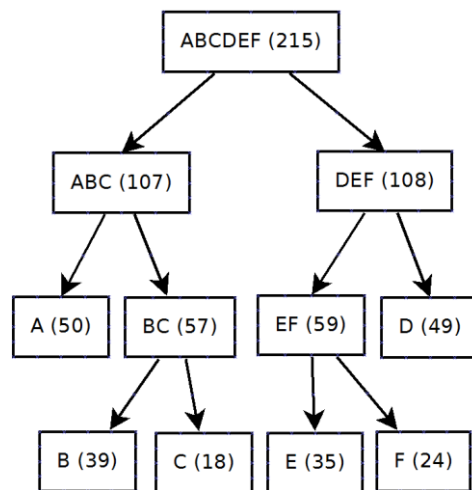
1. Инициализировать словарь всеми односимвольными строками;
2. Найти самый длинный блок w , встречающийся в словаре;
3. Кодировать его индексом в словаре;
4. Добавить к w следующий символ и добавить получившуюся строку в словарь;
5. Перейти к п.2.

Вот пример кодирования строки:

a b b a a b b a a b a b b a a a b a a b b a
0 1 1 0 2 4 2 6 5 5 7 1 4

В результате получаем сообщение 0110242655714, к которому необходимо добавить словарь для расшифровки. Коды типа LZ и более совершенные LZW применяются повсеместно — от сжатия данных архиватором Zip, до кодирования графической информации в профессиональном формате tiff.

Однако каким бы не был код, добиться совершенного сжатия, т.е. такого, при котором размер любого файла в процессе сжатия уменьшается хотя бы на 1 бит, в принципе невозможно. Это элементарно доказывается тем фактом, что для произвольного алфавита мощностью D , наборов строк длины k ровно D^k . Если же мы захотим перекодировать все строки и сократить их длину хотя бы до $k - 1$, их количество очевидным образом станет равно D^{k-1} .



Следовательно, согласно принципу Дирихле, минимум две строки при таком кодировании перейдут в одну, т.е. их однозначное восстановление станет невозможным, что полностью обесценивает такой гипотетический код. Тем не менее, с упорством, достойным лучшего применения, регулярно проскакивают сообщения о написании такого архиватора, вот только с распаковкой у авторов возникают определенные проблемы.

Теперь рассмотрим ситуацию, чрезвычайно важную для любой передачи информации – как гарантировать ее целостность? Тривиальным ответом будет передать код несколько раз, на случай ошибок, но нет ли более экономичного способа? Простейший вариант кодирования, при котором можно сразу же обнаружить единичную ошибку, задействовав при этом всего один дополнительный бит, работает так. Рассмотрим произвольное сообщение, например 01010110101110, и подвергнем его сложению по модулю два, также известному как XOR. Эта операция возвращает 0, если количество единиц четно, и 1 в противном случае. В результате, любая одиночная ошибка изменит данный бит и приведет к его несовпадению с изначальным, что позволит детектировать искажение сообщения. Подобные схемы использует серверная ОЗУ с контролем четности.

Однако один бит бесполезен против двойной ошибки, не меняющей четность, к тому же он не позволяет ее автоматически исправить. Для решения этой проблемы известный математик Ричард Хемминг предложил свою систему – т.н. код Хемминга.

В середине 1940-х годов он работал в Bell Labs на счётной машине Bell Mk. V. Это была электромеханическая машина, использующая релейные блоки, скорость которых была очень низка: один оборот за несколько секунд. Данные вводились в машину с помощью перфокарт, и поэтому в процессе чтения часто происходили ошибки. В рабочие дни использовались специальные коды, чтобы обнаруживать и исправлять найденные ошибки, при этом оператор узнавал об ошибке по свечению лампочек, исправлял и снова запускал машину. В выходные дни, когда не было операторов, при возникновении ошибки машина автоматически выходила из программы и запускала другую.

Хэмминг часто работал в выходные дни, и все больше и больше раздражался, потому что часто должен был перезагружать свою программу из-за ненадежности перфокарт. На протяжении нескольких лет он проводил много времени над построением эффективных алгоритмов исправления ошибок и в 1950 г. опубликовал способ, который сегодня известен как код Хэмминга.

Проще всего пояснить работу этого кода так. Рассмотрим исходную строку длиной 2^k бит. Для получения ее кода Хемминга добавим к ней k пар бит по следующему принципу:

- Первая пара – сумма четных и нечетных бит
- Вторая пара – сумма бит, в чьем номере второй бит с конца ноль, и сумма тех бит, в чьем номере второй бит с конца единица.
-
- $k - я$ пара – сумма бит, в чьем номере $k - й$ с конца бит равен нулю, и сумма тех бит, в чьем номере $k - й$ с конца бит равен единице.

В результате, увеличив код длиной n на $\log_2 n + 1$, можно обнаружить и исправить любую одиночную ошибку.

Естественно, коды Хемминга – не самое совершенное, что сейчас имеется в теории информации, существует огромный класс циклических (CRC) кодов, коды Рида-Соломона и т.п. Их важность сложно переоценить – ни один канал связи не бывает идеальным, ни одна ячейка памяти не может работать вечно без сбоев, в результате, без кодов коррекции ошибок работа ЭВМ была бы невероятно затруднена и практически невозможна.

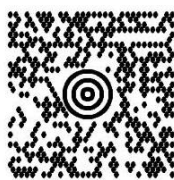
К теории кодирования относят и изучение графических кодов. Вот, например, несколько двумерных альтернатив стандартному штрих-коду:



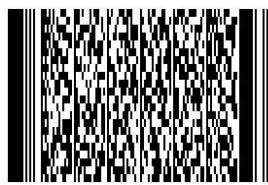
Aztec Code



Data Matrix



MaxiCode



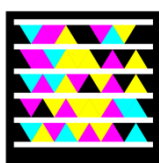
Pdf417



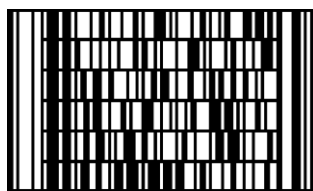
QR-код



Shotcode



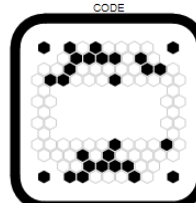
Microsoft Tag
(HCCB)



Codablock-F



EZcode



BeeTagg



DataBar GS1

Двухмерные символика были разработаны для кодирования большого объема информации. Расшифровка такого кода проводится в двух измерениях (по горизонтали и по вертикали). Двухмерные коды подразделяются на многоуровневые (stacked) и матричные (matrix). Многоуровневые штрих-коды появились исторически раньше и представляют собой поставленные друг на друга несколько обычных линейных кодов. Матричные же коды более плотно упаковывают информационные элементы по вертикали. В настоящее время разработано множество двухмерных штрих-кодов, применяемых с той или иной широтой распространения.

Еще одним важным аспектом теории информации является изучение каналов связи – возможностей дискретизации аналогового сигнала и его восстановления, изучения пропускной способности различных каналов и влияния помех. Одним из ключевых результатов здесь является теорема Найквиста-Шеннона (традиционно в русскоязычной литературе – Котельникова, о Шенноне и Найквисте как-то забывают):

Если аналоговый сигнал $x(t)$ имеет ограниченный по ширине спектр, то он может быть восстановлен однозначно и без потерь по своим отсчетам, взятым с частотой, строго большей удвоенной верхней частоты, $f > 2f_c$.

Такая трактовка рассматривает идеальный случай, когда сигнал начался бесконечно давно и никогда не закончится, а также не имеет во временной характеристике точек разрыва. Именно это подразумевает понятие «спектр, ограниченный частотой f_c ».

Разумеется, реальные сигналы (например, звук на цифровом носителе) не обладают такими свойствами, так как они конечны по времени и обычно имеют разрывы во временной характеристике. Соответственно, их спектр бесконечен. В таком случае полное восстановление сигнала невозможно, и из теоремы вытекают два следствия:

- Любой аналоговый сигнал может быть восстановлен с какой угодно точностью по своим дискретным отсчетам, взятым с удвоенной максимальной частотой, которой ограничен спектр реального сигнала.
- Если максимальная частота в сигнале превышает половину частоты дискретизации, то способа восстановить сигнал из дискретного в аналоговый без искажений не существует.

1.2.3. Обратимые вычисления

На стыке теории информации и физики расположен необычный эффект, который носит имя Ландауэра – теоретически сформулированный инженером IBM Рольфом Ландауэром (Rolf Landauer) в 1961 г. и обнаруженный совсем недавно, в 2012 г.

Этот эффект заключается в том, что в любой вычислительной системе, независимо от ее архитектуры и физической реализации, при любой потере 1 бита информации выделяется количество теплоты W , равное

$$W = k_b \cdot T \cdot \ln 2,$$

где k_b – константа Больцмана, а T – абсолютная температура системы. Также из этой формулы можно оценить минимальную энергию, потребляемую системой для обработки 1 бита информации при заданной температуре – т.н. энергию Шеннона-фон Неймана-Ландауэра. Эта величина чрезвычайно мала и при $T = 300\text{ K}$ составляет порядка 10^{-21} Дж/бит.

Но какое отношение термодинамика имеет к компьютерам? На самом деле эффект Ландауэра по своей силе сравним с квантовыми компьютерами – и то и то теоретически возможно и сулит огромные выгоды, но практически реализуемо с огромным трудом. Однако, если преимущества квантовых компьютеров более-менее понятны, то как нам поможет оценка энтропии на стираемый бит? На самом деле ответ лежит на поверхности.

Основной проблемой на пути создания все более мощных систем является колоссальное энергопотребление и тепловыделение. Уменьшение техпроцесса нельзя вести до бесконечности – на уровне менее 5 нм начинают активно проявляться квантовые эффекты, совсем нам не нужные. Увеличение же частоты без пропорционального уменьшения техпроцесса приводит к проблемам отведения тепла. Конечно, скорость можно наращивать, распараллеливая вычисления, но здесь мы наталкиваемся на стену энергопотребления.

Исследуя этот вопрос еще в 1961, Рольф Ландауэр показал, что даже если устранить все посторонние причины выделения энергии в счетном устройстве – трение, электрическое сопротивление и пр., тепло все равно будет выделяться. Этот совершенно парадоксальный факт связан с тем, что при стирании любого бита информации происходит выделение тепла в силу увеличения энтропии системы.

Рассуждения Ландауэра сводились к следующему. Бит стирается – значит, теряется информация о предыдущем состоянии системы. Раз теряется информация – значит, возрастает энтропия, а с возрастанием энтропии должна возрастать температура системы согласно законам термодинамики. При этом к стиранию относится не только собственно удаление информации, но и любая перезапись – так как при перезаписи предыдущее содержимое памяти исчезает.

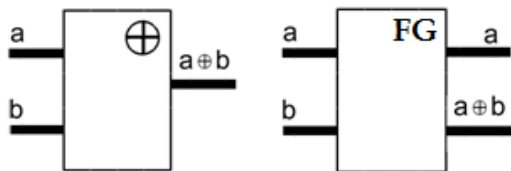
Оказывается, и это действительно парадоксально, что изрядная причина тепловыделения – не нагрев из-за прохождения тока через цепи компьютера, а ежесекундная перезапись миллионов бит информации. Поэтому системы, чувствительные к нагреву, типа квантовых компьютеров, построить вообще невозможно.

Изящное решение проблемы было сформулировано коллегой Ландауэра Чарльзом Беннетом (Charles Bennett) в 1973 г. Смысл решения прост – раз стирание битов приводит к выделению тепла, значит, при рабо-

те компьютера биты удаляться не должны, иначе говоря, требуется, чтобы по результату вычисления можно было однозначно восстановить исходные данные.

Например, операция $a := a \wedge b$ обратимой, очевидно, не является, в отличие от отрицания – зная его результат, мы легко восстановим изначальное значение. Аналогично блок, выполняющий операцию $\oplus$, необратим, но если он дополнительно возвращает на выходе одно из слагаемых – тогда операция будет обратима.

Итак, чтобы обратимо вычислить сумму двух булевых величин, достаточно просто не забывать хотя бы одно из слагаемых. Но если требуется перемножить два булевых значения, то оказывается, что никакой обратимый блок с двумя входами и двумя выходами этого сделать не может. Для вычисления булевого произведения был разработан обратимый вентиль Тоффоли (Tommaso Toffoli, CCNOT, 1980 г.) с тремя входами a, b, c и тремя выходами: $a, b, c \oplus (a \wedge b)$. Не менее известен вентиль Фредкина (Edward Fredkin, CSWAP) – если первый его вход установлен, он переставляет значения битов со входов 2 и 3, иначе все три бита остаются без изменений.



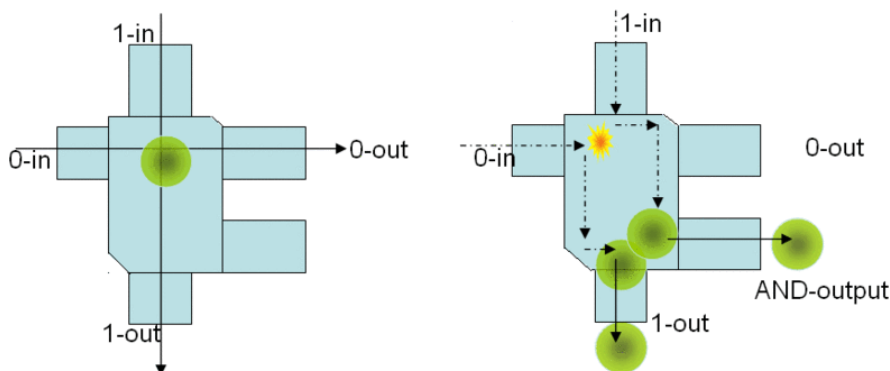
Слева обычный вентиль $\oplus$, справа – контролируемое отрицание (CNOT, Feynman Gate)

Если все мусорные биты будут сохранены к концу вычисления, то вычисление может быть проведено без выделения энергии, но на самом деле эти биты не такие уж мусорные. Чтобы использовать компьютер ещё раз, нужно вернуть его к исходному состоянию. Если просто удалить из компьютера результаты вычислений, то произойдёт то самое выделение тепла, которого хотелось бы избежать. Однако, если используя мусорные биты, запустить вычисления в обратном направлении, то компьютер так же без выделения тепла вернётся в исходное состояние.

Недостаток состоит в том, что для вычисления элементарных функций приходится вводить не вполне понятные новые переменные, которым приписываются значения констант, а многие выходы блоков нужны только для обратимости (как, например, первые два выхода блока Тоффоли) и даже называются мусорными.

Хорошая новость — блоков NOT, CNOT и CCNOT достаточно для вычисления любой функции, более того, обратимый процессор даже можно построить только на вентиле Тоффоли. Однако при попытке синтеза обратимых схем оказывается, что обратимые блоки можно выстраивать только каскадом, а количество константных переменных необходимо минимизировать, так как очень часто цена за использование дополнительной переменной очень высока. Это сильно осложняет задачу такого синтеза, и хороших методов ее решения пока не существует.

Идея обратимых вычислений идеально сочетается с концепцией квантовых компьютеров, о чем мы еще поговорим, а пока упомянем лишь несколько моделей, наглядно демонстрирующих идеи обратимости. Первая такая модель была предложена Эдвардом Фредкиным и носит название бильярдной (Billiard Ball Model, BBM).



Элементы CSWAP и CCNOT в BBM

Норман Марголус (Norman Margolus), обобщая идею бильярдных вычислений, пришел к идее обратимых клеточных автоматов, не делая, впрочем, никаких утверждений о возможности их физической реализации. Обратимый клеточный автомат работает по следующему принципу. Плоскость разбивается на квадраты размером 2×2 непрерывными и пунктирными линиями. Создается правило переписывания, по которому клеточки на каждом шаге эволюции автомата во времени меняют свой цвет. Затем в каждый нечетный момент времени правило используется для перекраски клеточек разбиения, заданного непрерывными линиями, а в каждый четный момент времени то же самое правило применяется к клеткам разбиения, заданного пунктиром.

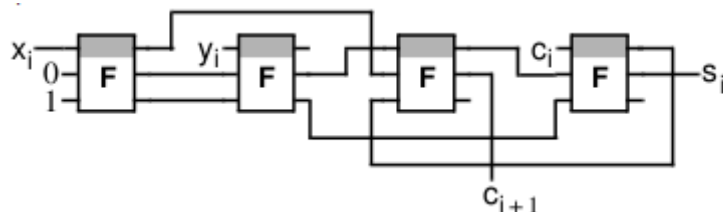
ВВМ является, пожалуй, самой интуитивно понятной обратимой моделью вычислений. Ее обратимость очевидна, так как шарики можно запустить в обратную сторону, от выходов к входам; обратимость правила переписывания клеточного автомата, моделирующего ВВМ, тоже несложно проверить непосредственно. ВВМ допускает и интуитивно понятную технологическую реализацию: использовать шарики и препятствия, пытаясь сделать объекты как можно меньше по размерам, чтобы использовать большее их количество. Проблемы такой технологии тоже очевидны: невозможность абсолютно упругого отражения, наличие трения, необходимость сверхточной механики, появление квантовых эффектов на молекулярном уровне.

Очень важно и то, что синтез схем для реальных вычислений здесь весьма сложен (попытайтесь создать ВВМ-компьютер, который был бы способен складывать и умножать хотя бы двухзначные двоичные числа!), поэтому описанная модель вряд ли имеет практические перспективы.

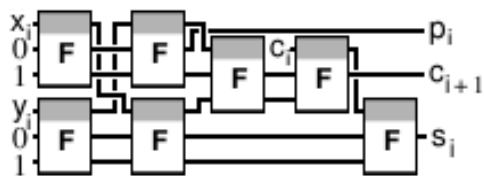
Естественно поставить вопрос: допускает ли природа физически обратимые технологии, которые могли бы быть использованы для обратимых вычислений? CMOS не является обратимой технологией ввиду необратимости единичного блока (транзистора), из которых строятся схемы. Требуются новые физические объекты или явления, которые могли бы быть использованы для представления информации. Сложными для понимания, но более простыми, чем модель Фредкина, с точки зрения технологической реализации, являются, например, кодирование информации магнитными спинами в технологии ЯМР или поляризацией фотонов в оптической технологии. Имеются и другие способы – ионных ловушек, нейтральных атомов и т.п. и именно их чаще всего объединяют под именем квантовых технологий.

Логика, изучающая процессы с сохранением энтропии, называется консервативной или адиабатической. В оригинальной статье Ландрауер показал, что закон сохранения энергии все равно обмануть не удастся, и попытка сделать компьютер полностью реверсивным приводит к эквивалентной нереверсивности при загрузке информации. В результате, на вычисления должна расходоваться некоторая минимальная энергия и, соответственно, выделяться минимальное тепло, но если не использовать консервативную логику, то эти величины будут на порядки больше.

Выигрыш от использования консервативных вентилях очевиден – и частично его можно получить даже не используя квантовый компьютер. Например, одноканальный сумматор требует четырех вентилях и срабатывает за $2 \cdot \tau$, где τ – время прохождения сигнала от входа одного вентиля до входа другого⁹. Сумматоры большей разрядности можно делать из одного одноканального сумматора и $(n - 1)$ трёхбитных суммирующих блоков. Минимальные по размеру трёхбитные суммирующие блоки состоят из четырех вентилях:



Однако такие блоки работают довольно медленно: сложение n -битных чисел занимает $(2n + 1) \cdot \tau$, при том, что используются $(4n - 2)$ блоков. Существуют оптимизированные по скорости (не минимальные по размеру) трёхбитные сумматоры. Лучший известный вариант выглядит следующим образом:



Эти блоки можно соединять как последовательно, реализуя сложение в столбик, так и, например, деревом, реализуя сложение Когге-Стоуна. Сложение в столбик занимает $(n + 2) \cdot \tau$ и требует $(7n - 3)$ вентилях. Сложение алгоритмом Когге-Стоуна занимает $[3 + \log_2 n] \cdot \tau$ и использует дополнительные вентилях в количестве $[n \cdot \log_2 n]$.

В случае 64-битных чисел, сложение в столбик потребует 66τ и 445 вентилях, а сложение Когге-Стоуна потребует 9τ и 957 вентилях. Компромиссные варианты, работающие за $10 - 12\tau$, могут обходиться ≈ 500 вентилями. В предположении, что удастся реализовать баллистические (т.е. не привносящие задержки в передачу сигнала) переключатели Фредкина размеров порядка применяемых в современных процессорах транзисторов, сложение двух 64-битных чисел можно будет производить (в совершенно реалистичных предположениях) с частотой, по меньшей мере, миллион ГГц.

⁹По материалам Bruce, Thornton et al., «Efficient Adder Circuits Based on a Conservative Reversible Logic», Mississippi State University, 2002.

1.3. Математическая логика

1.3.1. Булева алгебра

Одной из основных и ценнейших вещей в математике, с точки зрения создателей ЭВМ, является алгебра логики. Чем же она так важна? Если не углубляться в формализм, булевой алгеброй можно назвать алгебраическую систему, содержащую три основных операции: бинарные, $\wedge$ – конъюнкция и $\vee$ – дизъюнкция и унарную – отрицание, обычно обозначаемую чертой над операндом и двумя выделенными элементами: 0 и 1. При этом для любых a, b, c верны следующие аксиомы:

$\vee (b \vee c) = (a \vee b) \vee c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	ассоциативность
$a \vee b = b \vee a$	$a \wedge b = b \wedge a$	коммутативность
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	дистрибутивность
$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$	поглощение
$a \vee \bar{a} = 1$	$a \wedge \bar{a} = 0$	дополнительность

Самая простая нетривиальная булева алгебра содержит всего два элемента, 0 и 1, а действия в ней определяются следующей таблицей:

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

a	0	1
$\bar{a}$	1	0

Эта алгебра наиболее часто используется в логике, так как является точной моделью классического исчисления высказываний. В этом случае 0 называют ложью, 1 — истиной. Можно построить и менее тривиальные примеры, например, т.н. алгебра Линденбаума-Тарского (фактормножество всех утверждений по отношению равносильности в данном исчислении с соответствующими операциями).

Так же множество всех подмножеств данного множества S образует булеву алгебру относительно операций $\cup$ - объединение, $\cap$ - пересечение и унарной операции дополнения. Ноль здесь – пустое множество, а единица – само S .

Логическая функция, или функция алгебры логики от n переменных – отображение $B^n \rightarrow B$, где $B = \{0,1\}$ – булево множество. Число $n \in \mathbb{N}$ называют местностью функции, в случае $n = 0$ булева функция превращается в булеву константу. Элементы декартова произведения B^n называют булевыми векторами. Множество всех булевых функций от любого числа переменных часто обозначается P_2 , а от n переменных – $P_2(n)$.

Каждая булева функция местности n полностью определяется заданием своих значений на своей области определения, то есть на всех булевых векторах длины n . Число таких векторов равно 2^n . Поскольку на каждом векторе булева функция может принимать значение либо 0, либо 1, то количество всех n -местных булевых функций равно 2^{2^n} . То, что каждая булева функция задаётся конечным массивом данных, позволяет представлять их в виде таблиц. Такие таблицы носят название таблиц истинности и в общем случае имеют вид:

x_1	x_2	...	x_n	$f(x_1, x_2, \dots, x_n)$
0	0	...	0	$f(0, 0, \dots, 0)$
0	0	...	1	$f(0, 0, \dots, 1)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
1	1	1	1	$f(1, 1, \dots, 1)$

Нульместные функции и унарные функции.

При $n = 0$ количество булевых функций сводится к двум, первая из них тождественно равна 0, а вторая 1. При $n = 1$ число булевых функций равно 4. Это уже известные нам 0 и 1, а также тождественная функция (EQV) и отрицание (NOT).

Бинарные и тернарные функции

При $n = 2$ число булевых функций равно 16, и здесь начинается самое интересное. К уже известным нам 4-м, а также конъюнкции (AND) и дизъюнкции (OR) обычно добавляют:

$x \rightarrow y$ или GE(x, y) – импликация;

$\overline{x \rightarrow y}$, $x > y$ или LT(x, y) – больше, инверсия прямой импликации;

$\overline{x \leftarrow y}$, $x < y$ или GT(x, y) – меньше, инверсия обратной импликации;

$\overline{x \vee y}$, $x \downarrow y$ или $\text{NOR}(x, y)$ – антидизъюнкция, стрелка Пирса, функция Даггера;

$\overline{x \wedge y}$, $x | y$ или $\text{NAND}(x, y)$ – антиконъюнкция, штрих Шеффера;

$x \oplus y$, $x \neq y$, $x +_2 y$ или $\text{XOR}(x, y)$ – сложение по модулю 2, исключающее или, неравно;

При $n = 3$ число булевых функций равно 256. Конечно, нет никакого смысла перечислять все, упомянув лишь пару более-менее известных:

$x \downarrow y \downarrow z$ или $\text{Webb}(x, y)$ – функция Вебба;

$\geq 2(x, y, z)$ – переключатель по большинству, мажоритарный клапан.

Естественно, можно использовать и тройную конъюнкцию, равенство, дизъюнкцию и т.п.

Замкнутым классом функций называется такое множество функций P , замыкание которого относительно операции суперпозиции совпадает с ним самим: $[P] = P$. Другими словами, любая функция, которую можно выразить формулой с использованием функций из множества P , снова входит в это же множество. Множество P_2 всех возможных булевых функций замкнуто.

В 1941 году Эмиль Пост показал, что любой замкнутый класс булевых функций является пересечением конечного числа данных классов:

- Сохраняющие константу 0 функции:

$$T_0 = \{f(x_1, \dots, x_n) | f(0, \dots, 0) = 0\}$$

- Сохраняющие константу 1 функции:

$$T_1 = \{f(x_1, \dots, x_n) | f(1, \dots, 1) = 1\}$$

- Самодвойственные функции:

$$S = \{f(x_1, \dots, x_n) | f(\overline{x_1}, \dots, \overline{x_n}) = \overline{f(x_1, \dots, x_n)}\}$$

- Монотонные функции:

$$M = \{f(x_1, \dots, x_n) | \forall i (a_i \leq b_i) \Rightarrow f(a_1, \dots, a_n) \leq f(b_1, \dots, b_n)\}$$

- Линейные функции:

$$L = \{f(x_1, \dots, x_n) | f(x_1, \dots, x_n) = a_0 \oplus a_1 \wedge x_1 \oplus \dots \oplus a_n \wedge x_n, a_i \in \{0, 1\}\}$$

Если для каждого класса в системе найдётся функция, не входящая в него, то такая система будет полной, и с помощью входящих в неё функций можно будет получить любую другую булеву функцию. Э. Пост доказал, что множество замкнутых классов булевых функций — счётное множество, и установил, что любой замкнутый класс может быть порождён конечным базисом. Заметим, что существуют функции, не входящие ни в один из классов Э. Поста. Любая такая функция сама по себе образует полную систему. В качестве примеров можно назвать штрих Шеффера или стрелку Пирса.

Критерий Поста формулирует необходимое и достаточное условие полноты системы булевых функций:

Система булевых функций полна тогда и только тогда, когда она не содержится целиком ни в одном из классов T_0, T_1, S, M, L .

Широко известны такие полные системы булевых функций:

- $\{\vee, \wedge, \overline{}\}$ – конъюнкция, дизъюнкция, отрицание;
- $\{\vee, \oplus, 1\}$ – конъюнкция, сложение по модулю 2, константа 1.

Первая система используется, например, для представления функций в виде дизъюнктивных и конъюнктивных нормальных форм, вторая — для представления в виде полиномов Жегалкина.

Полная система функций называется базисом, если она перестаёт быть полной при исключении из неё любого элемента. Первая из упоминавшихся выше полных систем базисом не является, поскольку согласно законам де Моргана либо дизъюнкцию, либо конъюнкцию можно исключить из системы и восстановить с помощью остальных двух функций. Вторая система является базисом – все три её элемента необходимы для полноты. Максимально возможное число булевых функций в базисе – 4.

Теорема Поста открывает путь к представлению булевых функций синтаксическим способом, который в ряде случаев оказывается намного удобнее, чем таблицы истинности. Относительно выбранной системы функций полезно знать ответы на следующие вопросы:

- Как построить по данной функции представляющую её формулу?
- Как проверить, что две разные формулы эквивалентны, то есть задают одну и ту же функцию?
- В частности, существует ли способ приведения произвольной формулы к эквивалентной ей канонической форме такой, что две формулы эквивалентны тогда и только тогда, когда их канонические формы совпадают?
- Как по данной функции построить представляющую её формулу с теми или иными заданными свойствами (например, наименьшего размера), и возможно ли это?

Положительные ответы на эти и другие вопросы существенно увеличивают прикладное значение выбранной системы функций.

Стандартные базисы:

Дизъюнктивной нормальной формой или ДНФ называется дизъюнкция простых конъюнкций. Элементарная конъюнкция:

- правильная, если каждая переменная входит в неё не более одного раза (включая отрицание);
- полная, если каждая переменная (или её отрицание) входит в неё ровно 1 раз;
- монотонная, если она не содержит отрицаний переменных.

Совершенной дизъюнктивной нормальной формой или СДНФ относительно некоторого заданного конечного набора переменных называется такая ДНФ, у которой в каждую конъюнкцию входят все переменные данного набора, причём в одном и том же порядке.

Легко убедиться, что каждой булевой функции соответствует некоторая ДНФ, а функции, отличной от тождественного нуля — даже СДНФ. Для этого достаточно в таблице истинности этой функции найти все булевы векторы, на которых её значение равно 1, и для каждого такого вектора $b = (b_1, \dots, b_n)$ построить конъюнкцию $x_1^{b_1} \wedge x_2^{b_2} \wedge \dots \wedge x_n^{b_n}$, где $x_i^1 = x_i$, $x_i^0 = \bar{x}_i$.

Дизъюнкция этих конъюнкций является СДНФ исходной функции, поскольку на всех булевых векторах её значения совпадают со значениями исходной функции. Например, для импликации $x \rightarrow y$ результатом является $(\bar{x} \wedge y) \vee (x \wedge y) \vee (x \wedge \bar{y})$, что легко упростить до $\bar{x} \wedge y$.

Конъюнктивная нормальная форма (КНФ) определяется двойственно к ДНФ. КНФ — это конъюнкция простых дизъюнкций.

Совершенной конъюнктивной нормальной формой (СКНФ), относительно некоторого заданного конечного набора переменных, называется такая КНФ, у которой в каждую дизъюнкцию входят все переменные данного набора, причём в одном и том же порядке.

Алгебраическая нормальная форма (общепринятое название в зарубежной литературе) или полином Жегалкина (название, используемое в отечественной литературе) — это форма представления логической функции в виде полинома с коэффициентами вида 0 и 1, в котором в качестве произведения используется операция конъюнкции, а в качестве сложения — сложение по модулю 2.

Для получения полинома Жегалкина следует выполнить следующие действия:

- Получить СДНФ функции,
- Все $\vee$ заменить на $\oplus$,
- Во всех термах заменить элементы с отрицанием на конструкцию: («элемент» $\ll \oplus \gg 1$),
- Раскрыть скобки и привести попарно одинаковые термы.

Формально полином Жегалкина можно представить в виде:

$$P(x_1, \dots, x_n) = a \oplus a_1 \wedge x_1 \oplus \dots \oplus a_n \wedge x_n \oplus a_{12} \wedge x_1 \wedge x_2 \oplus \dots \oplus a_{1\dots n} \wedge x_1 \wedge \dots \wedge x_n$$

Классификация булевых функций

- По количеству входных операндов, от которых зависит значение на выходе функции.
- По количеству единиц и нулей в таблице истинности отличают узкий класс сбалансированных булевых функций¹⁰ от более широкого класса несбалансированных булевых функций¹¹. Сбалансированные булевы функции в основном используются в криптографии.

- По зависимости значения функции от перестановки её входных бит различают симметричные булевы функции (значение которых зависит только от количества единиц на входе) и несимметричные булевы функции (значение которых так же зависит от перестановки её входных бит).

- По значению функции на противоположных друг другу наборах значений аргументов отличают самодвойственные функции (значение которых инвертируется при инвертировании значения всех входов) от остальных булевых функций, не обладающих таким свойством. Нижняя часть таблицы истинности для самодвойственных функций является зеркальным отражением инвертированной верхней части (если расположить входные комбинации в таблице истинности в естественном порядке).

- По алгебраической степени нелинейности отличают линейные булевы функции (АНФ которых сводится к линейной сумме по модулю 2 входных значений) и нелинейные булевы функции (АНФ которых содержит хотя бы одну нелинейную операцию конъюнкции входных значений). Примерами линейных функций являются: дизъюнкция, сложение по модулю 2, а также все булевы функции, АНФ которых содержит лишь линейные операции сложения по модулю 2 без конъюнкций. Примерами нелинейных функций являются:

¹⁰ так же называемых уравновешенными или равновероятностными, поскольку при равновероятных случайных значениях на входе или при переборе всех комбинаций по таблице истинности вероятность получения на выходе значения 1 равна $1/2$.

¹¹ так же называемых неуравновешенными, поскольку вероятность получения на выходе значения 1 отлична от $1/2$.

ся: конъюнкция, штрих Шеффера, стрелка Пирса, а также все булевы функции, АНФ которых содержит хотя бы одну нелинейную операцию конъюнкции.

Таким образом, ценность булевых функций и приведения их к нормальной форме для разработки ЭВМ становится очевидной. Ведь, по сути:

- Любая программа, выполняемая процессором, есть преобразование одного числа в двоичной системе в другое.
- Любое такое преобразование может быть записано в виде одной сколь угодно сложной формулы.
- Любая такая формула однозначно представляется в том или ином базисе.
- Нам достаточно выбрать логические схемы¹² (физически инвертирующие сигнал, складывающие сигналы и т.п.), реализующие этот базис. Это очень экономно, нам понадобится не более 3-х разных вентилях, чтобы с гарантией собрать любую схему, физически осуществляющую требуемое преобразование двоичного кода, т.е. выполнить любую программу.
- Естественно, что базовые логические функции слишком просты для их эффективного применения, поэтому на их основе собирают фиксированный набор более сложных функций, образующий систему команд данного процессора.

1.3.2. Секвенциальная логика

Сейчас мы вели речь о т.н. логике комбинационных устройств, они занимаются преобразованием статического сигнала. На такой логике работают запоминающие устройства, она называется последовательной, или секвенциальной.

Секвенциальная логика отличается от комбинационной тем, что моделирует цифровые устройства с учётом предыстории их функционирования. В зависимости от регламента функционирования цифровые устройства подразделяются на синхронные и асинхронные. Соответственно их поведение подчиняется либо синхронной, либо асинхронной логике.

Асинхронная секвенциальная логика для выражения эффекта запоминания использует моменты смены состояний, которые задаются не в явном виде, а исходя из сопоставления логических величин по принципу «раньше-позже». Для асинхронной логики достаточно установить очерёдность смены состояний безотносительно каких-либо привязок к реальному или виртуальному времени. Основными операциями такой логики являются секвенция и веньюнкция.

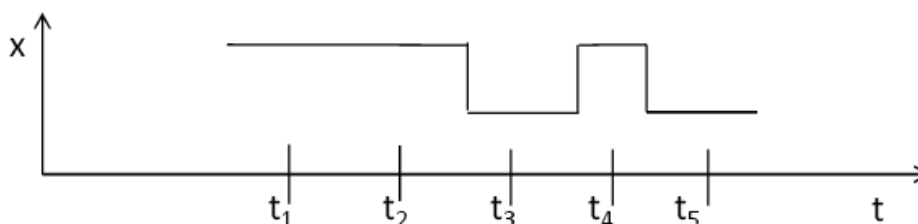
Секвенция, как явствует из названия, это последовательность элементов, представляемая упорядоченным двоичным множеством $\langle x \rangle = \langle x_1, \dots, x_n \rangle$, где $x_i \in \{0,1\}$. С помощью секвенции реализуется двоичная функция $z = \varphi(\langle x \rangle)$, такая, что $z = 1$ только в случае $(x_i = 1) \prec (x_j = 1) \forall i, j: i < j$. Символ $\prec$ означает, что выражение слева от него предшествует по времени выражению справа. Секвенция обращается в единицу при единичных значениях аргументов, считывание которых происходит поочередно. Во всех остальных случаях она равна 0.

Веньюнкция – это асимметрическая логико-динамическая операция $\angle$, определяемая так:

$x \angle y = 1$ только тогда, когда $x \wedge y$ и $\varphi(x, y) = 1$. Это означает, что веньюнкция истинна лишь тогда, когда $x = 1$ настало раньше, чем $y = 1$. В результате истинность ее означает переключение по времени $x = 0/1$ на фоне $y = 1$. Веньюнктор является основным операционным элементом памяти секвенциальной логики. Он реализуется на основании равенства $x \wedge (\bar{x} \vee x \angle y) = x \angle y$, где выражение $\bar{x} \vee x \angle y$ полностью описывает работу простейшей ячейки памяти – RS-триггера.

Секвентор можно очевидным образом построить на основании веньюнкторов. Например, для реализации секвентора $\langle x, y, z, u, v \rangle$ пригодны формулы $v \angle (u \angle (z \angle (y \angle x)))$ или $\langle x, y \rangle \wedge \langle y, z \rangle \wedge \langle z, u \rangle \wedge \langle u, v \rangle$.

Секвенции отличаются тем, что описать их обычными таблицами истинности весьма сложно – последовательность, например, $\langle 11010 \rangle$ означает изменение сигнала во времени. В результате для описания такой логики применяют временные диаграммы:



Здесь X – уровень сигнала, а t_i – моменты времени, называемые тактами.

¹² или вентили, как их называют

Если попытаться описать это таблицей, то получится:

t	x1	x2	x3	x4	x5
t0	0	0 ↓	1	1	0
t1	0 ↓	1 ↓	1 ↓	0	0 ↓
t2	1 ↓	1	0 ↓	0	1 ↓
t3	0 ↓	1	0 ↓	1	1 ↓
t4	1 ↓	1	1 ↓	1	0 ↓

Очевидно, что временная диаграмма намного удобнее.

1.3.3. Нечеткая логика и нейровычисления

Долгие годы считалось, что логика – эталон точности по определению. Логические высказывания строго формальны, однозначны и не допускают никаких разночтений. Однако, как и во многих подобных случаях, те, кто создал науку, сами обозначили ее границы – речь идет о древних греках и т.н. парадоксе корабля Тесея. Суть его заключается в том, что после похода аргонавтов прославленный корабль «Арго», принадлежащий герою Тесею, решено было сохранить для потомков. Однако, несмотря на тщательный уход, с течением времени, то одна доска, то другая начинали гнить и нуждались в замене, так же, как и такелаж. В результате возник вопрос – собственно, с какой по счету «неоригинальной запчасти» корабль уже перестанет быть тем самым, на котором плыл Тесей? Аналогичный вопрос ставился в апориях «Гора» и «Лысый». Один камень еще не образует гору, один выпавший волос еще не делает человека лысым, прибавление одного камня к куче также не делает ее горой, выпадение всего лишь еще одного волоса также не делает человека лысым. Так сколько же точно камней надо положить для гарантированного насыпания горы, и с какого по счету волоса человек лысеет?

Очевидно, что классическая логика здесь бессильна – рассуждением по индукции легко доказать, что раз один камень – не гора, и прибавление одного камня к другим не делает кучу горой, то создание горы из камней вообще невозможно. Тем не менее, это явно противоречит реальности. Эта проблема находилась в подвешенном состоянии много лет – очевидно, что и для горы и для лысины и множества других вещей невозможно создать единый алгоритм классификации – слишком уж субъективны человеческие оценки. Но неужели нет никакой математической базы, которая все-таки позволила бы работать с такими «расплывчатыми» объектами? Есть, и она называется нечеткая логика.

Термин нечёткое (общепринятый вариант, иногда говорят размытое, расплывчатое, туманное, путанное и даже пушистое – все это возможные значения английского слова fuzzy) множество — понятие, введенное Лотфи А. Заде¹³ в 1965 г. в одноименной статье «Fuzzy Sets» в журнале Information and Control. Специалист в теории вероятностей и теории множеств, он впервые догадался расширить классическое канторовское понятие множества, допустив, что характеристическая функция¹⁴ может быть не бинарной, а принимать любые значения в интервале [0,1].

Итак, нечетким множеством $\tilde{A}$ на универсальном множестве U называется совокупность пар вида $(\mu_A(u), U)$, где $\mu_A(u)$ - степень принадлежности элемента $u \in U$ к $\tilde{A}$. Степень принадлежности – число из диапазона [0,1]. Чем выше степень принадлежности, тем больше элемент соответствует свойствам $\tilde{A}$.

Функцией принадлежности (membership function) называется функция, которая позволяет вычислить степень принадлежности произвольного элемента универсального множества к нечеткому множеству. Если универсальное множество состоит из конечного количества элементов $\{u_1, \dots, u_k\}$, тогда

$$\tilde{A} = \sum_{i=1}^k \frac{\mu_A(u_i)}{u_i}$$

Если U непрерывно, то

$$\tilde{A} = \int_U \frac{\mu_A(u)}{u}$$

¹³ Лютфали Алескер-заде, знаменитый ирано-американский математик

¹⁴ функция принадлежности элемента множеству, также называется индикаторной функцией. Обычно, $I_A(a) = \begin{cases} 0, & a \notin A \\ 1, & a \in A \end{cases}$. Аналитически часто выражается через функцию Хевисайда $\theta(x)$, так же известную, как ступенька, иногда обозначается $H(x)$ или $1(x)$.
 $1(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$. используется в теории информации, автоматического управления и т.п.

Лингвистической переменной (linguistic variable) называется переменная, значениями которой могут быть слова или словосочетания некоторого естественного или искусственного языка. Терм-множеством (term set) называется множество всех возможных значений лингвистической переменной. Термом (term) называется любой элемент терм-множества. В теории нечетких множеств терм формализуется нечетким множеством с помощью функции принадлежности.

Дефаззификацией (defuzzification) называется процедура преобразования нечеткого множества в четкое число. В теории нечетких множеств процедура дефаззификации аналогична нахождению характеристик положения (математического ожидания, моды, медианы) случайных величин в теории вероятности. Простейшим способом выполнения процедуры дефаззификации является выбор четкого числа, соответствующего максимуму функции принадлежности. Однако пригодность этого способа ограничивается лишь одноэкстремальными функциями принадлежности.

Дефаззификация нечеткого множества

$$\tilde{A} = \int_{[\underline{U}, \bar{U}]} \frac{\mu_A(u)}{u}$$

по методу центра тяжести осуществляется по формуле

$$a = \frac{\int_{\underline{U}}^{\bar{U}} u \cdot \mu_A(u) du}{\int_{\underline{U}}^{\bar{U}} \mu_A(u) du}$$

Физическим аналогом этой формулы является нахождение центра тяжести плоской фигуры, ограниченной осями координат и графиком функции принадлежности нечеткого множества. В случае дискретного универсального множества дефаззификация нечеткого множества по методу центра тяжести осуществляется по формуле

$$a = \frac{\sum_{i=1}^k u_i \cdot \mu_A(u_i)}{\sum_{i=1}^k \mu_A(u_i)}$$

Нечеткой базой знаний (fuzzy knowledge base) о влиянии факторов $x = \{x_1, \dots, x_n\}$ на значение параметра y называется совокупность логических высказываний типа

Если

$$[(x_1 = a_1^{j_1}) \wedge \dots \wedge (x_n = a_n^{j_n})] \vee [(x_1 = a_1^{j_2}) \wedge \dots \wedge (x_n = a_n^{j_2})] \vee \dots \vee [(x_1 = a_1^{j_k}) \wedge \dots \wedge (x_n = a_n^{j_k})],$$

то $y = d_j \quad \forall j = \overline{1, m}$,

где $a_i^{j_p}$ - нечеткий терм, которым оценивается переменная x_i в строке номер j_p ,

$p = \overline{1, k}$,

k_j - количество строчек-конъюнкций, в которых выход y оценивается нечетким термом d_j ,

m - количество термов, используемых для лингвистической оценки выходного параметра y .

С помощью операций $\cup$ и $\cap$ нечеткую базу знаний можно переписать в более компактном виде:

$$\bigcup_{p=1}^{k_j} \left(\bigcap_{i=1}^n (x_i = a_i^{j_p}) \right) \mapsto y = d_j$$

Нечетким логическим выводом (fuzzy logic inference) называется аппроксимация зависимости $y = f(x_1, \dots, x_n)$ с помощью нечеткой базы знаний и операций над нечеткими множествами. Например, пусть

$\mu^{j_p}(x_i)$ - функция принадлежности входа x_i нечеткому терму $a_i^{j_p}$, $i = \overline{1, n}$, $j = \overline{1, m}$, $p = \overline{1, k_j}$, т.е.

$$a_i^{j_p} = \int_{\underline{x_i}}^{\bar{x_i}} \frac{\mu^{j_p}(x_i)}{x_i},$$

и $\mu^{d_p}(y)$ - функция принадлежности выхода y нечеткому терму d_j , т.е.

$$d_j = \int_{\underline{y}}^{\bar{y}} \frac{\mu^{d_j}(y)}{y}$$

Тогда степень принадлежности конкретного входного вектора $x^* = (x_1^*, \dots, x_n^*)$ нечетким термам d_j из нашей базы данных определяется следующей системой нечетких логических уравнений:

$$\mu^{d_j}(x^*) = \bigvee_{p=1, k_j} \bigwedge_{i=1, n} [\mu^{j_p}(x_i^*)],$$

где $\vee$ ($\wedge$) - операция максимума (минимума).

Нечеткое множество $\tilde{y}$, соответствующее входному вектору x^* , определяется следующим образом:

$$\tilde{y} = \bigcup_{j=1, m} \int_{\underline{y}}^{\bar{y}} \frac{\min(\mu^{d_j}(x^*), \mu^{d_j}(y))}{y},$$

где $\bigcup$ - операция объединения нечетких множеств. Четкое значение выхода y , соответствующее входному вектору x^* , определяется в результате дефаззификации нечеткого $\tilde{y}$.

Высотой нечеткого множества называется верхняя граница его функции принадлежности:

$$\text{hgt}(\tilde{A}) = \sup_{u \in U} (\mu_A(u))$$

Для дискретного универсального множества U супремум становится максимумом, значит, высотой нечеткого множества будет максимум степеней принадлежности его элементов. Нечеткое множество $\tilde{A}$ называется нормальным, если его высота равна единице. Нечеткие множества, не являющиеся нормальными, называются субнормальными. Нормализация – преобразование субнормального нечеткого множества в нормальное.

Носителем нечеткого множества называется четкое подмножество универсального множества, элементы которого имеют ненулевые степени принадлежности:

$$\text{supp}(\tilde{A}) = \{u: \mu_A(u) > 0\}$$

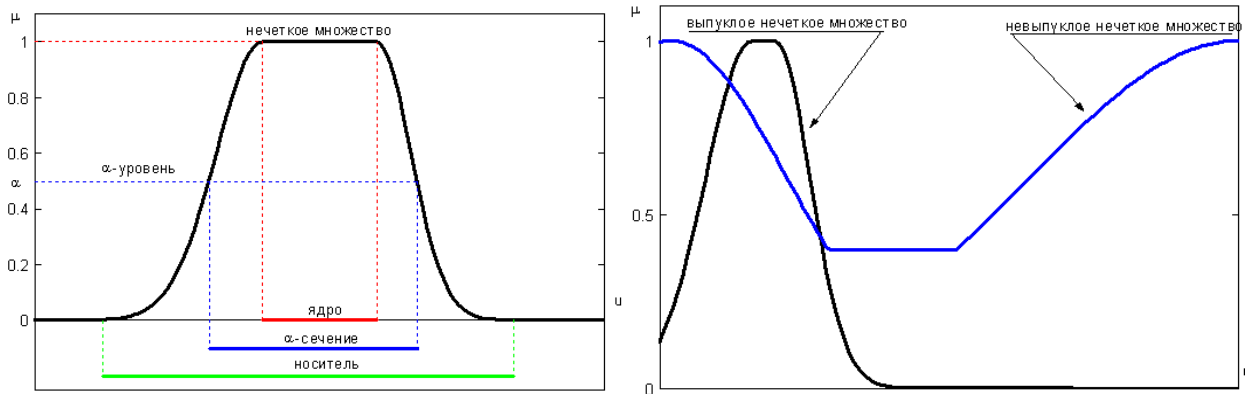
Нечеткое множество называется **пустым**, если его носитель является пустым множеством. **Ядром** нечеткого множества называется четкое подмножество универсального множества, элементы которого имеют степени принадлежности, равные единице:

$$\text{core}(\tilde{A}) = \{u: \mu_A(u) = 1\}$$

Ядро субнормального нечеткого множества всегда пустое. Альфа-сечением (или множеством α – уровня) нечеткого множества называется четкое подмножество универсального множества, элементы которого имеют степени принадлежности большие или равные α :

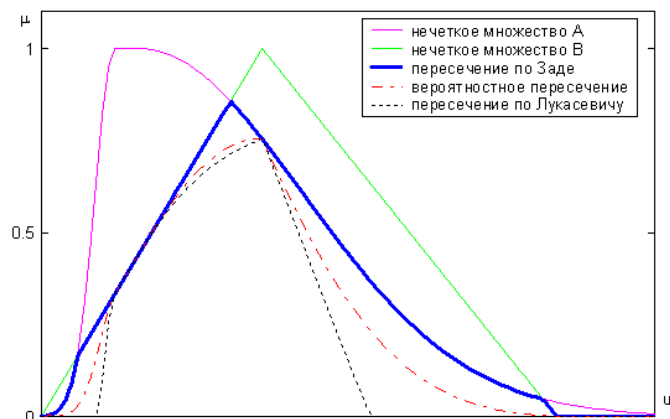
$$\tilde{A}_\alpha = \{u: \mu_A(u) \geq \alpha\}$$

Носитель (ядро) можно рассматривать как сечение нечеткого множества на нулевом (единичном) α -уровне.

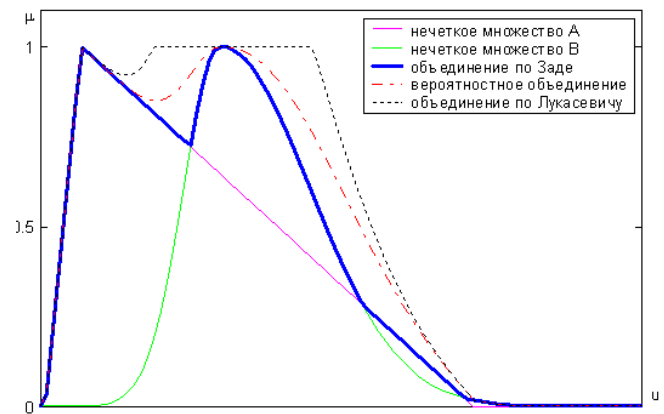


Нечеткое множество называется выпуклым если $\mu_A(\lambda \cdot u_1 + (1 - \lambda) \cdot u_2) \geq \min(\mu_A(u_1), \mu_A(u_2))$, $\lambda \in [0, 1]$.

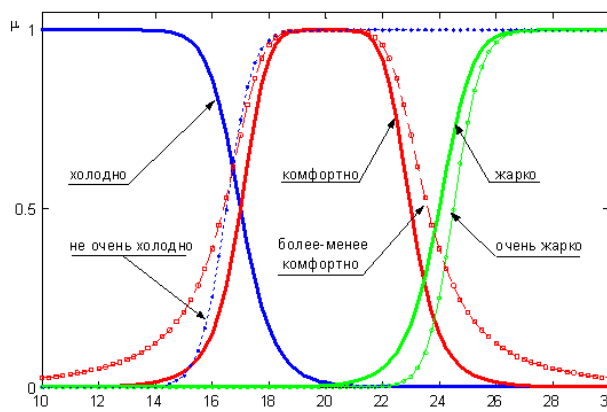
Естественно, нечеткие множества можно сравнивать, дополнять, пересекать и т.д. Пересечение их находится с использованием т.н. треугольных норм (t -норм). Они похожи на обычные и имеют свойства коммутативности, ассоциативности, ограниченности и монотонности. Вот, например, пересечение нечетких множеств с использованием различных t -норм:



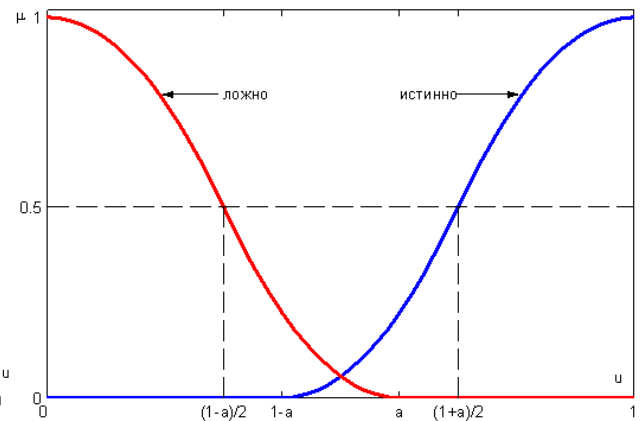
Объединение нечетких множеств находится с помощью т.н. конорм (S -норм):



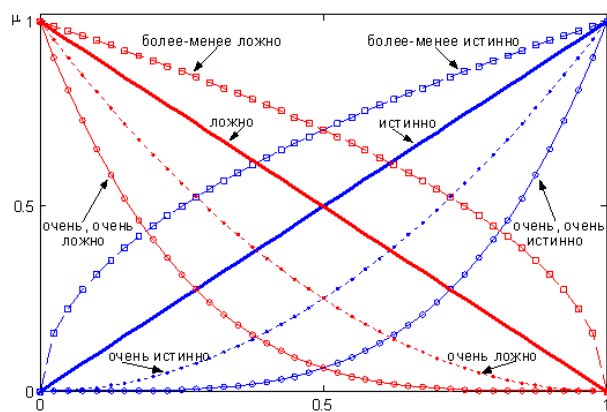
Вот еще несколько примеров использования нечетких множеств:



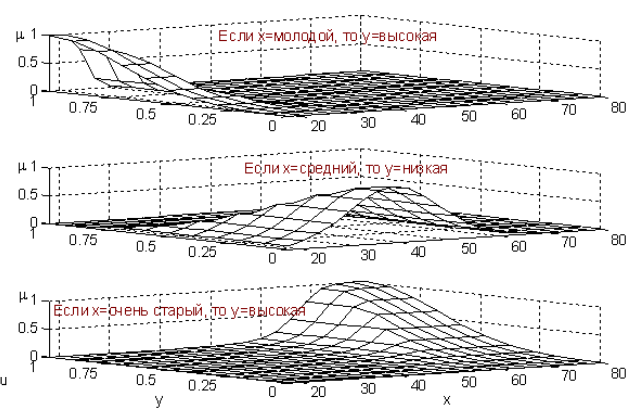
Лингвистическая переменная «температура»



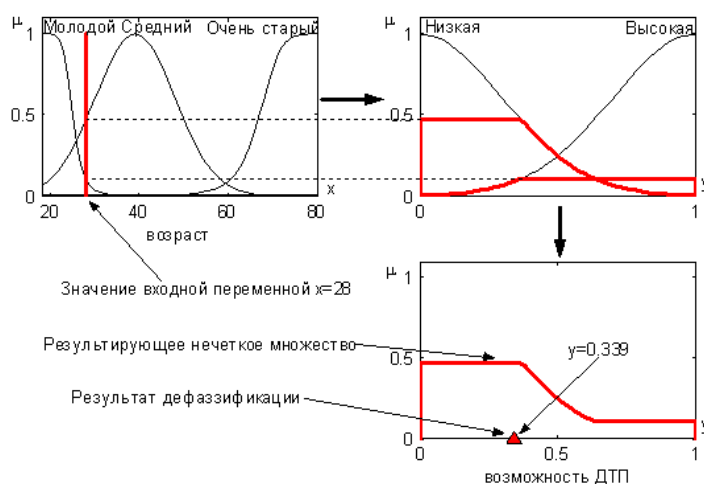
Истинность по Заде



Истинность по Бальдину



Нечеткие отношения, соответствующие правилам базы знаний «Зависимость вероятности аварии от возраста водителя»



Нечеткий логический вывод по Мамадани для примера выше

Таким образом, нечеткая логика позволяет эффективно проводить логический анализ любых событий, для которых нет четких критериев истинности. Наш мозг работает примерно таким же образом. Например, одна из сложнейших до появления нейропроцессоров задач для компьютера – распознавание изображения и сравнение его с данным в базе. С точки зрения классической логики лицо человека будет эффективно опознано машиной, как имеющееся в базе, если будет зафиксировано попиксельное совпадение двух фотографий. Конечно, человек действует совсем не так. Он узнает знакомого в толпе, даже если он внешне отличается от того лица, что заложено у человека в памяти – имеет другой цвет волос, причёску, шляпу и т.п. Естественно, оценка происходит с определенной степенью достоверности – сильно измененную внешность можно и не узнать, но всегда можно провести градацию – насколько примерно данный портрет соответствует запомненному образу знакомого – полностью, частично, очень похож и т.д. Это позволяет человеку эффективнее и в тысячи раз быстрее любого компьютера проводить сложнейший нечеткий анализ – узнавать звуки, запахи, людей, события и т.д.

Возникает вопрос, как воплотить нечеткую математику в железе? Машины, основывающие свою работу не на абсолютных, а вероятностных выводах, за свою аналогию с мозгом получили название нейрокомпьютеры.

Еще на заре компьютерной эры – до появления первых ЭВМ – были намечены два принципиально разных подхода к обработке информации: последовательный – обработка символов, и параллельный – распознавание образов. В данном контексте символы и образы – это слова, которыми оперируют вычислительные машины, их отличие лишь в размерности. Информационная емкость образа может на многие порядки превосходить число бит, которыми описываются символы. Но это чисто количественное различие имеет далеко идущие качественные последствия в силу экспоненциально быстрого возрастания сложности обработки данных с ростом их разрядности.

Когда в 1944 г. на головы сотрудников Муровской школы электротехники как снег неожиданно свалился легендарный фон Нейман, желающий ознакомиться с работами по созданию первого компьютера ENIAC, он нашел проект совершенно удивительным. Один из творцов машины Мокли в 1972 вспоминал об этом так: «Встретившись с ним, чтобы показать ENIAC и рассказать о ней, мы также обсудили наши идеи, касающиеся новой машины: она должна была содержать только одну адресуемую память, предназначенную для хранения, как данных, так и программ; в отличие от ENIAC все арифметические операции должны были выполняться в одном устройстве, а управление машиной должно быть централизовано... Джонни моментально все понял (быстрота мышления вообще была его характерной чертой). Но он предпочитал называть основные устройства машины, о которых мы ему рассказали, «органами», а ламповые схемы и другие блоки, выполнявшие логические функции, «нейронами». Таким образом, он видоизменил нашу терминологию, но это не повлияло на сущность того, о чем мы ему рассказали, и наша концепция следующей машины – EDVAC осталась неизменной».

Дело в том, что фон Нейман тогда находился под большим впечатлением от работы нейрофизиолога Уоррена Стёрджиса Мак-Каллоха (Warren Sturgis McCulloch, 1899-1969) и логика Уолтера Питтса (Walter Pitts, 1923-1969) «Логическое исчисление идей, относящихся к нервной активности», опубликованной в 1943 г. В ней авторы, в частности, показали, что машина Тьюринга может быть реализована с помощью конечной сети формальных нейронов, что послужило теоретической основой для разработки нейронных сетей. По иронии судьбы их архитектура (о ней ниже) диаметрально противоположна машине фон Неймана.

С этой работы принято отсчитывать эпоху нейрокомпьютеров, также называемых искусственные нейронные сети (ИНС). В чем их отличие от традиционной архитектуры? ИНС представляют собой сложную систему с обратной связью соединённых и взаимодействующих между собой простых процессоров (играющих роль искусственных нейронов). В таковом качестве обычно используют DSP – цифровые сигнальные процессоры. Каждый процессор подобной сети имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим процессорам.

Вторым важным отличием является то, что сеть не программируется в традиционном смысле слова, а обучается. С точки зрения машинного обучения, нейронная сеть представляет собой частный случай методов распознавания образов, дискриминантного анализа и методов кластеризации. С математической точки зрения, обучение нейронных сетей – это многопараметрическая задача нелинейной оптимизации. Технически обучение заключается в нахождении коэффициентов связей между нейронами. В процессе обучения нейронная сеть способна выявлять сложные зависимости между входными данными и выходными, а также выполнять обобщение. Это значит, что в случае успешного обучения сеть сможет вернуть верный результат на основании данных, которые отсутствовали в обучающей выборке, а также неполных и/или «зашумленных», частично искаженных данных, оперируя нечеткими алгоритмами.

После работ Мак-Каллоха и Питтса, а также Норберта Винера¹⁵ традиционные компьютеры все еще побеждали. Первые серийные коммерческие компьютеры IBM-650 появились в 1954 г. (за 15 лет их было продано 1,5 тысячи штук). В том же году был изобретен кремниевый транзистор.

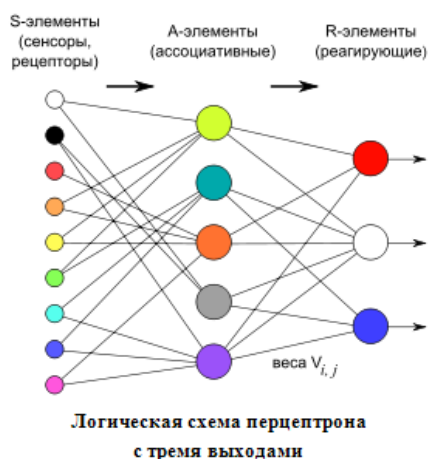
¹⁵ В 1948 он опубликовал программную работу «Кибернетика», в которой основной идеей является представление сложных биологических процессов математическими моделями. Несмотря на активную разработку этой темы, она мало что подарила архитектуре ЭВМ и являлась, скорее, развитием теории автоматического управления, об этом см. соответствующее приложение. Однако, несмотря на полную техническую бесполезность для развития компьютеров, кибернетика имела важный философский аспект – она послужила толчком не только когнитивной психологии, но и всеобщей уверенности в существовании искусственного интеллекта, что, в свою очередь, опосредованно повлияло на бум нейрокомпьютеров.

В 1961 г. появился его полевой аналог, используемый в микросхемах, а в 1962-м началось массовое производство микросхем. В 1970 г. фирма Intel выпустила свой первый микропроцессор, дав старт микропроцессорной революции 70-х. Всего через четыре года появился первый персональный компьютер Altair по весьма доступной цене 397 долларов, правда, еще в виде набора деталей, подобно тому, как тогда продавались комплекты для радиолюбителей. В 1977-м появился первый дружелюбный к пользователю компьютер Apple-2 с удобным интерфейсом, а в 1981 г. на рынок ПК вышел гигант IBM, с легендарным IBM PC – первым персональным компьютером в современном смысле слова. В результате в 90-х годах типичный пользователь уже имеет на своем рабочем столе аналог суперкомпьютера Cray-1.

Что касается нейросетевой архитектуры, то, несмотря на многочисленные реверансы в сторону нейронных сетей со стороны классиков кибернетики, их влияние на промышленные разработки вплоть до недавнего времени было минимальным. В 1949 г. Дональд Хэбб (Donald Hebb) выпустил книгу «Организация поведения», где исследована проблематика настройки синаптических связей между нейронами, он же предложил первый алгоритм обучения нейросети.

Первый экспериментальный нейрокомпьютер Snark был построен Марвином Минским в 1951 году, однако он не был приспособлен к решению практически интересных задач. В то же время Натаниелом Рочестером (Nathaniel Rochester) из исследовательской лаборатории IBM были проведены независимые работы по синтезу нейросетей.

В 1958 г. Джон фон Нейман предложил имитацию простых функций нейронов с использованием вакуумных трубок. Наконец, решающий прорыв совершил нейрофизиолог Фрэнк Розенблатт, предложивший схему устройства, моделирующего процесс человеческого восприятия. Его перцептрон (иногда персептрон,

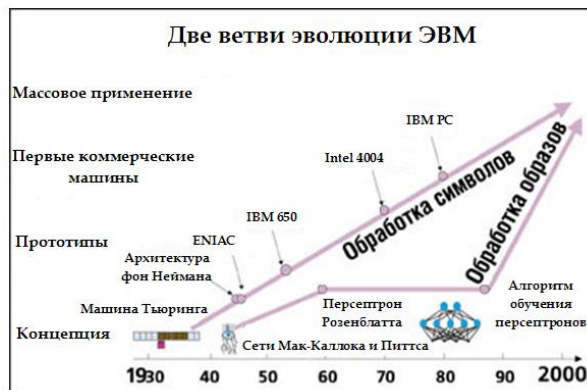


perceptron от лат. *perceptio* — восприятие) имел рецепторное поле из матрицы фотоэлементов 20x20, которая передавала сигналы в блоки электромеханических ячеек памяти. Эти ячейки соединялись между собой случайным образом в соответствии с принципами коннективизма. В 1957 году в Корнелльской Лаборатории Аэронавтики успешно было завершено моделирование работы перцептрона на компьютере IBM 704, а два года спустя, в Корнелльском университете, был продемонстрирован первый нейрокомпьютер — «Марк-1», который был способен распознавать некоторые буквы английского алфавита.

Чтобы «научить» перцептрон классифицировать образы, был разработан специальный итерационный метод обучения проб и ошибок, напоминающий процесс обучения человека — метод коррекции ошибки. Кроме того, при распознавании той или иной буквы перцептрон мог выделять характерные особенности буквы, статистически чаще встречающиеся, чем малозначимые отличия в инди-

видуальных случаях. Тем самым перцептрон был способен обобщать буквы, написанные различным образом (почерком), в один обобщенный образ. Однако возможности перцептрона были ограниченными: машина не могла надежно распознавать частично закрытые буквы, а также буквы иного размера, расположенные со сдвигом или поворотом, нежели те, которые использовались на этапе ее обучения.

Отчёт по первым результатам появился ещё в 1958 году — тогда Розенблатт опубликовал статью «Перцептрон: Вероятная модель хранения и организации информации в головном мозге». Но подробнее свои теории и предположения относительно процессов восприятия и перцептронов он описывает в книге «Принципы нейродинамики: Перцептроны и теория механизмов мозга» (1962). В книге он рассматривает не только уже готовые модели перцептрона с одним скрытым слоем, но и многослойных перцептронов с перекрёстными и обратными связями. В книге также вводится ряд важных идей и теорем, например, доказывается теорема сходимости перцептрона.



ной терминологии, перцептроны могут быть классифицированы как искусственные нейронные сети с одним скрытым слоем; с пороговой передаточной функцией; с прямым распространением сигнала.

Перцептрон Розенблатта обретает популярность – его используют для распознавания образов, прогнозирования погоды и т. д. Казалось, что построение полноценного искусственного интеллекта уже не за горами. К моменту создания перцептрона завершилось расхождение теоретических работ Мак-Каллоха с т.н. «кибернетикой» Винера. Осознав ее полную бесполезность, Мак-Каллох и его последователи вышли из состава «Кибернетического клуба».

В 1960 году Бернард Уидроу (Bernard Widrow) совместно со своим студентом Марсианом Хоффом (Marcian Hoff) на основе дельта-правила (формулы Уидроу) разработали ADALINE и MADALINE (Множественные Адаптивные Линейные Элементы - Multiple ADaptive LINear Elements), которые сразу же начали использоваться для задач предсказания и адаптивного управления. ADALINE был построен на базе созданных ими же принципиально новых элементов – мемисторах, о которых мы еще поговорим. Сейчас адаптивный сумматор является стандартным элементом многих систем обработки сигналов в DSP. MADALINE же действовала как адаптивный фильтр, устраняющий эхо на телефонных линиях. Эта нейросеть до сих пор используется по назначению.

В 1963 году в Институте проблем передачи информации АН СССР А. П. Петровым проводится подробное исследование задач, «трудных» для перцептрона. Эта пионерская работа в области моделирования ИНС в СССР послужила отправной точкой для комплекса идей М. М. Бонгарда — как «сравнительно небольшой переделкой алгоритма (перцептрона) исправить его недостатки». Работы А. П. Петрова и М. М. Бонгарда весьма поспособствовали тому, что в СССР первая волна эйфории по поводу ИНС была сглажена.

Сам Розенблатт выделил два фундаментальных ограничения для трёхслойных перцептронов (состоящих из одного S-слоя, одного А-слоя и R-слоя): отсутствие у них способности к обобщению своих характеристик на новые стимулы или новые ситуации, а также неспособность анализировать сложные ситуации во внешней среде путём расчленения их на более простые.

Но, несмотря на это, тогда же возникли первые коммерческие нейрокомпьютерные компании. Энтузиазм того героического периода был так велик, что многие, увлекшись, предсказывали появление думающих машин в самом ближайшем будущем. Эти пророчества раздувались прессой до совсем уже неприличных масштабов, что, естественно, отталкивало серьезных ученых.

В 1969 году бывший однокашник Розенблатта по Высшей научной школе в Бронксе Марвин Минский, сам в свое время отдавший дань конструированию нейрокомпьютеров, решил положить этому конец, выпустив вместе с южноафриканским математиком Сеймуром Пейпертом книгу «Перцептроны». В этой роковой для нейрокомпьютеринга книге была строго доказана принципиальная, как тогда казалось, ограниченность перцептронов. Утверждалось, что им доступен лишь очень узкий круг задач.

Они математически показали, что перцептроны, подобные розенблаттовским, принципиально не в состоянии выполнять многие из тех функций, которые хотели получить от перцептронов. К тому же в то время была слабо развита теория о параллельных вычислениях, а перцептрон полностью соответствовал принципам таких вычислений. По большому счёту, Минский показал преимущество последовательных вычислений перед параллельным в определённых классах задач, связанных с инвариантным представлением. Его критику можно разделить на три темы:

- Перцептроны имеют ограничения в задачах, связанных с инвариантным представлением образов, то есть независимым от их положения на сенсорном поле и относительно других фигур. Такие задачи возникают, например, если нам требуется построить машину для чтения печатных букв или цифр так, чтобы эта машина могла распознавать их независимо от положения на странице (то есть, чтобы на решение машины не оказывали влияния перенос, поворот, растяжение-сжатие символов); или если нам нужно определить, из скольких частей состоит фигура; или находятся ли две фигуры рядом или нет. Минским было доказано, что этот тип задач невозможно полноценно решить с помощью параллельных вычислений, в том числе — перцептрона.

- Перцептроны не имеют функционального преимущества над аналитическими методами (например, статистическими) в задачах, связанных с прогнозированием. Тем не менее, в некоторых случаях они представляют более простой и производительный метод анализа данных.

- Было показано, что некоторые задачи в принципе могут быть решены перцептроном, но могут потребовать нереально большого времени или нереально большой памяти.

В действительности критика относилась лишь к перцептронной сети с одним слоем обучающихся нейронов. Но для многослойных нейронных сетей алгоритм обучения, предложенный Розенблаттом, не годился. Холодный душ критики, умерив пыл энтузиастов, затормозил развитие нейрокомпьютеринга на многие годы. Это привело к смещению интереса исследователей искусственного интеллекта в противоположную от нейросетей область символьных вычислений. Кроме того, из-за сложности математического анализа перцептронов, а также отсутствия общепринятой терминологии, возникли различные неточности и заблуждения. Теперь мы уже знаем, чем закончился символьный подход — взлетом и падением логического языка Prolog и крахом многомиллиардной японской программы строительства суперкомпьютеров 5-го поколения

в 80-е годы, похоронившей японскую индустрию разработки ЭВМ. Персептроны же живут до сих пор и весьма неплохо. Как так произошло? В 70-е для нейрокомпьютинга настали тяжелые времена – все увлеклись логическим выводом или векторными и матричными процессорами, а также первыми компьютерными сетями.

Из значимых для нас событий можно отметить предложенную в 1973 г. Б. В. Хакимовым нелинейную модель с синапсами на основе сплайнов и внедренную им для решения задач в медицине, геологии и экологии. В 1974 г. Пол Дж. Вербос и А. И. Галушкин одновременно изобретают алгоритм обратного распространения ошибки для обучения многослойных перцептронов, но эпохальное изобретение не привлекло особого внимания и лишь спустя 12 лет было переоткрыто, что привело к второй нейрореволюции. Наконец, в 1975 г. Фукусима представляет «Когнитрон» — самоорганизующуюся сеть, предназначенную для инвариантного распознавания образов, но это достигается только при помощи запоминания практически всех состояний образа.

Интерес широкой научной общественности к нейросетям пробудился после теоретической работы физика Джона Хопфилда (1982), предложившего модель ассоциативной памяти в нейронных ансамблях. Хопфилд и его многочисленные последователи обогатили теорию нейросетей многими идеями из арсенала физики, такими как коллективные взаимодействия нейронов, энергия сети, температура обучения и т. д.

Хопфилд показал, что нейронная сеть с обратными связями может представлять собой систему, минимизирующую энергию (сеть Хопфилда). Кохоненом представлены модели сети, обучающейся без учителя (сеть Кохонена), решающей задачи кластеризации, визуализации данных (самоорганизующаяся карта Кохонена) и другие задачи предварительного анализа данных. Неудивительно, что вскоре после этого в 1983 г. нейроисследования, наконец, получили финансирование от Агентства перспективных военных исследований США (DARPA). Этот факт стал сигналом к возобновлению интереса ученых к нейросетям.

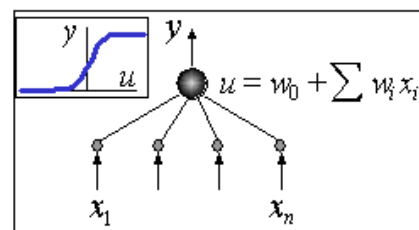
Однако настоящий бум практического применения нейросетей начался после публикации в 1986 году Дэвидом Румельхартом с соавторами Дж. Е. Хинтоном и Рональдом Дж. Вильямсом и независимо и одновременно С. И. Барцевым и В. А. Охониным (Красноярская группа) переоткрытого и значительно продвинутого метода обучения многослойного персептрона, названного ими методом обратного распространения ошибки (error back-propagation). Ограничения персептронов, о которых писали Минский и Пейперт, оказались преодолимыми, а возможности вычислительной техники – достаточными для решения широкого круга прикладных задач.

Наконец, в 90-х годах распространенность ПК и их производительность возросла настолько, что позволило массово и всем желающим легко моделировать программно с их помощью работу параллельных нейронных сетей с числом нейронов от нескольких сотен до десятков тысяч. Такие эмуляторы нейросетей способны решать многие интересные с практической точки зрения задачи. В свою очередь, нейросетевые программные комплексы станут тем носителем, который выведет на технологическую орбиту настоящее параллельное нейросетевое hardware.

Отличительной чертой нейросетей является глобальность связей. Базовые элементы искусственных нейросетей – формальные нейроны – изначально нацелены на работу с векторной информацией. Каждый нейрон нейросети, как правило, связан со всеми нейронами предыдущего слоя обработки данных.

Специализация связей возникает лишь на этапе их настройки – обучения на конкретных данных. Архитектура процессора или, что для нейросети то же самое, алгоритм решения конкретной задачи проектируется сам по мере обучения. Каждый формальный нейрон производит простейшую операцию - взвешивает значения своих входов со своими же локально хранимыми синаптическими весами и производит над их суммой нелинейное преобразование:

Нелинейность выходной функции активации принципиальна. Если бы нейроны были линейными элементами, то любая последовательность нейронов также производила бы линейное преобразование, и вся нейросеть была бы эквивалентна одному слою нейронов. Нелинейность разрушает линейную суперпозицию и приводит к тому, что возможности нейросети существенно выше возможностей отдельных нейронов.



Нейрокомпьютеры имеют следующие важные преимущества:

- Естественный параллелизм обработки информации из-за глобальности связей между нейронами. До обучения эти связи произвольны и обычно малы. Обучение на примерах подгоняет конкретную структуру сети под конкретную задачу, что приводит к колоссальной скорости работы по сравнению с традиционным компьютером.
- Единый и эффективный принцип обучения нейросетей – минимизация эмпирической ошибки методом ее обратного распространения по сети. Извне задается лишь цель обучения – то есть способ определения ошибки по выходам сети. Далее сеть постепенно модифицирует свою конфигурацию, минимизируя эту ошибку, все лучше справляясь с возложенной на нее задачей.

- Надежность функционирования. Избыточность связей приводит к тому, что значения каждого веса по отдельности не играют решающей роли. Вывод из строя ограниченного числа нейронов или обрыв некоторых связей не сказываются критическим образом на качестве работы всей сети.

- Способность решать неформализованные задачи следует из способности нейросетей самостоятельно вырабатывать весьма сложные алгоритмы обработки данных, формализовать которые самостоятельно зачастую не могут даже лучшие эксперты в данной предметной области.

Естественно, у нейрокомпьютеров есть и недостатки.

- Во-первых, для создания приемлемой модели необходимо около 100 наблюдений. Это достаточно большое число данных и существует много случаев, когда такое количество исторических данных недоступно. Даже для прогнозирования такой тривиальной вещи, как спрос на конкретный товар на следующий месяц, необходима статистика за предыдущие 7-10 лет его продаж. Это, конечно, нереально – товар может лишь недавно выйти на рынок или уже неоднократно устареть и обновиться, испортив всю математическую модель, а если он без изменений успешно продается более 10 лет, то нам и не понадобится никакая нейросеть для прогноза его 100% успеха в следующем месяце.

- Во-вторых, помимо значительных затрат по времени и другим ресурсам для построения удовлетворительной модели, известно, что обучение сети может занимать довольно много времени, причем заранее это время малопредсказуемо. Кроме этого, существует много непредсказуемых и формально сложно представимых факторов. Например, упомянутая выше модель 100% успешного товара все равно может потерпеть крах, если вдруг появится полный его аналог, но значительно дешевле, хотя, опять же, это верно не всегда и в общем случае требует оценки огромного количества дополнительных параметров, которые зачастую недоступны (типа статистики не только своей фирмы, но и взлетов и падений конкурентов за те же 10 лет). В результате, для составления адекватной модели, допускающей применение нейросети, требуются усилия огромного количества профильных специалистов, причем на выходе получаем монозадачную конструкцию, для переобучения которой потребуются столько же сил. Именно поэтому нейросети малоприменимы в таких ценных и удобных, казалось бы, вещах, как постановка диагнозов или игра на рынке, тем более что риск, в случае провала, здесь огромен – можно потерять минимум деньги, максимум жизнь.

- В-третьих, исходя из вышесказанного, разработка, обучение и интерпретация результатов работы по-настоящему серьезной и эффективной нейросети, при ее монозадачности, требует привлечения большого числа непрофильных специалистов, а не только информатиков и математиков – экономистов, врачей, технологов и т.д., что резко увеличивает стоимость действительно серьезных нейропродуктов. При этом при изменении модели сеть надо переобучать, что тоже недешево.

- Наконец, последним недостатком является практически полная невозможность использовать нейросеть как обычную универсальную ЭВМ – сложно производить вычисления, когда ответ на вопрос «умножить 2 на 2» выдается в форме «примерно 4, для большей точности продолжите обучение»

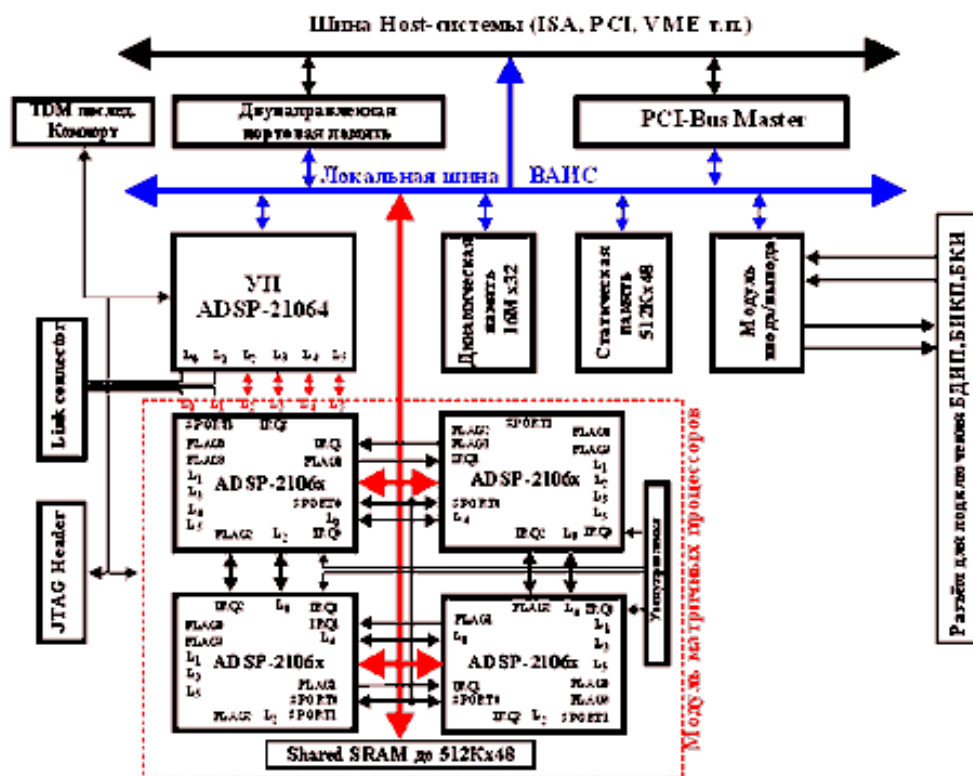
Какова же элементная база нейромашин? Стоит отметить, что традиционные «железные» нейрокомпьютеры уже устарели. Указанные выше недостатки заставляют отдавать предпочтение более дешевому, универсальному и концептуально простому аппаратному воплощению, например систолическим массивам, стековым мультиядерным машинам на FPGA-матрицах, GPGPU и прочим, даже в тех случаях, когда соответствующая нейросеть справилась бы лучше, скажем в задачах распознавания изображения. Большая часть современных нейромашин существует лишь в виде программных комплексов.

Тем не менее, существовало некоторое количество физических нейромашин. Элементной базой нейровычислительных систем второго и третьего поколений (середина 1980-х и середина 1990-х, соответственно) являлись по порядку процессоры общего назначения (CPU), специально изготовленные нейрочипы (ASIC), транспьютеры (TPU), встраиваемые микроконтроллеры (MC) и наиболее современные – цифровые сигнальные процессоры (DSP) и ПЛИС (FPGA).

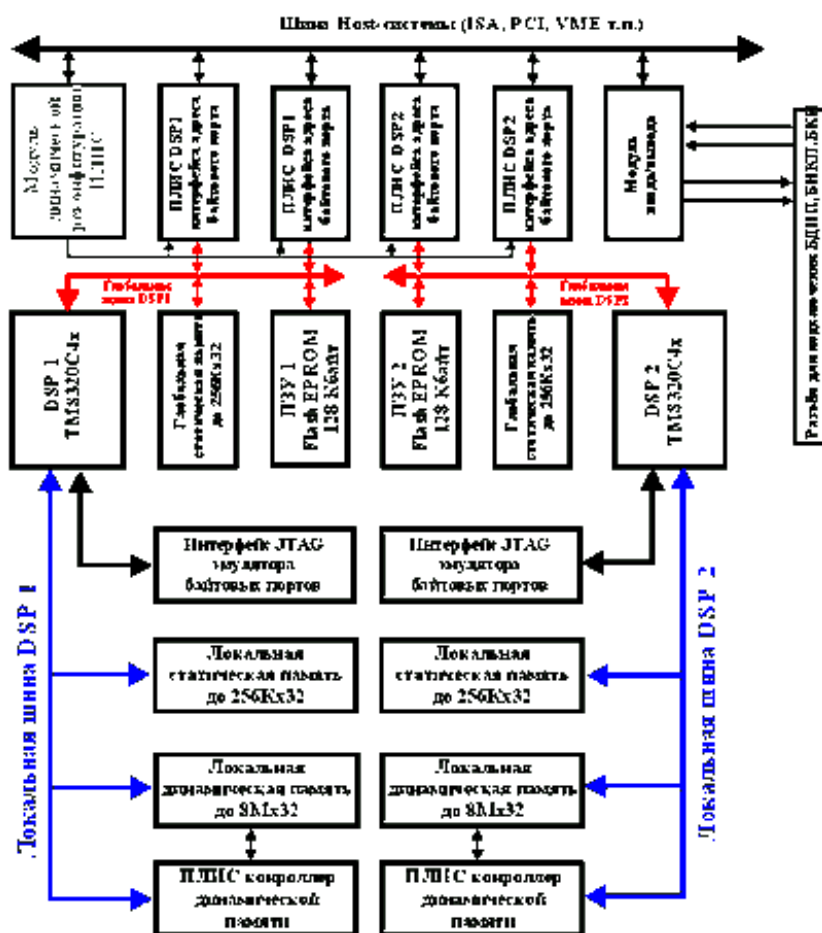
Примером относительно современного DSP-процессора, приспособленного для ускорения нейровычислений, является 5-ядерный Texas Instruments TMS320C80, ядра – независимые DSP. Большая производительность, требуемая при обработке сигналов в реальном времени, побудила Texas Instruments и Analog Devices выпустить транспьютероподобные семейства микропроцессоров TMS320C4x и ADSP2106x, ориентированные на использование в мультипроцессорных системах.

Похожими проектами в 1989-1999 гг. занималась отечественная фирма «Модуль», за десять лет разработавшая и выпустившая процессор «Neuro Matrix», вполне сравнимый производительностью с TS и AD. Естественно, продукты Analog Devices и Texas Instruments – не нейропроцессоры, а DSP, транспьютеры, микроконтроллеры и гарвардские машины¹⁶, такие, как TigerSHARC, но их вполне можно успешно использовать в качестве узлов нейросети.

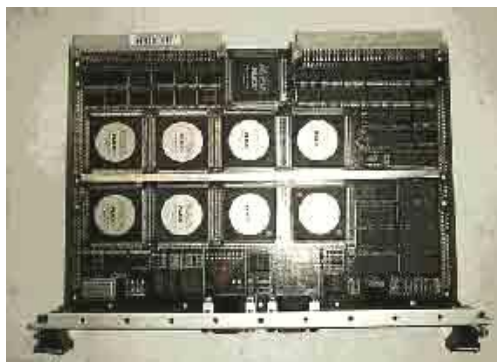
¹⁶ Классификация по шине команд и памяти, к ней мы еще вернемся далее.



Реализация ИИС на основе ADSP2106x

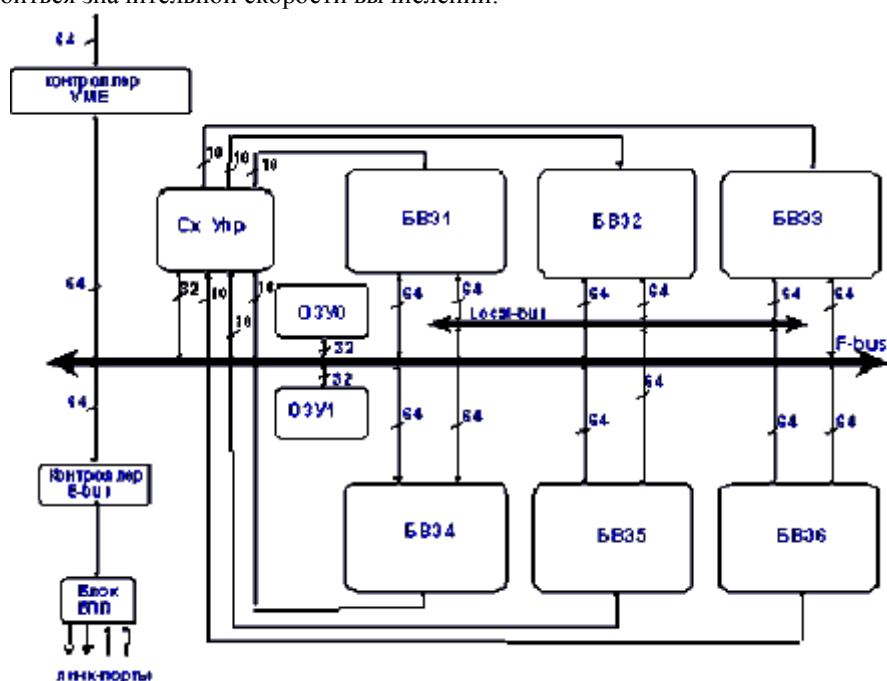


Реализация ИИС на основе TMS320C4x.



На таких чипах собирали целые нейроплаты и даже построенные на них компьютеры. Слева, например, нейроплата ППВ (Параллельный перепрограммируемый вычислитель), собранная на FPGA, скорее всего в конце 1990-х в НИИ Системных исследований (РАН). ППВ разработана в стандарте VME и реализована на базе перепрограммируемых микросхем семейства 10K фирмы Altera. Вычислитель предназначен для работы в качестве аппаратного ускорителя и является ведомым устройством на шине VME. Он должен включаться в систему как подчиненное устройство основной управляющей host-машины с универсальным процессором. Таковая частота вычислителя 33 МГц. Теоретически плата ППВ используется для построения систем распознавания

образов на основе обработки телевизионной, тепловизионной и другой информации, а также систем, основанных на реализации алгоритмов с пороговыми функциями и простейшими арифметическими операциями, и позволяет добиться значительной скорости вычислений.



Структурная схема ППВ

Вычислитель состоит из следующих функциональных блоков:

- схема управления (Сх Упр);
- базовые вычислительные элементы (БВЭ1-БВЭ6);
- контроллер внешней шины (Контроллер E-bus);
- контроллер системной шины (Контроллер VME);
- два массива статической памяти (ОЗУ0, ОЗУ1);
- блок высокоскоростных приемников/передатчиков.

Схема управления используется для управления БВЭ и потоками данных в вычислителе и представляет собой простейший RISC-процессор. Структура и набор команд процессора могут изменяться в зависимости от типа решаемой задачи. БВЭ используются для выполнения простейших арифметических операций типа суммирования, вычитания, умножения и вычисления пороговых функций. Так как БВЭ реализованы на перепрограммируемых микросхемах, их архитектура может изменяться. Архитектура БВЭ для различных алгоритмов может отличаться, но обычно легко реализуется путем комбинации библиотечных функций, компиляции их при помощи САПР (типа MaxPlus) и загрузки файла конфигурации в выбранный БВЭ. Два массива локальной статической памяти собраны из 8 микросхем статической памяти емкостью 0,5 Мбайт, имеют размер 4Мбайт и организованы как массив 512К 8-байтовых слов. Массивы памяти связаны со схемой управления отдельными адресными шинами и могут функционировать независимо друг от друга. Память предназначена для хранения общих коэффициентов, а также промежуточных результатов вычислений или окончательных результатов, подготовленных к передаче через контроллер системной шины в центральный процессор или через контроллер E-bus на линейные порты. Связь нескольких вычислителей между собой или вычислителя с устройством оцифровки изображения, при наличии у устройства оцифровки соответствующего

интерфейса, осуществляется последовательным каналом приемников/передатчиков HOTLink фирмы CYPRESS. Управление передачей данных выполняет контроллер внешней шины, который представляет из себя набор 4-х стандартных FIFO и регистров управления и данных. Контроллер шины VME выполняет функцию интерфейса с центральным процессором и является стандартным устройством. С точки зрения программиста, вычислитель можно представить как RISC-процессор (схема управления или управляющий процессор) и шесть векторных процессоров (вычислительных элементов), обрабатывающих SIMD-команды (одна команда для многих данных). Большое количество шин данных, возможность одновременной работы всех БВЭ и выполнение арифметических операций умножения и сложения за один такт позволяет эффективно распараллеливать процесс обработки информации.

Плата показывала неплохую производительность, например, ниже дано сравнение по скорости нескольких популярных алгоритмов, которые она выполняла с известными процессорами тех лет.

Название алгоритма	Pentium-100, с	PentiumII-333, с	Ultra SPARC, с	ППВ, с
Свертка с ядром 4x4 *	0.65	0.11	0.76	0.02
Медианный фильтр	1.97	0.49	0.75	0.001
Повышение контрастности	0.51	0.13	1.31	0.004
Прямое поточечное сравнение с маской 32x32 **	43.78	7.14	58.89	0.142
Поиск локальных неоднородностей 32x32	0.120	0.028	0.146	0.032
Умножение матрицы на матрицу	8.61	0.60	12.31	0.011

*Оценки приведены для: Pentium-100 при частоте 100 МГц, объем ОЗУ 16 Мбайт;

**Pentium-333 при частоте 350 МГц, объем ОЗУ 128 Мбайт; UltraSPARC при частоте 200 МГц, объем ОЗУ 64 Мбайт; ППВ при частоте 33 МГц.

Вот еще несколько древних плат, собранных на разной элементной базе, от FPGA до транспьютеров:

DSP60V6 – мультипроцессорный модуль сбора и цифровой обработки сигналов, основанный на ADSP-21060/62 SHARC. Он позволял выполнять программы SHARC во взаимодействии с устройствами, размещенными на дочерней плате ADM, в качестве которой могли использоваться модули АЦП-ЦАП. Он устанавливался в промышленные крейты 6U и мог работать как автономно, так и с компьютером, имеющим шину VME. Через пользовательские выводы разъема J2/P2 (X2), можно было загружать программы и данные, осуществлять сброс процессоров, просматривать память и инициировать выполнение программ.



ADP44PCI – процессорный модуль ориентирован на цифровую обработку сигналов. Он мог быть легко конфигурирован для решения конкретных задач путем загрузки прикладного ПО

и подключения submodule аналогового ввода/вывода и/или модуля стандарта TIM-40. Несколько ADP44PCI образуют распределенную вычислительную структуру из процессорных модулей, соединенных между собой с помощью высокоскоростных связных портов. Процессорный несущий модуль ADP44PCI использует параллельный 32-разрядный ЦПОС с плавающей точкой TMS320C44 фирмы Texas Instruments. Четыре 30Мбайт/сек связных порта и канала ПДП процессора обеспечивают разнообразные возможности высокоскоростного обмена. Две шины TMS320C44 работают на скорости 120 Мбайт/сек каждая. Работая совместно, но независимо, процессор и ПДП TMS320C44 обеспечивают параллельный обмен данными со скоростью до 560 Мбайт в секунду. Высокоскоростные связные порты позволяют объединять процессоры в кольца, деревья, гиперкуб и др.



Модуль ADP6701PCI фирмы «Инструментальные системы» был построен на базе процессора TMS320C670. Имеет в своем составе восемь параллельных вычислительных блоков. Отличительными особенностями платы являются: буферная память тракта ввода 64Kx32, вывода 64Kx16, двухпортовая статическая память 64Kx32, синхронная динамическая память до 16 Мбайт. Благодаря гибкому аппаратному интерфейсу на FPGA, обеспечивается программная совместимость различных submodule.

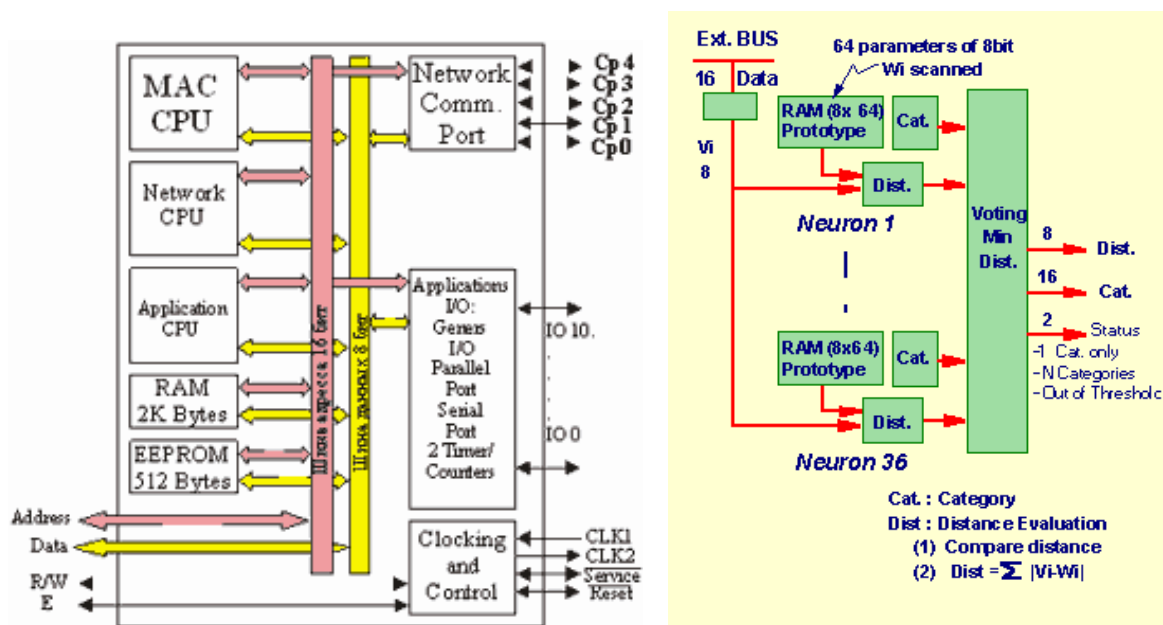
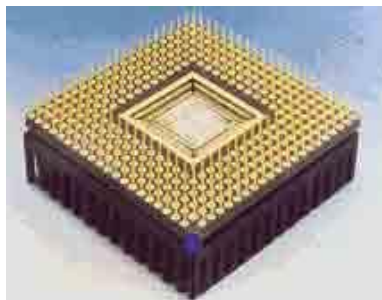
Впрочем, намного интереснее были настоящие нейропроцессоры, разрабатываемые изначально в таком качестве. Пик пришелся на середину 1990-х, вот лишь некоторые из них:

Модель	Изготовитель	Разрядность, бит	Максимальное количество синапсов*	Максимальное число слоев**	Примечание
MA16	Siemens	48 (умножители и сумматоры)	-	-	400 ММАС.
NNP (Neural Networks Processor)	Accurate Automation	Nx16	-	-	MIMD, N - число процессоров.
CNAPS-1064	Adaptive Solutions	16	128 Кбайт	64	
100 NAP Chip	HNC	32	512 Кбайт	4	4 процессорных элемента
Neuro Matrix NM6403.	Модуль, Россия	64 (вект. процессор), 32 RISC ядро	4096 шт.	24	Совместим с TMS320C4x
CLNN 32 CLNN 64	Bellcore	32 64	496 1024	32 нейрона	10 ⁸ перекл./с 2 x 10 ⁸ перекл./с
NC 3001	NeuriGam	16	4096 шт.	32	
ZISC 036 (Zero Instruction Set Computer)	IBM	64 разр. входного вектора	-	36 нейронов	Частота 20МГц, Векторно-прототипный нейрочип
ETANN 80170NW	Intel	64 входа	Два банка весов 64x80	64 нейрона в слое, 3 слоя.	Аналоговая
MD-1220	Micro Devices	16	64 шт.	8	8 нейронов
MT 19003 - Neural Instruction Set Processor	Micro Circuit Engineering (MCE)	16 разр. Умножитель 35 разр. сумматор	-	1	RISC МП с 7 специальными командами
Neuro Fuzzu	National Semiconductor	-	-	-	
NI 1000	Nestor	5-16 (одного нейрона)	-	1024 прототипных 256 мерных векторов	Векторно-прототипный нейрочип
NLX420 (NLX 110, 230)	Adaptive Logic	16	1 Мбайт	16	16 процессорных элементов
OBL Chip	Oxford Computer	16	16 Мбайт	-	
L-Neuro 1.0 L-Neuro 2.3	Philips	16 16	1536	16 нейронов 192 (12x16)	26 МГц 60 МГц
RSC (Speech Recognition Chip) - 164	Sensory Circuits	-	-	-	
ORC 110xx (Object Recognizer Chip)	Synaptics	-	-	-	
Pram-256 Chip	UCLi Ltd.	8 (одного нейрона)	-	256 нейронов	33МГц.
SAND	Datafactory	16	-	4	200 MCPS
ACC		16	-	-	
Геркулес	Россия	16	1 Мбайт	64	
Neuro Classifier	Университет Твента, DESY	70 вх. нейронов	-	6 (внутр) 1 вх., 1 вых.	2 x 1010 перекл./с
ANNA	AT&T	Число нейроннов 16-256	4096 весов	-	Число входов у нейрона 256-16.
WSC (Wafer Scale Integration)	Hitachi	-	64 связи на нейрон	576 нейронов	
SASLM2	Mitsubishi	2 (одного нейрона)	-	4096(64x64) нейронов	50 МГц
TOTEM	Kent (Univer UK), di Trento (Italy)	16 (одного нейрона)	-	64 нейрона	30 МГц

* - максимальное число синапсов определяет размер внутрикристалльной памяти весов.

** - максимальное число слоёв определяется числом операций умножения с накоплением, выполняемых за один такт для операндов длиной 8 бит.

Вот, например, конструктивная реализация NeuroMatrix 6403 и нейрочипа Siemens MA16:



Функциональная схема нейрочипа Echelon NEURON 3150 и IBM ZISC-036

Среди наиболее мощных и совершенных проектов были нейровычислители Synapse, разработанные компанией Siemens Nixdorf Informationssysteme (SNI) – дочернее предприятие концерна Siemens в сотрудничестве с Маннгеймским университетом. Нейрокомпьютер Synapse 1 появился на рынке в середине 1994 г., в дальнейшем были выпущены нейроускорители Synaps 2 и Synaps 3. Сборка осуществлялась в Европе, затем какое-то время ускорители распространялись французской фирмой Tiga Technologies.



Плата Synapse 1 и базовый комплект SYNAPSE1-N110 предполагают наличие главной ЭВМ – рабочей станции SUN SPARCSTATION 5 модели TX1 в качестве вспомогательного консолидирующего устройства, облегчающего процессы программирования, проектирования нейросетей, тестирования, управления внешними устройствами, вывода результатов и т.п. Главная ЭВМ сопрягается с аппаратурой SYNAPSE1 через шину VME.

В России похожий проект носил название «Эмбрион» и был разработан под руководством В.Д. Цыганкова. Было известно несколько модификаций данного нейрокомпьютера для различных приложений: датчик случайных многомерных управляемых импульсных потоков «Эмбрион-1»; техническая диагностика неисправностей энергогенератора самолетной электростанции «Эмбрион-2»; управление летательным аппаратом в реальном времени («Эмбрион-3,4»); орган технического зрения («Эмбрион-5»); управление тактильно

чувствительным адаптивным промышленным роботом при обслуживании карусельной плавильной печи на стекольном заводе («Поиск-1») и др.

Все эти замечательные платы в полной мере обладали вышеперечисленными достоинствами и недостатками. Развитие ПК привело к тому, что современный эмулятор нейросети на 12-ядерном Intel Xeon по производительности и простоте создания превзойдет любой специализированный аппаратный нейрокомплекс. Существует несколько проектов создания аналоговых нейромашин с использованием принципиально иной элементной базы, например, аналоговые микросхемы с локальными связями между элементами (т.н. клеточные нейросети, CNN - Cellular Neural Networks), аналоговая оптоэлектроника, биокомпьютеры и ДНК-машины и т.д.). Однако все эти технологии дороги, сложны, и зачастую реализуемы лишь на бумаге.

В результате мечты пионера российского нейрокомпьютинга Ф.В. Широкова, озвученные им в 1998 г. – «системы промышленной автоматизации, построенные на VME, обретут нейроморфный мозг, способность видеть и слышать, ощущать электрические и магнитные поля, воспринимать ультразвук и радиацию. Они смогут анализировать обстановку и принимать решения. Это будет прививкой разума системам промышленной автоматизации», остаются такими же невоплотимыми, как и подобные высказывания 1950-х.

Нейромашин не оказались панацеей и не помогли создать искусственный интеллект, но значительно продвинули информатику, математику и когнитивную психологию, а также породили несколько вполне успешных прикладных программных пакетов.

1.3.4. NULL Convention Logic и асинхронные машины

Практически все известные процессоры синхронны. Для современного процессора работать в синхронном режиме также естественно, как нам с вами говорить прозой. В роли устройства синхронизации выступает тактовый генератор, и увеличение его частоты автоматически поднимает производительность всех тактуемых им устройств.

Впрочем, на общем «синхронном» фоне были отдельные альтернативные проекты. В качестве удачного примера компьютера без тактового генератора можно привести построенный в начале 50-х годов в Иллинойском университете ORDVAC (Ordnance Variable Automated Computer), который с успехом применялся для управления артиллерийским огнем. В истории компьютеров оставил заметный след асинхронный проект IAS, выполнявшийся под руководством Джона фон Неймана. В течение долгих лет асинхронными компьютерами занимались в Манчестерском университете, где был построен MU5. Однако позже идеи построения асинхронных процессоров оказались вне сферы интересов крупных производителей.

В середине 90-х годов интерес к почти забытым асинхронным компьютерам пробудился вновь. Импульсом к реинкарнации послужил программный документ с внушительным названием National Roadmap for Semiconductors, подготовленный американской Ассоциацией производителей полупроводников в 1997 году. Этот документ содержал предупреждение о грядущем кризисе, который может возникнуть спустя какое-то время и затронуть всю полупроводниковую индустрию. По мнению большинства членов Ассоциации, угроза кроется не в достижении теоретического предела ширины проводника или той же тактовой частоты, а в самом ее наличии. Попробуем разобраться, так ли это?

Координацию деятельности в области разработки асинхронных процессоров осуществляет симпозиум International Symposium on Advanced Research in Asynchronous Circuits and Systems, который с 1994 года проводит ежегодные встречи.

Главная беда синхронности заключается в том, что она требует для своей реализации весьма сложный механизм управления, который может занимать от 10 до 50% общего числа транзисторов на микросхеме. Это имеет целый ряд негативных последствий – и, прежде всего, сложность. Во-первых, сложность самого изделия как такового, и, во-вторых, растущая сложность и трудоемкость проектирования, вызванная необходимостью совместить два потока информации, – обрабатываемый и управляющий.

Для асинхронных процессоров сложность проектирования резко снижается. В качестве примера можно привести разработанную сотрудниками Манчестерского университета Дагом Эдвардсом и Эндрю Бэрдсли систему Balsa («an asynchronous hardware synthesis system»), предназначенную для описания и синтеза схем путем прямой компиляции, что невозможно в случае синхронных машин.

Откуда же происходит сохраняющаяся приверженность синхронности, если она так плоха? Синхронность в современных процессорах есть прямое следствие использования исключительно двоичной булевой логики. Врожденный порок, которым страдает двоичная логика, состоит в том, что она не включает в себя условие завершения выполнения функции; она может существовать только на фоне соответствующей временной диаграммы, развертывающей процесс выполнения во времени. Точнее, комбинационным устройствам это не очень-то и нужно, а вот описать секвенциальную логику памяти, не используя временные диаграммы, и спроектировать работу машины, не синхронизируя эти диаграммы различных устройств, невозможно. Иными словами, булева логика не представляет собой символически законченную логическую систему (symbolically completed), поскольку ей для завершения недостаточно собственного потока логических переменных.

Булеву логику можно сделать законченной с привлечением внешнего управления, на этом и построены все современные процессоры. Для практической реализации к основной логике требуется добавить

управление во времени, осуществляющее синхронизацию. Понятно, что информационные входные сигналы поступают не одновременно, задержка может быть разной, следовательно, на какой-то момент возникает ситуация неопределенности, которую можно разрешить, если, например, регламентировать срабатывание логической схемы по управляющему синхросигналу. Синхронизация реализуется посредством дополнительных сигналов (управляющих импульсов, или «стробов»), поступающих от тактового генератора или системных часов. Иначе говоря, у двоичной схемы «и/или», имеющей два информационных входа, на самом деле обязан быть третий управляющий вход, который синхронизирует два остальных.

Троичная логика этой проблемы лишена, но у нее полно собственных. Как бы не были интересны машины на троичных элементах, построенные в 60-70-х годах, они остались без будущего в силу отсутствия легко реализуемой элементной базы. Следовательно, нужно каким-то образом совместить бинарные элементы с логикой более сложной, нежели булева. На это и оказались направлены усилия нескольких инженерных коллективов.

По-видимому, асинхроника, как математическая модель, проистекает из основополагающей работы Д. Хаффмена (1954), который предложил модель асинхронного конечного автомата. Его труд знаменовал начало формальной стадии асинхроники, которая стартовала после скрытой от нас наивной фазы, и вызвал к жизни тысячи работ, посвященных противогоночному кодированию автоматов и расширениям модели. Между тем все известные модели асинхронных автоматов базируются на допущении, что инициатором перехода автомата в следующее состояние является изменение входного символа, а новый входной символ может быть подан лишь после завершения переходных процессов предыдущего такта. В противном случае будут иметь место сбои. Другим существенным ограничением моделей хаффменовского типа является жесткая дисциплина смены входных сигналов, ужесточающая механизм взаимодействия автомата с внешней средой (обычно – соседние переходы). Это ограничение ставило под вопрос возможность композиции асинхронных автоматов и в конце концов стало тормозом на пути широкого практического использования подхода. Неясным оставался также вопрос, как избежать последствий нестабильности элементов, реализующих автоматы (равным образом не решенный и для синхронных реализаций).

Знаменательным для формальной фазы асинхронной науки надо признать 1959 г., когда Д. Е. Маллер и У. С. Бартки опубликовали отчет, в котором впервые предложили подход, связанный со схемами, не зависящими в своем поведении от задержек элементов. Сейчас такие схемы чаще называют асинхронными или самосинхронными (speed-independent, quasi delay insensitive, or self-timed). Д. Е. Маллер во втором томе своей книги пытался привлечь внимание к маллеровскому подходу, но особого успеха не добился. Чрезмерное увлечение формалистикой в этом подходе оттолкнуло практиков, не увидевших изящных схемных решений. Неудачей окончилась и попытка Маллера воплотить свои идеи в рамках проекта Illiac II, по-видимому, из-за недостаточно высокого уровня технологической базы того времени и слабой проработки схемотехники базовых узлов.

В 1990-е исследования в этом направлении вели несколько групп – Sun Microsystems разрабатывала Asynchronous Design Technology, а Theseus Logic – NULL Convention Logic. Помимо них, активные работы по созданию асинхронных процессоров возобновил в 1990 г. Манчестерский университет силами AMULET Group; было и еще несколько исследовательских групп в университетах Соединенных Штатов и Западной Европы. С математической точки зрения, пожалуй, наиболее изящной является разработка альтернативной логики NULL Convention Logic (NCL), выполненная специалистами из Theseus Logic.

Логику, предложенную Кардом Фантом, основателем Theseus, можно назвать «расширенной двоичной логикой», поскольку, помимо двух основных сигналов TRUE и FALSE, она оперирует еще двумя, NULL и INTERMEDIATE. Логика NCL существует в двух вариантах, как троичная, где используются TRUE и FALSE + NULL, и как четверичная, где используются все четыре значения.

Избавление от обязательных часов и необходимости в синхронизации в NULL Convention Logic достигается путем интеграции данных и управления в одном логическом выражении, в одном логическом потоке, где NULL, ее называют переменной-разделителем, выполняет синхронизирующую функцию. Информативные сигналы, нули и единицы, NULL передаются по разным проводам, поэтому иногда такой подход образно называют «dual rail», подразумевая под этим, что для передачи сигнала используется «два рельса» (точнее, два провода). Важно заметить разницу в смысловом значении сигнала: проводник сам по себе не несет в себе логического значения, изменение напряжения на нем свидетельствует только о том, что сигнал поступил. Сигнал приобретает логический смысл лишь тогда, когда по параллельному проводнику поступает NULL; такое разделение обеспечивает самосинхронизацию. Логику NCL можно представить как второй логический уровень над булевой логикой, который позволяет воспроизвести троичные элементы с использованием принятых сегодня технологий.

В качестве элементной базы, на которой реализуется NCL, применяются пороговые вентили с гистерезисом. В отличие от простого полупроводникового вентиля, этот имеет два состояния – DATA или NULL. Исходное состояние – всегда NULL, вентиль сохраняет NULL. Переход в состояние DATA возможен, когда на всех входах будет значение DATA, каждое подтвержденное своим разделителем NULL. Использование четвертого сигнала INTERMEDIATE и «четырёхрельсовой» логики позволяет сделать конструкцию вентиля и схемы в целом более эффективной. Очевидно, расплачиваться за избавление от синхронизации приходится увеличенным числом проводников и транзисторов.

В качестве одной из наиболее реальных областей применения асинхронных микросхем на основе NCL называют системы на кристалле (system-on-chip – SoC). В 2000 г. Theseus выпустила 8-разрядный микроконтроллер NCL08, изготавливаемый по 0,25-микронной технологии и насчитывающий 1 млн. транзисторов. В 1999 г. Motorola перепроектировала свой микроконтроллер STAR08 на основании предложений от Theseus.

Вторая работа по асинхронным процессорам велась в исследовательской лаборатории Sun Microsystems под руководством А. Сазерленда. За свои работы по приданию процессорам асинхронности он еще в 1989 г. получил премию имени Тьюринга. А. Сазерленду принадлежит идея микроконвейера, функционирование которого организовано на принципах, названных «рукопожатием» (handshaking). Рукопожатие в данном случае – реализация обратной связи путем обмена служебными сигналами. Этот принцип получил большое распространение, он существует в разных версиях, активное и пассивное, двух- и четырехфазное. Однако сам А. Сазерленд развил идею асинхронности, но уже на уровне полупроводниковых переходов в проекте FleetZero.

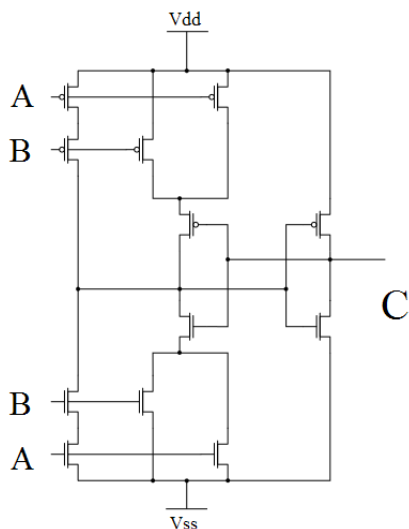
В Калифорнийском технологическом институте в Пасадене исследователи, взяв в качестве прообраза процессор MIPS R3000, сделали его асинхронный аналог MiniMIPS. Аналогичное направление работ избрала для себя и английская группа AMULET Group, выпустившая в 1994 г. процессор AMULET. Эти процессоры родственны синхронному процессору StrongARM. Кроме того, в этом исследовательском центре ведутся работы по использованию нейронных сетей в асинхронных процессорах.

Самотактируемый процессор (self-timed processor, clockless processor), как иногда называют асинхронную машину, нельзя назвать полностью лишенным часов. Он имеет один или несколько внутренних тактовых генераторов и потому не требует центрального генератора тактовой частоты и схем, поддерживающих распространение общего внешнего тактового сигнала. Такие процессоры излучают гораздо меньше радиоволн и расходуют меньше энергии по сравнению с синхронными процессорами. При проектировании внутренних блоков асинхронных процессоров не рассматриваются вопросы их синхронизации с другими блоками. Для связи между блоками используется единый протокол: запрос – ответ с квитированием. Когда вычисления не производятся, блоки не потребляют почти ничего, находясь в спящем режиме, но, тем не менее, мгновенно реагируют на запросы. Мы видим здесь, что процессом вычислений управляют данные: нет данных – нет вычислительной активности. Понятно, что такие блоки легче интегрировать в SOC, и в итоге их общая производительность выше, чем у синхронных процессоров. Повышение производительности достигается за счет того, что блоку нет необходимости ждать прихода внешнего тактового сигнала, чтобы выполнить свою работу.

Асинхронная система получается более надежной, поскольку ее подсистемы работают в более широких диапазонах изменений напряжения питания и температуры. Важно, чтобы самотактируемым был не только процессор, но и остальные микросхемы: чипсет, ОЗУ, контроллеры и т. д. Весь набор таких микросхем и именуется самотактируемой логикой (clockless logic). Американский термин clockless logic (дословно – нетактируемая логика) может ввести в заблуждение, поэтому подчеркнем, что для работы асинхронных

микросхем тактовые сигналы, как правило, нужны, но их источник локален.

Помимо уже указанных преимуществ самотактируемых процессоров перед традиционными есть и ряд других. В архитектуре, построенной на самотактируемых микросхемах, процессор становится достаточно простым и элегантным. Многие компании, в том числе Intel, IBM и Motorola, проводили исследования в области асинхронных вычислений. К этому подталкивали трудности, возникающие с распространением общего тактового сигнала в ультрабольших интегральных схемах. Первый самотактируемый процессор разработал профессор Элэйн Мартин (Alain Martin) в Калифорнийском технологическом институте (Калтех), а сама идея принадлежит одному из создателей компьютерной графики А. Сазерленду (Ivan Sutherland), написавшему первую статью о нетактируемой логике. В 1997 г. корпорация Intel создала совместимый с процессором Pentium асинхронный тестовый кристалл, который был в три раза производительнее и потреблял вдвое меньше энергии. В 1998 г. Philips выпустила асинхронный процессор для своих пейджеров. В 2001 г. Intel в Pentium 4 частично реализовала элементы асинхронной логики.



Один из элементов самотактируемого процессора - Muller C-Gate

Что касается современности, то практически все чипы сейчас являются в той или иной мере асинхронными. Эпоха, когда на материнской плате стоял один генератор и множители частоты для системной шины, памяти и процессора, осталась в начале 1990-х. Другое дело, что создание полностью асинхронных машин по типу FleetZero не оправдало себя – слишком большие проблемы вызывает беспорядочная работа всех устройств компьютера, тем более что любая программа имеет задержки, ветвления, команды занимают разное время и добиться более-менее равномерного исполнения всего этого без тактующего сигнала невозможно. В

то же время, если этих сигналов несколько, каждый для своих задач – получается вполне жизнеспособная конструкция.

Стоит отметить, что первый советский процессор (вернее, микропроцессорная секция) К587 работал без тактового генератора – он отсутствовал уже на этапе идеи. В Советском Союзе этим направлением занимался В.И. Варшавский, признанный специалист асинхронной логики.

1.3.5. Формальные системы

Математическая логика ценна не сама по себе, а тем, что позволяет строить строгие формальные системы и заниматься их исследованием. Несколько терминов нам понадобится далее, поэтому будет резонно ввести их сейчас, тем более, что на формальные системы опирается не только математика, но и языки программирования, и теория алгоритмов, и т.д.

Любая формальная (т.е. бессодержательная, не имеющая интерпретации) теория в математике считается заданной, если определены:

1) алфавит. Алфавитом Σ назовем конечное непустое множество. Его элементы назовем буквами и введем операцию конкатенации – приписывания (для определенности справа) одной буквы к другой. Определим рекурсивно слово над алфавитом Σ так:

- всякая буква является словом над Σ ,
- всякая конечная конкатенация слов является словом над Σ .

Слово, не содержащее символов (а пробел, например, тоже символ!), обозначается ε и называется пустым словом. Множество слов над Σ обозначается Σ^* , множество непустых слов Σ^+ ;

2) множество $L \subseteq \Sigma^*$ называется формальным языком над алфавитом Σ . Чтобы как-то выделять слова, входящие в язык, необходим инструмент. Одним из таких инструментов являются т.н. формальные грамматики, предложенные математиком и анархистом Ноамом Хомским в середине XX века. Формальная грамматика (порождающая грамматика) включает набор правил вывода, позволяющих построить все слова заданного языка. Распознающая грамматика позволяет, наоборот, по данному слову за конечное число шагов определить, принадлежит оно языку или нет. Если она существует – язык распознаваем;

3) если с помощью придуманного языка мы хотим выразить какие-то более сложные вещи, то нам нужно научиться соединять слова в непротиворечивые выражения. Для этого выделяем произвольное множество слов и нарекаем их аксиомами. Если множество аксиом бесконечно, обычно существует процедура, позволяющая генерировать все семейство аксиом из некоторого базового набора;

4) после этого мы придумываем непротиворечивые правила, по которым данные слова соединяются (или, как говорят, формулы вступают в отношения), называемые правилами вывода. Все, формальная теория готова;

5) если всегда есть эффективная процедура, позволяющая выяснить, является ли произвольное слово языка аксиомой, то теория признается аксиоматизированной или аксиоматической. Аксиомы обычно делятся на логические (общие для всех формальных теорий) и собственные (зависящие от интерпретации теории);

6) наша порождающая грамматика пригодится теперь лишь для создания аксиом. Более сложные слова-формулы мы будем генерировать с помощью правил вывода прямо из аксиом. Значит, для каждой произвольной формулы всегда можно установить, является ли она следствием применения правил вывода к аксиомам или нет. Если да – то процесс ее получения называется вывод, а сама выведенная формула – теоремой;

7) заметим, что для хорошей, удачно придуманной системы правил и аксиом все формулы теории – либо теоремы, либо аксиомы, но, как показала практика, все более-менее интересные теории (для которых есть содержательные нетривиальные модели) таким счастьем не обладают. Теорема Геделя о неполноте гласит, что ни одна формальная теория, сложнее чем арифметика, не может быть эффективно разрешена, то есть, не существует никакого набора правил вывода, который позволяет для любой формулы установить – выводима она или нет;

8) формальные теории различаются по сложности – от самых простых, типа исчисления высказываний, до языков высших порядков. Различие заключается в богатстве алфавита и правил вывода и допустимости применять правила к чему-либо.

Рассмотрим ряд формальных теорий по возрастанию сложности.

Классическая логика нулевого порядка (пропозициональная логика, исчисление высказываний) – самая простая формальная теория.

Алфавит: буквами служат элементарные высказывания (пропозициональные переменные типа «Идет снег»), обозначаются они, как ни странно, буквами, обычно A, B, C. Также в алфавит входят логические связки и скобки. К порождающей грамматике относятся правила употребления букв и составления из них сложных высказываний:

1. Если P — пропозициональная переменная, то P — формула.
2. Если P — формула, то $\bar{P}$ — формула.

3. Если A и B — формулы, то $A \vee B$, $A \wedge B$, $A \rightarrow B$ — формулы.

4. Других формул нет.

Непротиворечивых систем аксиом можно придумать много, наиболее употребительна система Гильберта:

1. $P_1 \rightarrow (P_2 \rightarrow P_1)$
2. $((P_1 \rightarrow (P_2 \rightarrow P_3)) \rightarrow ((P_1 \rightarrow P_2) \rightarrow (P_1 \rightarrow P_3)))$
3. $P_1 \wedge P_2 \rightarrow P_1$
4. $P_1 \vee P_2 \rightarrow P_2$
5. $P_1 \rightarrow (P_1 \vee P_2)$
6. $P_2 \rightarrow (P_1 \wedge P_2)$
7. $P_1 \rightarrow (P_2 \rightarrow (P_1 \wedge P_2))$
8. $(P_1 \rightarrow P_3) \rightarrow ((P_2 \rightarrow P_3) \rightarrow ((P_1 \vee P_2) \rightarrow P_3))$
9. $\overline{P_1} \rightarrow (P_1 \rightarrow P_2)$
10. $(P_1 \rightarrow P_2) \rightarrow ((P_1 \rightarrow \overline{P_2}) \rightarrow \overline{P_1})$
11. $P_1 \vee \overline{P_1}$

В качестве единственного правила обычно используется Modus ponens –

$$Mp: \frac{P_1 \rightarrow P_2, P_1}{P_2},$$

однако никто не воспрещает обогатить нашу теорию правилом подстановки –

$$S: \frac{A}{A(Q_1, \dots, Q_n | B_1, \dots, B_n)}.$$

Выводимые формулы, т.е. теоремы, стандартно обозначаются как $\models$.

Пример вывода.

Выведем очевидное $A \rightarrow A$.

1. Из а.2, применив S : $(P_1, P_2, P_3 | A, (A \rightarrow A), A)$, имеем:

$$((A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow ((A \rightarrow A) \rightarrow (A \rightarrow A))))$$

2. Из а.1, применив S : $(P_1, P_2 | A, (A \rightarrow A), A)$, имеем:

$$A \rightarrow ((A \rightarrow A) \rightarrow A))$$

3. Из а.2 и а.3, применив Mp имеем:

$$(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)$$

4. Из а.1, применив S : $(P_1, P_2 | A, A)$, имеем:

$$A \rightarrow (A \rightarrow A)$$

5. Из а.3 и а.4, применив Mp , имеем:

$$A \rightarrow A$$

Таким образом, видно, что в исчислении высказываний легче убиться, чем вывести сколько-нибудь сложные теоремы. К сожалению, эта особенность характерна для любых формальных систем – на начальном этапе, когда еще вообще ничего не доказано, приходится возводить здание математики из очень маленьких кирпичей, что подобно написанию ОС на ассемблере.

Наша получившаяся теория достаточно слаба, т.е. ей не соответствует никакая сильно сложная модель, зато она аксиоматизируема, полна и непротиворечива. Для придания какой-либо содержательности можно ввести т.н. истинностные константы, обычно 0 и 1, и работать лишь с теми формулами, которые являются тавтологиями, т.е. истинны вне зависимости от истинности входящих в формулу переменных. Полезной моделью может являться, например, теория логических цепей в компьютерах.

Логика первого порядка (исчисление предикатов) – вторая по сложности теория. Алфавит расширяется следующим образом. Теперь он состоит из множества F функциональных символов (они же индивидуальные или предметные переменные, обозначаются $x, y, z, \dots$) и множества P предикатных символов (они же предикатные буквы, обозначаются P_i^j).

С каждым функциональным и предикатным символом связано число возможных аргументов символа, начиная от нуля. Число возможных аргументов в P обозначается как раз верхней i , если $i = 0$, то P – обычный пропозициональный символ, который у нас уже был. Верхний индекс почти всегда опускают для простоты записи. Кроме того, в алфавите используются следующие дополнительные символы:

Пропозициональные связи: $\vee, \wedge, \rightarrow, \overline{}$.

Кванторы всеобщности и существования: $\forall, \exists$.

Служебные символы: скобки, запятая, пробел.

Грамматика выглядит так:

1. Если P — пропозициональный символ, то P — формула и $P^n(x_1, \dots, x_n)$ — формула.

2. Если P — формула, то $\bar{P}$ — формула.

3. Если A и B — формулы, то $A \vee B, A \wedge B, A \rightarrow B$ — формулы.

4. Если A — формула и x — предметная переменная, то $\forall x A$ и $\exists x A$ — формулы.

5. Других формул нет.

Вхождение переменной x в формулу A может быть свободным или связанным:

1. В формулу $P(x_1, \dots, x_n)$ переменные $x_1, \dots, x_n$ входят свободно.

2. В формулы $A \vee B, A \wedge B, A \rightarrow B$, где $A = A(x_1, \dots, x_n)$ и $B = B(x_1, \dots, x_n)$, а также $\forall y A$ и $\exists y A$ (если $\forall i x_i \neq y$) переменные $x_1, \dots, x_n$ входят свободно.

3. В формулы $\forall x A$ и $\exists x A$ переменные $x_1, \dots, x_n$ входят связанным образом.

4. Единожды связанные в п.3 переменные остаются таковыми и в формулах $A \vee B, A \wedge B, A \rightarrow B$.

Разницу между переменными легко уяснить, если осознать, что вместо связанной переменной нельзя подставить конкретное значение, но можно заменить ее любой другой буквой, без ущерба смыслу. Вот пример не из логики.

В выражении $\int_{-1}^y xy dx$ переменная y свободна, а x связана кванторной приставкой dx . Это означает, что подставив везде вместо x , скажем, z , мы не изменим смысла выражения, а подставив, скажем, 6 — совершим синтаксическую ошибку. При этом вместо y мы можем подставлять что угодно, но, подставив z , мы не изменим смысла выражения, а подставив 6 — изменим.

Для придания большего пафоса обычно употребляют также конструкции «терм» и «атом», понимая под ними уже упомянутые нами предметные переменные и предикатные символы. При этом термом является и любая логическая комбинация переменных, использующая стандартные связки, а атомом — предикатные символы. Тогда формула из одного символа станет называться атомарной формулой, а из многих со связками — просто формулой. Кроме интересных слов, определив термы и атомы строго формально, мы получаем выгоду в виде возможности строить рекурсивные функции, индуктивно доказывать всякие интересные нам свойства и прочие хорошие вещи.

Выражением в нашем языке станет комбинация термов и атомов, например:

$(x \rightarrow y)$ — терм;

$A \rightarrow (B \rightarrow A)$ — формула из нескольких атомарных;

$\forall x A(x_1, \dots, x_n) \exists y B(y_1, \dots, y_n), y_i = \bar{x}_i$ — выражение.

Система логических аксиом логики первого порядка состоит из аксиом исчисления высказываний, дополненная двумя новыми аксиомами:

1. $\forall x A \rightarrow A[t/x]$;

2. $A[t/x] \rightarrow \exists x A$,

где $A[t/x]$ — формула, полученная в результате подстановки терма t вместо каждой свободной переменной x , встречающейся в формуле A .

К правилам вывода добавляется обобщение:

$$\frac{A}{\forall x A}$$

Формальную подстановку можно обозначить как $\theta(x)$, она превращает терм x в новый терм $t = \theta(x)$. Также как и обычную функцию, ее легко можно задать таблично:

$$\begin{array}{ccc} x_1 & \dots & x_n \\ t_1 & \dots & t_n \end{array}$$

Пусть T — выражение нашего языка, тогда $\theta(T)$ превращает каждую свободную переменную $x \in T$ в $t = \theta(x)$. При этом, если некоторые x_i не свободны, их трогать не надо! Вместо $\theta(T)$ обычно употребляют явную запись:

$$T_{t_1, \dots, t_n}^{x_1, \dots, x_n}$$

Вот правильный индуктивный рецепт проведения подстановок, который может пригодиться, в т.ч. в лямбда-исчислении (о нем ниже):

1. $\theta(A(t_1, \dots, t_n)) = A(\theta(t_1), \dots, \theta(t_n))$

2. $\theta(A \Delta B) = \theta(A) \Delta \theta(B)$

3. $\theta(\bar{A}) = \overline{\theta(A)}$

4. $Qx \theta(A) = Qx (A(\theta(x) \setminus \{x\}))$, здесь Q — квантор

Однако, несмотря на соблюдение этих правил, не все корректные подстановки пригодны с точки зрения логики. Рассмотрим, например, такую ситуацию.

Пусть атомарная формула $P(x, y, z)$ выражает высказывание « $x + y = z$ », а формула $\exists y P(x, y, z)$ выражает « $x \leq y$ ». Пусть в этом же языке терм $f(x, y)$ определяет операцию умножения « $x \cdot y$ ». Теперь мы хотим подставить $f(x, y)$ в $\exists y P(x, y, z)$ вместо свободной переменной y , чтобы вывести « $x \cdot y \leq z$ ». Однако, совершив подстановку

$$\exists y P(x, y, z) \stackrel{x}{\mid} f(x, y)$$

мы увидим, что получили совсем не то, что хотели – это ясно, хотя бы из того, что результат высказывается о двух переменных, а не о трех, как должен был. Эта случилось из-за того, что возникла т.н. коллизия – переменные, которые до этого были свободны, связаны и выражают теперь совсем не то, что нужно. Правильным подходом здесь будет сначала переименовать связанную переменную y , например в u , а затем произвести подстановку вида $\exists u P(f(x, u), u, z)$.

Как же избежать коллизий? Существуют подстановки, которые никогда к ним не приводят. Говорят, что подстановка θ допустима для T , если:

1. $\forall x \in \text{dom} \theta$ любое свободное вхождение x в T не попадает в область действия кванторов по переменным, свободно входящим в $\theta(x)$.
2. $\forall x \in \text{dom} \theta$ выражение T не содержит кванторов по параметрам терма $\theta(x)$.

Наконец, если и этого недостаточно, то в дело вступает **логика второго порядка**, расширяющая логику первого порядка возможностью квантификации общности и существования не только над переменными, но и над предикатами. Логика второго порядка несводима к логике первого порядка. В свою очередь, она расширяется логикой высших порядков и теорией типов.

Формула языка математической логики тогда становится истинной или ложной, когда она ассоциируется с некоторым множеством, называемым *носителем интерпретации*. Этот носитель служит прямым или косвенным источником для назначения значений встречающихся в формуле имён и переменных — прямым, если в качестве значений берутся элементы носителя; косвенным, если в качестве значений берутся функции или отношения, определённые на носителе. В языке первого порядка в качестве значений переменных разрешается брать только элементы носителя, в языке второго порядка в качестве значений переменных разрешается брать также функции и отношения.

Выразительные возможности языков первого порядка довольно ограничены. Например, на языке первого порядка можно сообщить, что носитель содержит ровно 17 элементов, но невозможно выразить его конечность. На языке второго порядка выразить конечность носителя возможно. Возникает совершенно естественное недоумение: а зачем тогда пользоваться языками первого порядка с их бедными выразительными средствами, не лучше ли пользоваться языками второго порядка?

Однако выразительные средства языков второго порядка оказываются слишком богатыми. В частности, на языке второго порядка можно написать формулу, выражающую существование множества с мощностью, промежуточной между мощностями счётной и континуальной. В свете известных результатов Гёделя (1940) и Кёэна (1963) сама постановка вопроса о том, является эта формула истинной или ложной, теряет ясный смысл.

1.4. Машинная арифметика в кольце вычетов

1.4.1. Введение в алгебру

С умножением, сложением и вообще арифметическими операциями в компьютере все обстоит просто лишь на первый взгляд. Дело в том, что при операциях вне компьютера мы имеем полное право никак не ограничивать их результат. Даже если мы получим число, слишком большое или маленькое, чтобы записать его на бумаге, нам на помощь всегда придут гипероператоры, стрелочная нотация Кнута и т.п. вещи, предназначенные для выражения произвольно больших и малых чисел. В машине же все совсем не так – даже при сложении слишком больших или вычитании слишком маленьких чисел (не говоря уже об умножении или делении) мы постоянно рискуем выйти за допустимый в разрядной сетке диапазон.

Например, сложив две восьмибитные целочисленные переменные без знака $(260 + 61)_{10} += (11010010 + 00111101)_2$, мы получим не 271 (100001111), как ожидалось, а 15 (00001111), т.к. первый лишний бит будет отброшен. Если в процессе исполнения программы не проверять, не произошло ли переполнение, то возникают серьезные и трудноуловимые ошибки. Ситуация намного ухудшается, если речь идет о вещественных переменных. Для них дополнительно характерна ошибка, называемая исчезновением порядка или антипереполнением (floating-point underflow) — ситуация, когда результат операции с плавающей запятой становится настолько близким к нулю, что порядок числа выходит за пределы разрядной сетки. Например, в арифметике одинарной точности (с минимальным числом $1.4 \cdot 10^{-38}$) к исчезновению порядка приведёт операция $10^{20} \cdot 10^{20} = 10^{40}$. Результатом такой операции можно сделать 0, но это чревато дальнейшими ошибками, например, делением на ноль.

В стандарте IEEE 754¹⁷, описывающем арифметику с плавающей запятой, такие числа называются денормализованными (denormalized numbers, subnormal numbers). При записи в форматах float, double, long double их экспонента будет записана как 0, так что для получения их значения не требуется использование неявной единицы; мантисса просто умножается на наименьшую для данного формата экспоненту. Денормализованные числа находятся ближе к 0, чем наименьшее представимое нормализованное число, занимая промежуточное между ними место.

Математические сопроцессоры обычно работают с денормализованными числами аппаратным способом с той же скоростью, что и с нормализованными. Однако в некоторых процессорах такая аппаратная обработка не реализована (например, чтобы избежать усложнения реализации FPU), и малые значения либо приводятся сразу к нулю, либо обрабатываются в операционной системе программным способом. Вторым вариантом приводит к ощутимому увеличению времени обработки денормализованных чисел.

Рассмотрим машинную арифметику с самого простого варианта: что делать, когда ошибка переполнения происходит с целочисленными переменными без знака? Наиболее логичным вариантом будет округление посредством отброса лишних битов, не помещающихся в разрядную сетку. В результате мы приходим к арифметике, в которой $\max + 1 = \min$. Такая арифметика называется арифметикой в кольце вычетов или просто циклической.

Для того чтобы понять ход дальнейших рассуждений, необходимо привести элементарные сведения из алгебры. Простейшим математическим объектом, как известно, является множество. Для простоты можно считать, что это произвольная совокупность различных элементов, не имеющая ни структуры, ни операций, ни каких-либо особенностей. Обозначается оно обычно так: $A = \{a_1, a_2, \dots\}$. Для 2-х и более множеств можно рассмотреть операцию, называемую декартовым произведением.

Декартовым произведением множеств A и B , называют множество

$$A \times B = \{(a_1, b_1), (a_1, b_2), \dots, (a_2, b_1), (a_2, b_2), \dots, (a_i, b_j), \dots\},$$

состоящее из всех пар элементов этих множеств. Отображением множеств $A \rightarrow B$ называют сопоставление элементов a_i и b_i , не обязательно однозначное. Отображение $A \times A \rightarrow A$, которое каждой паре элементов из A ставит в соответствие элемент также из A , называют операцией на A .

Например, сложение в $\mathbb{N}$. Это операция, ставящая в соответствие паре чисел третье, скажем так: $(4, 5) \rightarrow 9$, то же самое можно записать в привычном для нас виде, как $4 + 5 = 9$. Если наши операции, например, на том же $\mathbb{N}$ определены не просто так, а чтобы удовлетворять особым правилам (например, $4 + 5 = 5 + 4$), то говорят, что на множестве задана алгебраическая структура. Одна из самых простых таких структур носит название группа. Это множество G с заданной на нем операцией, называемой «умножением», удовлетворяющей следующим правилам:

- в G существует особый элемент, нейтральный относительно умножения – при умножении на него любого другого элемента, этот другой не изменяется: $\exists e \in G, \forall a \in G a \cdot e = a$. Такой элемент называется единица группы и обозначается 1 или e ;

- для любого элемента группы существует элемент, называемый обратным – умножение на него дает 1: $\forall a \in G \exists a^{-1} \in G, a \cdot a^{-1} = e$;

- наконец, операция умножения ассоциативна: $\forall a, b, c \in G a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

Конечно, вместо умножения можно выбрать сложение, или более экзотические операции. Заметим, что коммутативности от нас не требуется – вполне может существовать группа, в которой $(a, b) \neq (b, a)$, где скобками обозначена произвольная операция. Например, умножение матриц, как известно, не коммутативно, следовательно, группа матриц по умножению – не коммутативная группа.

Кольцом называется структура посложнее. Она уже содержит сразу обе операции – умножение и сложение (отметим, что как и в случае группы, это, вообще говоря, не наши обычные операции над числами – под ними может подразумеваться все, что угодно – умножение матриц или многочленов, сложение векторов и т.п.). При этом по одной операции кольцо – коммутативная группа (обычно по сложению), по второй (умножению) – моноид – группа без требования обратимости. Между собой сложение и умножение связаны дистрибутивным законом: $a \cdot (b + c) = a \cdot b + a \cdot c$.

Среди отношений, которые могут связывать несколько элементов множества, особо выделяется т.н. отношение эквивалентности. Различие элементов, связанных таким отношением, если оно есть, признается в контексте данного отношения несущественным, т.е. элементы считаются эквивалентными.

Такое отношение, обозначаемое $\sim$, должно удовлетворять трем свойствам:

- рефлексивность: $\forall a, a \sim a$;
- симметричность: $a \sim b \Leftrightarrow b \sim a$;
- транзитивность: если $a \sim b$ и $b \sim c$, то $a \sim c$.

¹⁷ Очень рекомендуется к изучению, например, здесь: <http://www.softelectro.ru/ieee754.html> и <http://docs.oracle.com/cd/E19422-01/819-3693/>

Простейший пример такого отношения – обычное равенство чисел, но бывают и более сложные, например, коллинеарность векторов и т.п. Если на множестве задано такое отношение, то оно разбивается на подмножества одинаковых элементов. Например, множество векторов $\{\vec{a}_1, \vec{a}_2, \dots\}$ можно разбить на подмножества векторов, коллинеарных ортам $\{\vec{e}_1, \vec{e}_2, \vec{e}_3\}$, конечно, в них войдут не все вектора. Такие подмножества эквивалентных объектов, с точки зрения очень многих операций, можно заменить одним единственным произвольным представителем от подмножества – все равно остальные ему эквивалентны. Такая совокупность представителей от классов эквивалентности называется фактормножеством данного множества. Если наше исходное множество имело какую-либо структуру, например, было группой или кольцом, то фактормножество наследует эту же структуру и называется, соответственно, факторгруппа или факторкольцо по данному отношению эквивалентности.

Как же все это связано с машинной арифметикой?

В теории чисел сравнимыми по модулю n называют такие числа a и b (обозначается, обычно, $a \equiv b \pmod{n}$ или $a \equiv_n b$), что остатки от деления a/n и b/n совпадают. Иначе говоря, если ¹⁸ $a = q_1 \cdot n + r_1$ и если $b = q_2 \cdot n + r_2$, где $0 \leq r_i < n$, то, в случае сравнимых чисел, $r_1 = r_2$. Из этого легко заметить, что эквивалентным определением сравнимости будет деление разности a и b на c без остатка. Например, $16 \equiv 30 \pmod{7}$, т.к. $30 - 16 = 2 \cdot 7 + 0$.

Несложно видеть, что сравнение имеет ряд очевидных свойств:

- рефлексивность: $\forall a \in \mathbb{N} a \equiv a \pmod{n}$;
- симметричность: $a \equiv b \pmod{n} \Leftrightarrow b \equiv a \pmod{n}$;
- транзитивность: если $a \equiv b \pmod{n}$ и $b \equiv c \pmod{n}$, то $a \equiv c \pmod{n}$.

В результате, сравнение по модулю натурального числа n есть не что иное, как отношение эквивалентности в кольце целых чисел, связанное с делимостью на n . Факторкольцо же по этому отношению называется кольцом вычетов. Совокупность соответствующих тождеств и алгоритмов в этом факторкольце образует модулярную (циклическую) арифметику, которую использует ЭВМ.

Это связано с тем, что сравнения обладают многими свойствами обычных чисел – их можно складывать, перемножать и т.п. – например, если числа a_1 и a_2 попарно сравнимы по модулю n с b_1 и b_2 , то сравнимы и $a_1 + a_2$ и $b_1 + b_2$, то же справедливо и для умножения.

Сравнения, однако, нельзя делить друг на друга или на другие числа. Можно делить обе части сравнения на число, взаимно простое с модулем, и можно одновременно разделить обе части сравнения и модуль на их общий делитель, нельзя также выполнять операции со сравнениями, если их модули не совпадают.

Множество всех чисел, сравнимых с a по $\pmod{n}$ называют классом вычетов по $\pmod{n}$ и обозначают $[a]_n$ или $\overline{a}_n$. Фактормножество классов вычетов над $\mathbb{Z}$ обозначают $\mathbb{Z}_n$ или $\mathbb{Z}/n\mathbb{Z}$. Как уже упоминалось, $\mathbb{Z}_n$ является кольцом относительно операций:

- $[a + b]_n = [a]_n + [b]_n$
- $[a \cdot b]_n = [a]_n \cdot [b]_n$

Одним из важнейших утверждений теории вычетов является знаменитая китайская теорема об остатках. Она формулируется так:

Если натуральные числа $a_1, a_2, \dots, a_n$ попарно взаимно просты, то для любых целых $r_1, r_2, \dots, r_n$ таких, что $0 \leq r_i < a_i$ для любых $i \in 1 \dots n$, найдется число N , которое при делении на a_i дает остаток r_i для любых $i \in 1 \dots n$. Более того, если найдутся два таких числа N_1 и N_2 , то $N_1 \equiv N_2 \pmod{a_1 \cdot a_2 \cdot \dots \cdot a_n}$.

Китайская теорема об остатках широко применяется в теории чисел, криптографии и машинной арифметике. Например, взаимно однозначное соответствие между некоторым числом и набором его остатков, определяемым набором взаимно простых чисел, существование которого утверждается в теореме, на практике помогает работать не с длинными числами, а с наборами их коротких по длине остатков. Кроме того, вычисления по каждому из модулей можно выполнять параллельно.

Если в качестве базиса взять, к примеру, первые 500 простых чисел, длина каждого из которых не превосходит 12 бит, то этого хватит для представления десятичных чисел длины 1519 знаков¹⁹. Например, в алгоритме RSA вычисления производятся по модулю очень большого числа n , представимого в виде произведения двух больших простых чисел. Теорема позволяет перейти к вычислениям по модулю этих простых делителей, которые по величине имеют порядок $\sqrt{n}$, а значит, имеют в два раза меньшую битовую длину.

¹⁸ а такое представление всегда возможно и единственно.

¹⁹ Откуда взялось число 1519 понять легко: сумма десятичных логарифмов первых 500 простых чисел равна 1519,746...

1.4.2. Арифметика в кольце вычетов

Ценность системы вычетов заключается в том, что она позволяет осуществлять арифметические операции над конечным набором чисел, не выходя за его пределы. Полная система вычетов по модулю n — любой набор из n попарно несравнимых по модулю n целых чисел. Обычно в качестве полной системы вычетов берутся наименьшие неотрицательные вычеты

$$0, 1, \dots, n-1, \dots$$

или абсолютно наименьшие вычеты, состоящие из

$$0, \pm 1, \dots, \pm \frac{n-1}{2}, \dots$$

в случае нечётного n , и чисел

$$0, \pm 1, \dots, \pm \frac{n}{2} - 1, \dots$$

в случае чётного n .

Технически класс вычетов $\mathbb{Z}_n$ состоит из целых чисел $\{0, \dots, n-1\}$ и является, в сущности, тем же, что и обычное множество $\mathbb{Z}$. Все его отличие заключается в специфически определенных операциях сложения и умножения, приводящих к циклической арифметике. Уникальностью классов вычетов, например, является наличие т.н. делителей нуля — чисел, которые при умножении дают ноль, но сами не равны нулю. Например, в $\mathbb{Z}_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ справедливо $2 \cdot 4 = 0$. Кажущаяся нелепость этого пропадает, если вспомнить, что 2 и 4 — не числа, а элементы $[2]_8$ и $[4]_8$, и очевидно, что это не просто умножение, а умножение в кольце вычетов:

$$[2 \cdot 4]_8 \equiv 0 \pmod{8},$$

что очевидно. В результате операция деления в классическом виде в таком кольце определена далеко не для всех элементов. Компьютер использует арифметику кольца вычетов по модулю $2^n - \mathbb{Z}_{2^n}$ (wraparound mode), т.к. отбрасывание разрядов, не помещающихся в слово, является нахождением остатка от деления на 2^n .

В результате, в таком кольце каждое число, кратное 2, является делителем нуля, например, $xxx0 \cdot 1000 = 0000$, где $x \in \{0, 1\}$. Такое построение операций реализует как раз то, что нам необходимо — корректную обработку переполнения, когда $max + 1 = min$.

При машинном сложении и вычитании положение разрядной точки не имеет значения с точки зрения переполнения — точка является абстракцией, предназначенной для нас, а не для компьютера, он обработает по одинаковым правилам и $2.54 + 3.35 = 5.9$ и $254 + 335 = 590$. Перенос же не может быть больше единицы (в любой системе счисления). Также при поразрядном сложении чисел перенос распространяется справа налево до тех пор, пока оба складываемых разряда не станут равны нулю, заем при вычитании распространяется справа налево до тех пор, пока разряд уменьшаемого не станет равен 1, а вычитаемого 0:

$$\begin{array}{r} \begin{array}{c} \text{1} \leftarrow \text{1} \\ + \quad \begin{array}{|c|} \hline 0 \ 1 \ 0 \ 1 \ 1 \ 1 \\ \hline 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\ \hline 1 \ 1 \ 0 \ 0 \ 0 \ 1 \end{array} \\ \hline 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \end{array} \quad \begin{array}{c} \text{1} \leftarrow \text{1} \\ - \quad \begin{array}{|c|} \hline 1 \ 1 \ 0 \ 0 \ 0 \ 1 \\ \hline 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\ \hline 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \end{array} \\ \hline 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \end{array} \end{array}$$

Умножение легко определить через последовательность сложений, а деления в кольце, как уже говорилось, никто не обещал, тем более в случае $\mathbb{Z}_{2^n}$, так что в машинной арифметике деление a на b понимается в смысле Евклида, т.е. нахождение такого частного и остатка, что $a = b \cdot q + r$.

В результате, традиционное деление приближается к частным от деления по Евклиду: $a/b \approx q$, в этом кроется источник погрешности даже целочисленных машинных вычислений, но в силу того, что $r < b$, ошибка в любом случае не превышает единицу младшего разряда.

Если мы рассмотрим отрицательное число, то станет ясно, почему в циклической арифметике срывает дополнительный код. Для получения такого кода надо, как известно, инвертировать все разряды числа и прибавить к нему единицу. В результате, после сложения того, что получилось, с исходным числом гарантированно произойдет переполнение и результат обнулится:

$$\begin{array}{r} \begin{array}{c} \text{1} \ 0 \ 1 \ 0 \\ + \quad \begin{array}{|c|} \hline 0 \ 1 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \\ \hline 1 \end{array} \\ \hline \text{X} \ 0 \ 0 \ 0 \ 0 \end{array}$$

Таким образом, отрицательное число для машины совсем не то же самое, что для нас. С точки зрения человека, это число, равное данному, но с минусом впереди, с точки зрения машины — произвольное положительное, при сложении с которым данного образует ноль. Сложение выполняется одинаковым способом, вне зависимости от того, считаем ли мы все числа положительными или половину отрицательными. Если результат операции не помещается в машинном слове, то говорят, что возникло переполнение.

При сложении (вычитании) без знака, переполнение – это перенос из старшего разряда, при сложении (вычитании) со знаком – переполнением является не перенос из старшего разряда, а факт того, что перенос в знаковый разряд и из знакового разряда различны.

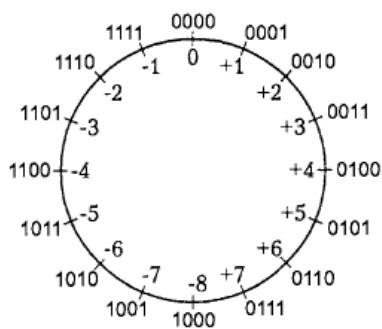
Например, сложение:

Нет переполнения		Переполнение	
$1 + 1 = 2$	$-1 + 1 = 0$	$max + 1 = min$	$-1 + min = max$
0001	1111	0111	1111
0001	0001	0001	1000
00010	10000	01000	10111

Арифметические операции, очевидно, приводят к разному результату, в зависимости от того, какое число мы рассматриваем – беззнаковое или со знаком. Например:

$$13 \cdot 5 = 65 \leftarrow \times \begin{matrix} 1101 \\ 0101 \end{matrix} \Rightarrow -3 \cdot 5 = -15$$

При делении остаток по модулю всегда меньше делителя, а его знак зависит от правил округления частного. Например, в процессорах Intel частное округляется в сторону нуля.



Наглядное представление циклической арифметики

Однако использование такой арифметики не всегда оправдано. После наступления эры мультимедиа, но еще до появления GPU, все операции по преобразованию и выводу изображения производил процессор. Работа с изображением в растровом формате сводится к операциям с массивными матрицами, описывающими цвет и яркость каждого пикселя. Каждый пиксель кодировался одним 32-битным словом. Применение циклической арифметики стало приводить к невообразимым искажениям, например, увеличение яркости больше максимального – $FF + 1$ приведет к экрану черного цвета – 00.

Для борьбы с этим, начиная с набора инструкций MMX (matrix mathematics extension / multi-media extension), был введен новый вид машинной арифметики – арифметика с насыщением (saturating arithmetic). В таком режиме при возникновении переполнения в результате фиксируется максимально возможное значение для данного типа данных, а перенос игнорируется. В случае переполнения снизу в результате фиксируется минимально возможное значение. Граничные значения определяются типом (знаковый или беззнаковый) и разрядностью переменных.

В результате, большинство команд MMX используют суффиксы US (unsigned saturation) – арифметика с насыщением, данные без знака S или SS (signed saturation) – арифметика с насыщением, данные со знаком. Если в суффиксе нет ни S, ни SS, используется циклическая арифметика.

1.4.3. Вычисления в полях Галуа

Познакомимся еще с одной математической структурой – полем. Это коммутативное ассоциативное кольцо с единицей, в котором обратимы все ненулевые элементы. Поле является чрезвычайно хорошей структурой – т.е. в нем действуют все привычные для нас аксиомы операций. Поле, состоящее из конечного числа элементов, называется соответственно конечным или полем Галуа и обозначается $GF(q)$, где q – количество элементов поля. Простейшим примером такого поля является кольцо вычетов по модулю простого числа p .

Зачем нам это понадобится? Машина позволяет точно хранить числа, но, как уже упоминалось, циклическая арифметика не позволяет их точно делить из-за «плохой» структуры кольца вычетов – наличия делителей нуля. Одним из способов преодолеть эту проблему стал переход к другой структуре, устроенной более удачно – полям Галуа.

К счастью, слово длиной n позволяет точно записать 2^n различных элементов, т.е. точно представить какую-либо структуру. Существуют поля размера 2^n , и все они изоморфны, т.е. устроены одинаковым образом – это $GF(2^n)$. $GF(2)$ строится как $\mathbb{Z}_2$, при этом операции в поле совпадают с обычными логическими операциями – сложение и вычитание эквивалентно XOR, умножение и деление – AND.

Рассмотрим n -разрядное машинное слово, как многочлен $GF(2)(x)$ степени $n - 1$:

$$\begin{matrix} a_{n-1} & \dots & a_2 & a_1 & a_0 \end{matrix} \longleftrightarrow a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$$

Далее для простоты будем записывать многочлены в 16-ричном виде в скобках, например:

$$101000111 \sim x^7 + x^5 + x^2 + x + 1 \sim \{A7\}$$

Тогда операция сложения – это поразрядное XOR машинных слов, от нормального сложения оно отличается полным отсутствием переносов между разрядами.

Например:

Как многочлены	Как числа
0 0 1 0 1 1 0 1	0 0 1 0 1 1 0 1
+ 0 1 1 0 0 1 1 1	+ 0 1 1 0 0 1 1 1
0 1 0 0 1 0 1 0	1 0 0 1 0 1 0 0
{2D} + {67} = {4A}	2D + 67 = 94

Операция вычитания происходит так же. Операция умножения эквивалентна операции умножения чисел с выключенными цепями переносов и может быть составлена из комбинации сдвигов и сложений. Для умножения произвольных слов нам понадобится n операций, для умножения на константу их количество равно количеству единиц множителя. Результат потребует для размещения двойного слова. Наконец самое главное – кольцо многочленов является евклидовым, т.е. в нем можно выполнять традиционное деление столбиком. Требуемое количество делений равно разности степеней делимого и делителя.

Как упоминалось выше, умение находить остаток от деления позволяет строить кольца классов вычетов и кольцо, построенное по модулю неразложимого многочлена степени n , будет в точности полем $GF(2^n)$.

Например, по модулю $g(x) = x^8 + x^4 + x^3 + x + 1 \sim \{11B\}$

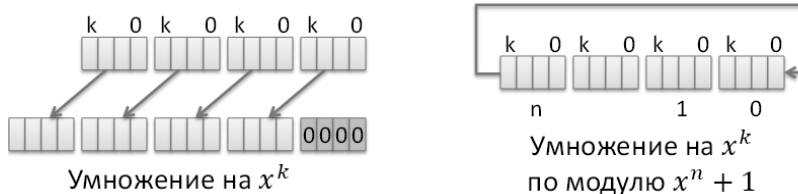
$$\begin{array}{c}
 GF(2) \\
 2 \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}
 \end{array}
 \xRightarrow{\text{присоединение } x}
 \begin{array}{c}
 GF(2)(x) \\
 \infty \begin{Bmatrix} 0 \\ 1 \\ 10 \\ 11 \\ 110 \\ 111 \\ \dots \end{Bmatrix}
 \end{array}
 \xRightarrow{\text{факторизация}}
 \begin{array}{c}
 GF(2)(x)/g(x) \\
 256 \begin{Bmatrix} 00000000 \\ 00000001 \\ 00000010 \\ 00000011 \\ 00000100 \\ 00000101 \\ 00000110 \\ 00000111 \\ \dots \\ 11111111 \end{Bmatrix}
 \end{array}$$

В результате при рассмотрении слова, как элемента $GF(2^n)$, операции сложения и вычитания заменяются XOR, умножение же заменяется либо умножением с отключенными цепями переноса, либо комбинацией сдвигов и сложений, затем нахождением остатка от деления на модуль. В качестве недостатков можно отметить, что не все процессоры умеют отключать цепи переноса, в результате сложность операции растет линейно с размером слова, к тому же результат требует двойного слова для записи.

Для того чтобы решить эти проблемы, произвольное умножение часто заменяют через умножение на фиксированный многочлен по какому-нибудь модулю, которое сводится к сдвигу на разряд влево и сложению с модулем, если возник перенос. Много вариантов быстрого умножения в полях Галуа можно найти в классическом труде Кнута «Искусство программирования».

И наконец, самое главное – деление. Точное деление в $GF(2^n)$ осуществляется очень просто. Так как $\forall a \in GF(2^n), a \neq 0 \quad a^{2^n-1} = 1$ то $a^{-1} = a^{2^n-2}$. В результате деление сводится к умножению на элемент a^{2^n-2} , например в $GF(8) \quad a^{-1} = a^{254}$.

Далее при описании ассемблерных команд будут встречаться такие вещи, как, например, циклический сдвиг слова длиной n , с точки зрения арифметики Галуа, это не что иное, как умножение на x по модулю $x^n + 1$. Сдвиг влево на k разрядов эквивалентен умножению на x^k . Если n и k кратны слову ($n > k$), то умножение осуществляется не командами сдвига, а командами пересылки.



В настоящее время вычисления в полях Галуа, как правило, реализуются программно, а не аппаратно, и широко используются в криптографии (например, алгоритмы RIJNDAEL (AES) и A5/1).

Криптографы также используют арифметику по модулю неприводимых многочленов степени n , коэффициентами которых являются целые числа по модулю q .

1.4.4. Вещественная арифметика

Вещественные вычисления, в отличие от целочисленных, формализовать таким способом невозможно. Все дело в ошибках округления. Рассмотрим кратко историю представления вещественных чисел в компьютере, чтобы понять, почему так происходит, и можно ли с этим что-нибудь сделать.

Если бы вы были программистом 1960-70-х гг., то написание программ с вещественными числами могло бы свести вас с ума. Единого стандарта представления чисел с плавающей запятой, способов округления и арифметических операций не существовало, в результате программы были практически непереносимы, но еще большей проблемой было то, что у разных компьютеров были свои странности реализации вещественной арифметики, и их нужно было знать и учитывать.

На некоторых платформах определенные числа были ненулевыми при сравнении, но являлись нулями при умножении и делении, в результате чего для безопасного их деления предварительно нужно было умножить на 1.0, а затем сравнить с нулем. На других платформах применение этого трюка вызывало недокументированную ошибку переполнения, при том, что реальный выход за границы диапазона отсутствовал. Еще один компьютер при попытке выполнения такой операции произвольно отбрасывал последние 4 значащих бита, большая часть машин возвращала нулевой результат при отыскании разности малых X и Y даже в случае, когда они были неравными, а несколько могли получить ноль даже в случае огромной разницы между X и Y .

В результате работы таких машин выражения « $X=Y$ » и « $X-Y=0$ » вступали в противоречие, и умельцы обходили эту проблему очень хитрыми трюками, например, делали присваивание « $X=(X-X)+X$ » перед операциями умножения и деления, чтобы избежать проблем. При этом мы еще не говорим о сложности компиляторов!

Инициатива создать единый стандарт для представления чисел с плавающей запятой подозрительно совпала с попытками в 1976 году компанией Intel разработать «лучшую» арифметику для новых микропроцессоров к 8086 и i432. За разработку взялись известные специалисты, проф. Джон Палмер и Уильям Кэхэн. Последний в своем интервью высказал мнение, что серьезность, с которой Intel разрабатывала свою арифметику, заставила другие компании объединиться и тоже начать процесс стандартизации, правда, как всегда, альтернативный.

Все были настроены серьезно, ведь очень выгодно продвинуть свою архитектуру вещественной математики и сделать ее стандартной. Свои предложения представили компании DEC, National Superconductor, Zilog и Motorola, производители мейнфреймов Cray, CDC и IBM наблюдали со стороны. Компания Intel, разумеется, тоже представила свою новую арифметику – авторами предложенной спецификации стали У. Кэхэн, Д. Кунен и Г. Стоун, и их предложение сразу прозвали KCS.

Анархия среди арифметики с плавающей точкой заставила пошевелиться даже всемогущий Комитет по стандартизации IEEE, и доктор Роберт Стюарт созвал в 1977 г. специальное совещание, на котором был принят черновик IEEE P754. На второй встрече присутствовали три инженера и математика от Intel, National Semiconductor послал двух человек, Motorola была представлена Джоэлом Бони (который в 1984 проектировал микропроцессор MC68881) и даже Zilog послала лоббиста для продвижения своей архитектуры Z8070 + Z8000.

Практически сразу же были отброшены все предложения, кроме двух: арифметика VAX от DEC и KCS от Intel. Спецификация VAX была значительно проще, уже была реализована в компьютерах PDP-11, и было понятно, как на ней получить максимальную производительность. С другой стороны, в KCS содержалось много полезной функциональности, такой, как специальные и денормализованные числа. В KCS все арифметические алгоритмы заданы строго, и требуется, чтобы в реализации результат с ними совпадал, что позволяет выводить строгие выкладки в рамках этой спецификации. Если раньше математик решал задачу численными методами и доказывал свойства решения, то не было никакой гарантии, что эти свойства сохраняются в программе. Строгость арифметики KCS сделала возможным доказательство теорем, опираясь на арифметику с плавающей запятой.

DEC делала упор на то, что Intel не сможет достичь такой же производительности. Кэхэну и Стоуну запрещено было раскрывать детали архитектуры вещественного микропроцессора 8087, поэтому аргументированно возразить представителям DEC им было нечего. Тем не менее, Intel, как всегда, обоснованно делала ставку на то, что их процессоры должны корректно работать с множеством программ, написанных не только специалистами по численным методам и теории чисел, но и простыми людьми²⁰.

При этом стандарт должен был подразумевать и продвинутые доказательные методы, которые убедили бы специалистов в том, что программы работают именно так, как задумано. К этому добавлялась проблема получения конкурентоспособной скорости в рамках вещественной арифметики. Это были почти взаимоисключающие требования. Неудивительно, что Intel не спешила раскрывать детали их реализации.

Специалисты Intel могли распылить детали их реализации уже широко известных функций, таких, например, как SQRT – без этого IEEE вообще отказался бы обсуждать их стандарт. Тем не менее, когда дело

²⁰ Презрение к простым программистам – основная беда, погубившая многие великие компании, DEC в том числе. К сожалению, они так и не осознали, что массовый продукт, даже уступая элитарному, всегда победит, хотя бы в силу того, что компьютеры нужны везде, а специалистов и железа такого уровня, который пытался поддерживать DEC, на всех никогда не хватит. Это погубило потом их уникальную платформу Alpha, а вслед за ней и саму компанию.

дошло до трансцендентных функций, таких как логарифмы, они были вынуждены молчать, несмотря на то, что их реализация частично превосходила решения конкурентов. Например, формат двойной точности от DEC (8 бит) был равным одинарному Intel, в результате последний превосходил его в точности. DEC позднее увеличил разрядность в своем стандарте, доведя ее до 11 бит, как у машин CDC 6600/7600, где такая точность успешно служила уже более 10 лет.

Однако наибольшей проблемой стали денормализованные числа. Дело в том, что при длительных вычислениях, близких к нулю, очень быстро накапливающаяся ошибка, вкупе с неспособностью большинства платформ корректно обрабатывать малые числа, приводила к неправильному результату. До начала 80-х производители и программисты практически не замечали эту проблему – даже в суперкомпьютерах Cray не было способа корректно отделить малое число от нуля, VAX пошел по их стопам, и эта проблема даже не была затронута в стандарте DEC.

В результате нижний диапазон вещественных чисел для типичных платформ того времени вообще не был документирован и, зачастую, не был известен даже разработчикам. В тех случаях, когда программист подозревал, что ошибка произошла из-за неожиданного обнуления переменной, он должен был постепенно увеличивать подозрительное число до исчезновения проблемы. Заметные сбои, однако, были редкостью, т.к. компьютеры не были самым распространенным оборудованием, обслуживались весьма квалифицированно, и большая группа инженеров всегда могла потратить лишние пару часов, копаясь в исходниках. Сколько программистов попало в эту ловушку? Официальные данные никогда не были предоставлены, но информация из университетов, оснащенных машинами VAX и PDP, говорила не менее чем об одной длительной отладке в месяц(!), вызванной некорректной обработкой малых чисел, и это для ценного оборудования, час работы которого стоил как месячная зарплата специалиста.

В эпоху микропроцессоров все изменилось – количество вычислений возросло на порядки, средний уровень программистов понизился – в результате ошибок стало много. Компания Intel осознала, что игнорирование неизвестной нижней границы вычислений в массовом компьютере приведет к чудовищным последствиям – никто вообще не сможет больше ничего посчитать! Ведь если на несколько сотен машин приходится по ошибке в месяц, а число ПК будет измеряться миллионами, при сравнимой, и даже превосходящей производительности, то компьютеры будут падать раз в пару минут.

Введение денормализованных чисел значительно снизило остроту проблемы, вместе с выходом из абсурдной ситуации, когда результаты выполнения $X = Y$ и $XY = 0$ не согласовывались между собой.

Основным возражением DEC стало то, что денормализованные числа приведут к резкому падению производительности. Intel использовала особую комбинацию алгоритмов управления кэшем, конвейером и регистрами, чтобы обойти это, но детали реализации были тогда еще секретны.

Война продолжалась несколько лет, компания DEC сделала все, чтобы ее спецификацию сделали стандартом. Она даже заручилась поддержкой некоторых авторитетных ученых в том, что арифметика KCS в принципе не может достигнуть такой же производительности. На стороне Intel выступил сам Дональд Кнут и несколько ведущих специалистов в теории чисел, но, в конце концов, она была вынуждена открыть часть спецификаций. Естественно, утверждение своего стандарта на вещественную арифметику невероятно вознесло бы добившуюся этого компанию – по сути, она стала бы монополистом на вычисления с плавающей точкой.

Победа Intel в итоге стала пирровой для их бизнеса. Комитет принял KCS под именем IEEE 754, закат DEC стал неизбежен²¹, но раскрытие деталей реализации перед конкурентами привело к тому, что к 1984 г. вещественный сопроцессор – клон i8087 стал неотъемлемой частью процессоров AMD, MOS Tech., Elxsi, IBM, Motorola, National Semiconductor, Weitek, Zilog, AT&T и всех остальных. Планы монополии провалились, но ненадолго, следующее сражение произошло уже между RISC и CISC системами команд, но об этом в следующей части.

Разработчики KCS победили, и их детище воплотилось в стандарт, работающий до сих пор. Числа с плавающей запятой в нем представлены в виде знака (s), мантиисы (M) и порядка (E) следующим образом:

$$(-1)^s \cdot 1.M \cdot 2^E$$

В новом стандарте IEEE754-2008, кроме чисел с основанием 2, присутствуют числа с основанием 10 (decimal), числа с плавающей запятой. Обычный стандарт допускает следующие особые числа: 0, ∞ и NaN (not a number – не число). Стандарт IEEE 754-2008, помимо этого, определяет два типа NaN: Quiet и Signaling. Первый молча передает значение NaN в последующие вычисления, в то время как операции над вторым приводят к исключению.

В IEEE754 число «0» представляется значением с порядком, равным $E = E_{min} - 1$ (для single это -127) и нулевой мантиисой. Введение нуля как самостоятельного числа (т.к. в нормализованном представлении нельзя представить ноль) позволило избежать многих странностей в арифметике. И хотя операции с нулем нужно обрабатывать отдельно, обычно они выполняются быстрее, чем с обычными числами.

Также в IEEE754 предусмотрено представление для специальных чисел, работа с которыми вызывает исключение. К таким числам относится бесконечность $\pm\infty$ и неопределенность (NaN). Эти числа позволяют вернуть адекватное значение при переполнении. Бесконечности представлены как числа с порядком

²¹ всего через десять лет они уже были на грани банкротства

$E = E_{max} + 1$ и нулевой мантисой. Получить бесконечность можно при переполнении и при делении ненулевого числа на ноль. Бесконечность при делении разработчики определили исходя из определения пределов, соответственно, $\pm Const/0 = \pm \infty$. При $0/0$ результатом будет неопределенность.

Неопределенность или NaN – это представление, придуманное для того, чтобы арифметическая операция могла всегда вернуть какое-то не бессмысленное значение. В IEEE754 NaN представлен как число, в котором $E = E_{max} + 1$, а мантисса не нулевая. Любая операция с NaN возвращает NaN. При желании в мантиссу можно записывать информацию, которую программа сможет интерпретировать. Стандартом это не оговорено, и мантисса чаще всего игнорируется.

Как можно получить NaN? Одним из следующих способов:

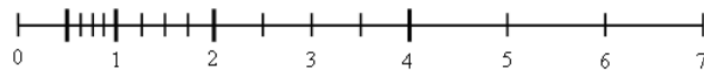
- $-\infty + \infty$
- $0 \cdot \infty$
- $0/0$
- ∞/∞
- $\sqrt{x}, x < 0$

По определению $NaN \neq NaN$, поэтому для проверки значения переменной нужно просто сравнить ее с собой.

Внимательный читатель, вероятно, заметил, что в описанном представлении чисел с плавающей запятой существуют два нуля, которые отличаются только знаком. Так, $3 \cdot (+0) = +0$, а $3 \cdot (-0) = -0$, но при сравнении $+0 = -0$. В стандарте знак сохранили умышленно, чтобы выражения, которые в результате переполнения или потери значимости превращаются в бесконечность или в ноль, при умножении и делении все же могли представить максимально корректный результат. Например, если бы у нуля не было знака, выражение $1/(1/x) = x$ не выполнялось бы верно при $x = \pm \infty$, так как $1/+\infty$ и $1/-\infty$ равны 0. Еще один пример: $(+\infty/0) + \infty = +\infty$, тогда как $(+\infty/-0) + \infty = NaN$.

Чем бесконечность в данном случае лучше, чем NaN? Тем, что если в арифметическом выражении появился NaN, результатом всего выражения всегда будет NaN. Если же в выражении встретилась бесконечность, то результатом может быть ноль, бесконечность или обычное число с плавающей запятой.

Что же такое знаменитые денормализованные (subnormal) числа, рассмотрим на простом примере. Пусть имеем нормализованное представление с длиной мантиссы $|M| = 2$ (плюс один бит нормализации) и диапазоном значений порядка $-1 \leq E \leq 2$. В этом случае получим 16 чисел:



Крупными штрихами показаны числа с мантиссой, равной 1,00. Видно, что расстояние от нуля до ближайшего числа (0 – 0.5) больше, чем от этого числа к следующему (0.5 – 0.625). Это значит, что разница двух любых чисел от 0.5 до 1 даст 0, даже если эти числа не равны. Что еще хуже, между 0.5 и 0 попадает разница чисел, больших 1. Например, $1.5 - 1.25 = 0$.

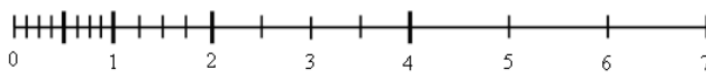
В «околонулеву ю» подпадает не каждая программа. Как уже упоминалось, согласно неофициальной статистике 70-х годов в среднем каждый компьютер сталкивался с такой проблемой один раз в месяц. Учитывая, что компьютеры приобретали массовость, разработчики KCS посчитали эту проблему достаточно серьезной, чтобы решать ее на аппаратном уровне. Предложенное ими решение состояло в следующем. Мы знаем, что при $E = E_{min} - 1$ и нулевой мантиссе число считается равным нулю. Если же мантисса не нулевая, то число считается не нулевым, его порядок полагается $E = E_{min}$, причем неявный старший бит мантиссы полагается равным нулю. Такие числа называются денормализованными.

Строго говоря, числа с плавающей запятой теперь имеют вид:

$(-1)^S \cdot 1.M \cdot 2^E$ если $E_{min} \leq E \leq E_{max}$ это нормализованные числа и

$(-1)^S \cdot 0.M \cdot 2^{E_{min}}$ если $E = E_{min} - 1$ это денормализованные числа.

Вернемся к примеру. Наш $E_{min} = -1$. Введем новое значение порядка, $E = -2$, при котором числа являются денормализованными. В результате получаем новое представление чисел:



Интервал от 0 до 0.5 заполняют денормализованные числа, что дает возможность не проваливаться в 0 в рассмотренных выше примерах. Это сделало представление более устойчиво к ошибкам округления для чисел, близких к нулю. Но роскошь использования денормализованного представления чисел в процессоре не дается бесплатно. Из-за того, что такие числа нужно обрабатывать по-другому во всех арифметических операциях, трудно сделать работу в такой арифметике эффективной. Это накладывает дополнительные сложности при реализации АЛУ в процессоре. И хотя денормализованные числа очень полезны, они не яв-

ляются панацеей, и за округлением до нуля все равно нужно следить. Поэтому эта функциональность стала камнем преткновения при разработке стандарта и встретила самое сильное сопротивление.

Очередность чисел в IEEE754

Одна из удивительных особенностей представления чисел в формате IEEE754 состоит в том, что порядок и мантисса расположены друг за другом таким образом, что вместе образуют последовательность целых чисел $\{n\}$, для которых выполняется

$$n < n + 1 \Rightarrow F(n) < F(n + 1),$$

где $F(n)$ – число с плавающей запятой, образованное от целого n разбиением его битов на порядок и мантиссу. Поэтому если взять положительное число с плавающей запятой, преобразовать его к целому и прибавить единицу, мы получим следующее число, которое представимо в этой арифметике.

На Си это можно сделать так:

```
float a=0.5;
int n = *((int*) &a);
float b = *((float*) &((++n)));
printf("После %e следующее число: %e, разница (%e)\n", a, b, b-a);
```

Этот код будет работать только на архитектуре с 32-битным int.

Теперь рассмотрим особенности арифметики с плавающей запятой, к которым нужно проявить особую осторожность.

Проблема ошибки округления.

С ошибками из-за погрешностей округления в современной арифметике с плавающей запятой встретиться сложно, особенно если использовать двойную точность. Правило округления в стандарте IEEE754 говорит о том, что результат любой арифметической операции должен быть таким, как если бы он был выполнен над точными значениями и округлен до ближайшего числа, представимого в этом формате. Это требует от АЛУ дополнительных усилий, и некоторые опции компилятора (такие как «-ffast-math» в gcc) могут отключить такое поведение. Особенности округления в IEEE754:

- Округление до ближайшего в стандарте сделано не так, как мы привыкли. Математически показано, что если 0.5 округлять до 1 (в большую сторону), то существует набор операций, при которых ошибка округления будет возрастать до бесконечности. Поэтому в IEEE754 применяется правило округления до четного. Так, 12.5 будет округлено до 12. Стандарт IEEE 754-2008 помимо описанного способа округления, используемого по умолчанию, предусматривает еще 4 режима округления: в сторону $\pm\infty$, 0 и до ближайшего в сторону 0;
- Самая опасная операция с точки зрения округления в арифметике с плавающей запятой — это вычитание. При вычитании близких чисел значимые разряды могут потеряться, что может в разы увеличить относительную погрешность;
- Для многих широко распространенных математических формул математики разработали специальную форму, которая позволяет значительно уменьшить погрешность при округлении. Например, расчет $x^2 - y^2$ лучше вычислять используя формулу $(x + y)(x - y)$.

Проблема неассоциативности операций (да-да, вещественные числа с точки зрения компьютера это даже не кольцо!)

В арифметике с плавающей запятой любые операции не ассоциативны! Например,

$(10^{20} + 1) - 10^{20} = 0$, но при этом, $(10^{20} - 10^{20}) + 1 = 1$. Допустим, у нас есть программа суммирования чисел:

```
double s = 0.0;
for (int i=0; i<n; i++) s = s + t[i];
```

Некоторые компиляторы по умолчанию могут переписать код для использования нескольких АЛУ одновременно (будем считать, что n делится на 2):

```
double sa[2], s;
sa[0]=sa[1]=0.0;
for (int i=0; i<n/2; i++) {
    sa[0]=sa[0]+t[i*2+0];
    sa[1]=sa[1]+t[i*2+1];
}
S=sa[0]+sa[1];
```

Так как операции суммирования не ассоциативны, эти две программы могут выдать различный результат.

Проблема констант

Помните, что не все десятичные числа имеют двоичное представление с плавающей запятой. Например, число 0.2 будет представлено как 0.200000003 с одинарной точностью. Соответственно, $0.2 + 0.2 \neq 0.4$! Результатом будет ≈ 0.4 . Абсолютная погрешность в отдельном случае может и не высока, но если использовать такую константу в цикле, можем получить накопленную погрешность. Также вещественные числа никогда нельзя применять в управляющей логике программы, например `if 0.2 + 0.2 = 0.3` может не сработать, потому что $0.2 + 0.2$ даст 0.400000004 и т.п.

Проблема сравнения и выбора минимального

Допустим, из двух значений нам нужно выбрать минимальное. В Си это можно сделать одним из следующих способов:

- `x < y? x: y`
- `x <= y? x: y`
- `x > y? x: y`
- `x >= y? x: y`

Часто компилятор считает их эквивалентными и всегда использует первый вариант, так как он выполняется за одну инструкцию процессора. Но если мы учтем ± 0 и NaN, эти операции никак не эквивалентны:

x	y	<code>x < y? x: y</code>	<code>x <= y? x: y</code>	<code>x > y? x: y</code>	<code>x >= y? x: y</code>
+0	-0	-0	+0	+0	-0
NaN	1	1	1	NaN	NaN

Помимо этого, нельзя складывать/вычитать большие и маленькие числа, которые не пересекаются по мантиссе. В цикле с накоплением есть вероятность выхода за пределы работы операции суммирования, т.к. накапливающая переменная может содержать слишком большое число для дальнейшего суммирования и алгоритм перестанет правильно работать. В процессе необходимо следить, чтобы все промежуточные результаты были представлены в одном компьютерном формате. Например, вычисления переменных `single`, а промежуточные вычисления и константы могут быть в `double`. Это приведет к ошибке, потому что форматы `single` и `double` по-разному кодируют одинаковые числа и не совпадают.

Очень распространенная ошибка при работе с вещественными числами возникает при проверке на равенство. Например,
`float fValue = 0.2;`
`if (fValue == 0.2) DoStuff();`

Ошибка здесь, во-первых, в том, что 0.2, как уже говорилось, не имеет точного двоичного представления, а во-вторых 0.2 это константа двойной точности, а переменная `fValue` – одинарной, и никакой гарантии о поведении этого сравнения нет.

Лучший, но все равно ошибочный способ, это сравнивать разницу с допустимой абсолютной погрешностью:
`if (fabs(fValue - fExpected) < 0.0001) DoStuff(); // fValue=fExpected?`

Недостаток такого подхода в том, что погрешность представления числа увеличивается с ростом самого этого числа. Так, если программа ожидает 10000, то приведенное равенство не будет выполняться для ближайшего соседнего числа (10000.000977). Это особенно актуально, если в программе имеется преобразование из одинарной точности в двойную. Выбрать корректную процедуру сравнения сложно, Брюс Доусон²², например, предлагает делать это путем преобразования вещественных чисел к целочисленным, например, так:

```
bool AlmostEqual2sComplement(float A, float B, int maxUlps) {  
    // maxUlps не должен быть отрицательным и не слишком большим, чтобы  
    // NaN не был равен ни одному числу  
    assert(maxUlps > 0 && maxUlps < 4 * 1024 * 1024);  
    int aInt = *(int*)&A;  
    // Уберем знак в aInt, если есть, чтобы получить правильно упорядоченную последовательность  
    if (aInt < 0) aInt = 0x80000000 - aInt;  
    // aInt &= 0x7fffffff;  
    // Аналогично для bInt  
    int bInt = *(int*)&B;  
    if (bInt < 0) bInt = 0x80000000 - bInt;  
    /* aInt &= 0x7fffffff; */  
    unsigned int intDiff = abs(aInt - bInt);  
    if (intDiff <= maxUlps)  
        return true;  
    return false;  
}
```

²² <http://www.cygus-software.com/papers/comparingfloats/comparingfloats.htm>

В этой программе `maxUlp` (от `Units-In-Last-Place`) – это максимальное количество чисел с плавающей запятой, которое может лежать между проверяемым и ожидаемым значением. Другой смысл этой переменной – это количество двоичных разрядов (начиная с младшего), которое в сравниваемых числах разрешается упустить. Например, `maxUlp=16`, означает, что младшие 4 бита ($\log_2 16$) могут не совпадать, а числа все равно будут считаться равными. При этом, при сравнении с числом 10000 абсолютная погрешность будет равна 0.0146, зато при сравнении с 0.001, погрешность будет менее 10^{-8} .

Ну и наконец, вы думаете, что если процессоры полностью соответствуют стандарту IEEE754, то любая программа, использующая стандартные типы данных (такие как `float/double` в Си), будет выдавать один и тот же результат на разных компьютерах? На самом деле, на соответствие стандарту влияет компилятор и опции оптимизации. Уильям Кэхэн написал программу на Си (есть версия и для Фортрана), которая позволяет проверить, удовлетворяет ли связка «архитектура+компилятор+опции» IEEE754. Называется она «Floating point paranoia» и ее исходные тексты доступны для скачивания²³. Аналогичная программа доступна для GPU²⁴. Так, например, компилятор Intel (`icc`) по умолчанию использует облегченную модель IEEE754, и в результате не все тесты выполняются. Опция «`-fp-model precise`» позволяет компилировать программу с точным соответствием стандарту. В компиляторе GCC есть опция «`-ffast-math`», использование которой приводит к несоответствию IEEE754.

Сейчас арифметика с плавающей запятой почти совершенна. Практически всегда наивный подход сработает, и программа, не учитывающая все ее особенности, выдаст правильный результат, а описанные подводные камни касаются только экзотических случаев. Но нужно всегда оставаться бдительным: в таком вопросе как компьютерная математика легко наступить на грабли. Кстати, деньги, например, нельзя хранить в виде числа с плавающей запятой, т.к. в этом случае нельзя выделить значимые разряды, и в результате ошибок округления баланс не сойдется. Если в языке программирования нет типов данных с фиксированной запятой, можно выйти из положения и хранить деньги в виде целого числа, подразумевая копейки (иногда доли копеек).

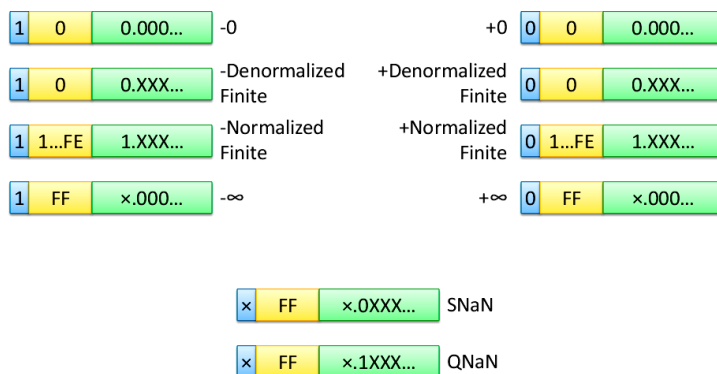
На самом деле главная проблема даже не в округлении как таковом, а именно в плавающей точке и в ограниченности длины мантииссы. При добавлении малых величин к большим, младшие разряды могут банально потеряться, т.к. просто не влезут в мантииссу. Простой пример: найти $\sum_{i=1}^k 1/n_i$ для больших k . При малых k нет никакой разницы, суммировать от меньшего к большему или наоборот, но как только k становится достаточно большим, разница становится заметной на глаз, например:

```
#include <stdio.h>
void main(int argc, char* argv[]) {
    float sum1 = 0.f;
    float sum2 = 0.f;
    int n = 20000;
    int i;
    printf("%i numbers\n", n);
    for ( i = 1; i <= n; ++i ) {
        sum1 += 1.f / i;
        sum2 += 1.f / (n + 1 - i);
    }
    printf("%f\n", sum1 - sum2);
}
```

даст разницу 0.000007 для $k = 1000$, 0.000009 для $k = 10000$ и 0.000023 для $k = 20000$ итераций. Происходит это вот почему. При подсчёте `sum1` в неё сразу кладётся 1, фиксируя экспоненту, после чего начинают добавляться всё более мелкие числа. В итоге последние из них попросту не помещаются в мантииссу. При подсчёте `sum2`, напротив, сначала берутся более мелкие числа, которые накапливаются постепенно, отбрасывая младшие разряды по одному по мере роста экспоненты. Конечно, `double` немного отодвигает эту проблему, но в точных расчётах и его может не хватить (особенно с учётом накладываютой погрешности округления).

²³ http://orion.math.iastate.edu/burkardt/c_src/paranoia/paranoia.html

²⁴ <http://www.cs.unc.edu/~jbr/projects/paranoia/>



Все возможные виды чисел и не чисел
в формате Single Precision 32-bit IEEE754-2008

	Double Precision (64бита)	Single Precision (32 бита)
Нулевые числа	FFFF FFFF FFFF FFFF	FFFF FFFF
Денормализованные числа	FFF0 0000 0000 0001	FF80 0001
	FFF0 0000 0000 0000	FF80 0000
	FFFF FFFF FFFF FFFF	8F7F FFFF
	8010 0000 0000 0000	8080 0000
	800F FFFF FFFF FFFF	807F FFFF
Нормализованные числа	8000 0000 0000 0001	8000 0001
	8000 0000 0000 0000	8000 0000
	0000 0000 0000 0000	0000 0000
	0000 0000 0000 0001	0000 0001
	000F FFFF FFFF FFFF	007F FFFF
	0010 0000 0000 0000	0080 0000
Нечисла (NaNs)	7FEF FFFF FFFF FFFF	7F7F FFFF
Бесконечности	7FF0 0000 0000 0000	7F80 0000
	7FF0 0000 0000 0001	7F80 0001
	7FFF FFFF FFFF FFFF	7FFF FFFF

Полный диапазон чисел IEEE754-2008

1.4.5. К чему приводят ошибки

F-22 против 180-го меридиана

F-22 Raptor на сегодняшний день считается самым дорогим истребителем в мире и одним из сложнейших образцов техники, когда-либо созданных руками человека. Стоимость производства одной такой машины оценивается в 146,2 млн. долларов (на 2008 год), а полная цена с учётом всех косвенных затрат и при ожидаемом объёме производства – 350-400 млн. Себестоимость лётного часа F-22 составляет около 44 тыс. долларов, что также превышает все мыслимые для самолета цифры. При этом назвать F-22 технически совершенным отнюдь нельзя – в различных лётных происшествиях было потеряно уже пять истребителей, и это только случаи, получившие огласку в СМИ. Наиболее интересный нам случай произошел в 2007 г. во время перегона самолетов с Гавайских островов на базу ВВС Кадена на Окинаве. В перелете участвовали шесть самолетов. После пересечения 180-го меридиана (международной линии перемены дат, это условная линия, по разные стороны которой местное время одно и то же, с точностью до часового пояса, но календарные даты различаются на одни сутки) в истребителях полностью отключились все навигационные компьютеры, частично связь и даже система подачи топлива.

Перемена дат осуществляется (и вообще имеет смысл) лишь при использовании местного времени. При пересечении линии перемены дат необходимо либо прибавлять, либо вычитать одни сутки – в зависимости от того, в каком направлении осуществляется движение. По всей видимости, этот парадокс Земного шара, осознанный еще участниками экспедиции Магеллана, был позабыт разработчиками навигационного компьютера F-22 Raptor, и самолет, попытавшись синхронизировать часы, решил, что провалился в прошлое. Ошибка вызвала цепное обрушение всех систем, использующих GPS, вплоть до индикаторов топлива. Лишь одному пилоту удалось связаться с экспертами разработчика (компании Lockheed Martin). Несколько пилотов попытались перезагрузить ПО истребителя в полете, но результата не последовало.

Истребители и их пилоты чудом уцелели. Возвращение на Гавайские острова потребовало дополнительной дозаправки в воздухе и визуального ориентирования на самолеты-заправщики для прокладывания курса. После тщательного изучения проблема была поправлена патчем Upgrade 3.5 и самолеты-таки смогли пересечь злополучный меридиан²⁵.

Деление на ноль в Motorola.

Знаменитый процессор Motorola 68k использовался одно время (а в варианте 88k используется и сейчас) для автопилотов истребителей. В одном из испытаний задачей был полет с огибанием рельефа с севера на юг над Израилем. Истребитель прекрасно пролетел на автопилоте над равнинной частью, над горной частью, над долиной реки Иордан и приближался к Мёртвому морю. Однако при подлете к нему неожиданно происходит общий сброс процессора, автопилот выключается на полном ходу, пилоты переходят на ручное управление и сажают истребитель. Процессор отправили на доработку и тестирование. Все тесты снова прошли без сбоев. В реальности же истребитель пролетел над всеми территориями, но при подлете к Мёртвому морю ситуация повторилась: общий сброс, выключение автопилота, ручная посадка.

Длительные тесты не могли выявить никаких изъянов. После продолжительных попыток было найдено, что программы автопилота при вычислении параметров управления производили деление на значение текущей высоты истребителя над уровнем океана. При подлете к Мёртвому морю высота над уровнем океана становилась нулевой (Мёртвое море расположено ниже уровня океана), и процессор при делении на ноль выбрасывал исключение и перезагружался.

F-16 вверх ногами

Испытания американского истребителя F-16 проводились в северном полушарии, но на заключительном этапе самолет решили проверить в Латинской Америке. При переводе самолета в режим автопилота он автоматически развернулся вверх ногами. Произошло это, как несложно догадаться, из-за некорректной обработки отрицательных чисел системой управления.

Ошибка наведения

В последнем конфликте в Афганистане двое наводчиков-наблюдателей подсвечивали цель для наведения на нее управляемой авиабомбы. После сброса бомбы в GPS-приемнике сели батарейки и солдаты их быстро заменили. После отключения питания и, соответственно, жесткой перезагрузки произошел сброс всех настроек прибора в значения по умолчанию и координаты, цели инициализировались текущими координатами приемника. Бомба прилетела прямо в наводчиков, результат очевиден.

²⁵ Отметим, что в данном случае, вина лежит скорее не на проектировщиках самолета, а на заказчике – Пентагоне, который хотел получить сразу самый незаметный, самый мощно вооруженный, самый быстрый, самый автономный и с наибольшей высотой полета истребитель, да еще и за более-менее приемлемую цену и закончить его побыстрее. Очевидно, от того, что генералы заседают не в Москве, а в Вашингтоне, мозгов у них не прибавляется, но Lockheed Martin честно попыталась удовлетворить эти взаимоисключающие требования.

Испытания Су-24

Редкий пример не опасной, а скорее забавной ошибки. На испытаниях Су-24 регулярно случался отказ аппаратуры бомбометания. Причем происходило это только в том случае, если на цель заходил летчик-испытатель Ильюшин. Причина оказалась в том, что только он заходил на цель с точностью, превышавшей машинную точность отечественного процессора, в результате возникало исключение деления на ноль, и аппаратура отключалась.

Гибель Arian V

Долгое время эта ошибка, приведшая к убыткам в размере более \$500.000.000, считалась самой дорогостоящей в программной системе. Ошибка состояла в том, что без изменений использовался модуль расчета траектории из системы управления Ариан IV. В нем горизонтальная составляющая скорости ракеты представлялась 16-битным целым числом. Ариан V могла выдерживать большие ускорения и, соответственно, иметь траекторию большей кривизны, из-за чего в ходе полета значение горизонтальной скорости вышло за пределы представимых 16-ю битами чисел. Специальной процедуры обработки такой ситуации не было, поэтому возникшее исключение обрабатывалось модулем обработки общих сбоев, который остановил «ошибочный» процесс и запустил новый с теми же исходными данными, что вновь привело к той же ошибке и т.д. В результате система не смогла вычислить правильное текущее положение ракеты и стала использовать ранее полученные данные. С каждой секундой ошибка в определении координат нарастала в геометрической прогрессии, это приводило все к новым попыткам процессора «скорректировать» курс, ракета отклонилась от траектории настолько, что это стало угрожать безопасности, и она была уничтожена сработавшей системой самоуничтожения.

Коллапс бортовой ЭВМ Yorktown

Корабль ВМС США «Йорктаун», проект CG48 – это ракетный крейсер класса «Тикондерога», названный в честь сражения при Йорктауне, произошедшего в ходе Войны за независимость США. Корабль был спущен на воду в 1983 г. и прослужил больше 20 лет до списания в 2004 г.

В начале 1990-х он был переоборудован по т.н. проекту «умный корабль» (smart ship), важнейшие системы жизнеобеспечения которого управляются компьютерами без участия человека. Основной бортовой ОС была тогда еще новая серверная Windows NT 4.0. В один из рейсов оператор-калибровщик клапанов топливной системы случайно записал в одну из ячеек расчетной таблицы нулевое значение. Система выполнила деление на ноль и зависла. Само по себе это было бы не так страшно, но в сочетании с некорректной обработкой главным процессором исключения, бортовой сервер, управляющий локальной сетью корабля, упал с ошибкой переполнения памяти, что потребовало его перезагрузки и восстановления, на что ушло более двух с половиной часов. В течение этого времени крейсер был лишен хода, систем вооружения и жизнеобеспечения, связи и всего прочего²⁶.

Ошибка системы ПВО Patriot

25 февраля 1991 г. во время первой войны в Персидском заливе американская система ПВО Patriot не смогла сбить иракскую ракету «Скад», которая в результате попала в барак американской армии, убив 28 и ранив больше ста человек. Причиной промаха было накопление ошибок округления за время работы системы. Время в бортовом компьютере Patriot измерялось в десятых долях секунды, а числа были представлены в 24-битном двоичном формате с плавающей точкой. Как известно, при представлении $1/10$ как двоичной дроби возникает ошибка (такое число представляется бесконечной периодической дробью, которую нельзя безошибочно округлить). В рассматриваемом случае система Patriot работала без перезагрузки около 100 часов. За это время накопление погрешности определения времени дало ошибку около $1/3$ секунды. Поскольку ракета «Скад» летит по баллистической траектории со скоростью 1700 м/с, ошибка в определении ее координат достигла 500 метров, и Patriot промахнулся с уже упомянутыми последствиями. Стоит отметить, что за всю войну этот инцидент унес больше всего жизней американских солдат, при этом, для его предотвращения всего-то нужно было при проектировании процессора наведения выбрать базовым тактом не $1/10$ секунды, а степень двойки, которая преобразуется без погрешностей вообще.

²⁶ Этому кораблю вообще не очень везло – не менее примечательный случай случился с ним в 1988, когда советские сторожевики и американские эсминцы не поделили территориальные воды в Черном море. Результатом являлось то, что brave советские пограничники, по приказу адмирала, таранили ракетный крейсер, на полном ходу проехавшись своим катером по половине левого борта и снеся все, что торчало, от леера до пусковых установок ПКР «Гарпун», которые только чудом не взорвались, угробив бы при этом оба корабля.

1.5. Алгоритмизация и машина Тьюринга

1.5.1. Теория графов

Одним из наиболее ценных инструментов для анализа алгоритмов является теория графов. Было бы несправедливо не рассказать о ней. Пусть V – непустое множество, $V(2)$ – множество всех его двухэлементных подмножеств. Пара (V, E) , где E – произвольное подмножество множества $V(2)$, называется графом (неориентированным графом). Элементы множества V называются вершинами графа, а элементы множества E – ребрами.

Множества вершин и ребер графа G обозначаются соответственно V_G и E_G . Вершины и ребра графа называются его элементами. Число $|V_G|$ вершин графа G называется его порядком и обозначается $|G|$. Если $|G| = n$, $|E_G| = m$, то граф называют (n, m) -графом.

Говорят, что две вершины u и v графа смежные, если множество $\{u, v\}$ является ребром, и не смежные в противном случае. Если $e = \{u, v\}$ – ребро, то вершины u и v называют его концами. В этой ситуации говорят также, что ребро e соединяет вершины u и v .

Граф порядка n называется помеченным, если его вершинам присвоены некоторые метки. Отождествив каждую из вершин графа с ее номером (и, следовательно, множество вершин – с множеством чисел $\{1, \dots, n\}$), определим равенство помеченных графов G и H одного и того же порядка: $G = H$ тогда и только тогда, когда $E_G = E_H$.

Два ребра называются смежными, если они имеют общий конец. Вершина v и ребро e называются инцидентными, если v является одним из концов ребра e , и не инцидентными в противном случае. Множество всех вершин графа G , смежных с некоторой вершиной v , называется окружением вершины v и обозначается $N_G(v)$ или просто $N(v)$.

Степенью вершины графа G называется число инцидентных ей ребер, т.е. число вершин в ее окружении. Будем обозначать степень вершины v через $\deg G$ (или $\deg v$). Тем самым $\deg v = |N(v)|$. Максимальная и минимальная степени вершин графа G обозначаются символами $\Delta(G)$ и $\delta(G)$ соответственно.

Чередующаяся последовательность $v_1, e_1, \dots, e_n, v_{n+1}$ вершин и ребер графа такая, что $e_i = v_i \cdot v_{i+1}$, $i \in 1, \dots, n$ называется маршрутом, соединяющим вершины v_1 и v_{n+1} . Очевидно, что для задания маршрута в графе достаточно задать последовательность $v_1, \dots, v_{n+1}$ его вершин, либо последовательность $e_1, \dots, e_n$ его ребер.

Вершина v называется достижимой из вершины u , если существует (u, v) -маршрут. Любая вершина считается достижимой из себя самой. Маршрут называется цепью, если все его ребра различны, и простой цепью, если все его вершины, кроме, возможно, крайних, различны.

Маршрут называется циклическим, если $v_1 = v_{n+1}$. Циклическая цепь называется циклом, а циклическая простая цепь – простым циклом. Число ребер в маршруте называется его длиной. Цикл длины 3 часто называют треугольником. Длина всякого цикла не менее трех, если речь идет о простом графе, поскольку в таком графе нет петель и кратных ребер. Минимальная из длин циклов графа называется его обхватом.

Пусть G – помеченный граф порядка n , $V_G = \{1, \dots, n\}$. Определим бинарную матрицу $A = A(G)$, положив

$$A_{ik} = \begin{cases} 1, & \{i, k\} \in E_G \\ 0, & \{i, k\} \notin E_G \end{cases}$$

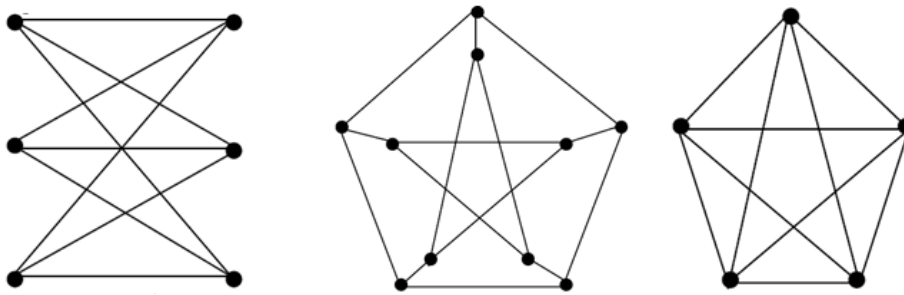
$A(G)$ называется матрицей смежности графа G . Это симметричная матрица с нулями на диагонали. Число единиц в строке равно степени соответствующей вершины.

Граф называется двудольным, если существует такое разбиение множества его вершин на две части (доли), что концы каждого ребра принадлежат разным частям. Если при этом любые две вершины, входящие в разные доли, смежны, то граф называется полным двудольным.

Если G – двудольный граф с долями X и Y и множеством ребер E , то пишут $G = (X, Y, E)$. Полный двудольный граф, у которого $|X| = p$ и $|Y| = p$, обозначается $K_{p,p}$. Аналогично двудольным определяются k -дольный и полный k -дольный графы для $k = 3, 4, \dots$

Деревом называется связный граф, не содержащий циклов. Любой граф, не содержащий циклов, называется ациклическим (или лесом). Пусть H – остоновой подграф произвольного графа G . Если на каждой компоненте связности графа G графом H порождается дерево, то H называется остовом (или каркасом) графа G . Очевидно, что в каждом графе существует остов: разрушая в каждой компоненте циклы, т.е. удаляя лишние ребра, приходим к остову.

Вот несколько знаменитых графов (слева направо): граф Понтрягина-Куратовского II-го рода $K_{3,3}$, граф Петерсена, граф Понтрягина-Куратовского I-го рода K_5 :

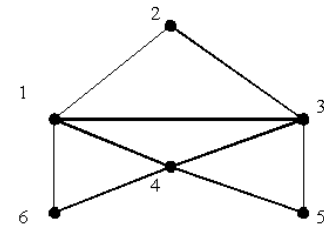


Числом вершинной связности (или просто числом связности) $\chi(G)$ графа G называется наименьшее число вершин, удаление которых приводит к несвязному или одновершинному графу. Пусть G – граф порядка $n > 1$. Числом реберной связности $\lambda(G)$ графа G назовем наименьшее число ребер, удаление которых приводит к несвязному графу. Число реберной связности графа будем считать равным нулю, если этот граф одновершинный.

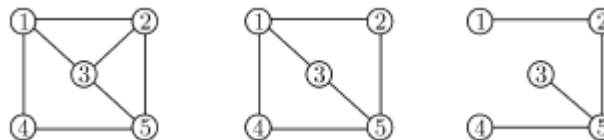
Вершина v графа G называется точкой сочленения (или разделяющей вершиной), если граф G/v имеет больше компонент, чем G . В частности, если G связен и v – точка сочленения, то G/v не связен. Аналогично ребро графа называется мостом, если его удаление увеличивает число компонент. Максимальный k -связный подграф графа называется его k -связной компонентой, или просто k -компонентой.

Плоским графом назовем граф, вершины которого являются точками плоскости, а ребра – непрерывными плоскими линиями без самопересечений, соединяющими соответствующие вершины так, что никакие два ребра не имеют общих точек, кроме инцидентной им обоим вершины. Любой граф изоморфный плоскому графу будем называть планарным. О планарных графах говорят, что они укладываются на плоскости (имеют плоскую укладку).

Граф называется эйлеровым, если существует маршрут без повторения ребер (такой цикл называют эйлеровым), обходящий все вершины графа, т.е. цикл, содержащий все вершины и все ребра лишь 1 раз. Такой граф можно начертить, не отрывая карандаша от бумаги. Последовательность рисования называется эйлеровым циклом – замкнутая цепь, проходящая через все ребра. Получить его позволяет т.н. алгоритм Флери – начинаем с любой вершины и стираем пройденные ребра. При этом по мосту (перешейку) проходим только, если нет других возможностей:



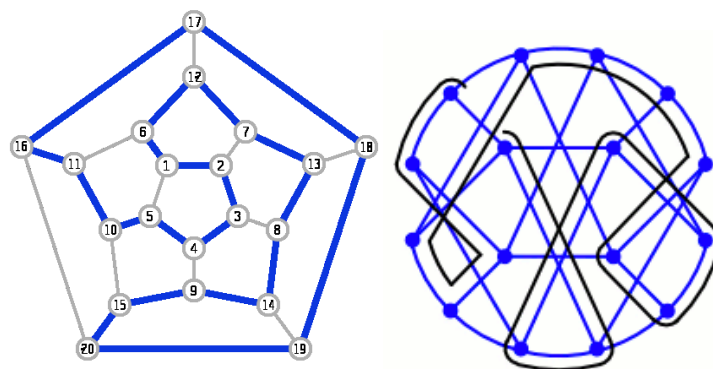
Граф называется полуэйлеровым, если существует маршрут без повторения ребер (эйлеров путь), обходящий все ребра графа ровно один раз. Граф называется гамильтоновым, если в нем существует маршрут, обходящий каждую вершину, кроме первой, только один раз.



Внешне определение гамильтонова цикла похоже на определение эйлерова цикла. Однако есть кардинальное различие в сложности решения соответствующих задач распознавания и построения. Мы видели, что имеется достаточно простой критерий существования эйлерова цикла и эффективный алгоритм его построения. Для гамильтоновых же циклов (и путей) неизвестно никаких просто проверяемых необходимых и достаточных условий их существования, а все известные алгоритмы требуют для некоторых графов перебора большого числа вариантов.

Гамильтонов цикл представляет собой, с комбинаторной точки зрения, просто перестановку вершин графа. При этом в качестве начальной вершины цикла можно выбрать любую вершину, так что можно рассматривать перестановки с фиксированным первым элементом. Самый бесхитростный план поиска гамильтонова цикла состоит в последовательном рассмотрении всех этих перестановок и проверке для каждой из них, представляет ли она цикл в данном графе. Такой способ действий уже при не очень большом числе вершин становится практически неосуществимым ввиду быстрого роста числа перестановок – имеется $(n - 1)!$ перестановок из n элементов с фиксированным первым элементом.

Вот классический пример гамильтонова додекаэдра и гамильтонова цикла в нем:



Определить, гамильтонов ли граф перед нами, сложно – общих теорем на этот счет нет. Немного помогает условие Дирака:

Пусть p - число вершин в графе. Если степень каждой вершины $\geq p / 2$, то граф называется графом Дирака. Такой граф гамильтонов.

Также выручает условие Оре:

Пусть p - число вершин в графе. Если для любой пары несмежных вершин x, y выполнено неравенство $d(x) + d(y) \geq p$ (степени двух несмежных вершин не меньше общего числа вершин), то граф называют графом Оре. Такой граф гамильтонов.

Покажем еще одну интересную задачу теории графов - алгоритм нахождения минимальной сети.

Сетью называется граф, каждому из рёбер (дуг) которого поставлено в соответствие некоторое неотрицательное число (длина, вес, пропускная способность). Наша задача состоит в том, чтобы найти минимальный путь из пункта А в пункт В для связанных между собой дорог на транспортной сети. Примем следующие обозначения:

- Метка вершины 1 всегда полагается равной нулю, метки остальных вершин — бесконечности.
- Все вершины графа помечаются как непосещённые. Когда все вершины посещены, алгоритм завершается.
- Из ещё не посещённых вершин выбирается вершина u , имеющая минимальную длину ребра.
- Рассматриваются всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, соединённые с вершиной u рёбрами, называются «соседами» этой вершины.
- Для каждого «соседа» рассматривается новая длина пути, равная сумме текущей метки u и длины ребра, соединяющего u с этим соседом.
- Если полученная длина меньше метки «соседа», то метка «соседа» заменяется этой длиной. После «осмотра» всех «соседей» вершина u помечается как посещённая.
- На этом алгоритм возвращается к шагу 3.

Этот алгоритм называется алгоритмом Дейкстры. Работает он для графов без отрицательных весов рёбер и находит кратчайшее расстояние от одной из вершин графа до всех остальных.

Как мы уже говорили, граф без циклов называют ациклическим, или лесом. Если он связан, то такой граф называют деревом, т.е. компоненты связности леса являются деревьями. В результате мы представляем как дизъюнктивное объединение деревьев. Вот несколько важных свойств таких графов:

- Граф является лесом тогда и только тогда, когда каждое его ребро – мост.
- Дерево с n вершинами содержит $n - 1$ ребер.
- Пусть в лесе n вершин, m ребер и k компонент связности. Тогда $m = n - k$.
- Граф является деревом тогда и только тогда, когда любые две его вершины соединены ровно одной простой цепью.

Важным представляется также утверждение, носящее имя теоремы Кэли.

Обозначим через P_n число помеченных деревьев с n вершинами. Ясно, что $P_1 = P_2 = 1$. Помеченное дерево с тремя вершинами полностью определяется своей центральной вершиной, поэтому $P_3 = 3$. Если в дереве 4 вершины, то это либо полный двудольный граф $K_{1,3}$, либо простая цепь длины 4.

Существует 4 типа первого вида, а второго уже 12. Дерево с 5 вершинами имеет один из 3 возможных видов:



Теорема Кэли (1898 г.) гласит:

Существует ровно n^{n-2} помеченных деревьев с n вершинами.

Стягивающим или осовным деревом связного графа G является любой его подграф, содержащий все вершины G и являющийся деревом. Существуют эффективные алгоритмы нахождения стягивающего дерева минимального веса для любого графа, например, алгоритм Д. Краскала (1956 г.).

Наконец, для укладок графов необходимо знать следующие важные утверждения:

- Любой граф укладывается в $\mathbb{R}^3$.
- Граф является планарным тогда и только тогда, когда он укладывается на сфере.
- И, наконец, одно из самых знаменитых утверждений, положившее начало теории графов: для любого связного плоского графа справедливо соотношение $n - m + f = 2$, где n, m, f – число вершин, граней и ребер (Л.Эйлер, 1758 г.).

1.5.2. Алгоритмы и вычислимость

В этой части мы поговорим о такой важной вещи, как математическая модель вычислительной машины, такой как конечный автомат или машина Тьюринга. Перед тем как обсудить эти важные вещи, прежде всего нужно разобраться с вопросом, что такое процесс вычислений с точки зрения математики. Очевидно, что речь идет об алгоритмах. Общепринятое определение алгоритма сконструировать достаточно сложно, и в большинстве источников оно вводится через описание свойств, как то наличие начала и конца, однозначность операций и т.п.

Давайте задумаемся, что такое алгоритм. Прежде всего, формально это символическая запись на естественном или искусственном языке. Конечно, они бывают разные – хорошие и не очень, работающие с ошибками, без ошибок и вообще неправильные и неработающие. При этом разница между ними, формально, опять же всего лишь в словах или даже отдельных буквах.

Так кто мешает интерпретировать как алгоритм (пусть и не работающий, возможно) любую произвольную запись на данном языке L ? Среди этого множества обязательно найдутся работающие корректно и неработающие вообще (при этом мы не отличаем, почему он не работает – из-за опечатки в одной букве, или из-за того, что он весь – бессмысленный набор слов).

Ведь с точки зрения компьютера, алгоритм не несет никакого внешнего смысла или контекста – для него это просто последовательность символов, если она совпадает с тем, что он в состоянии интерпретировать как программу – он выполнит ее, если не в состоянии – нет. Ему неважно, почему он не может ее разобрать – потому, что это бессмысленный набор букв, или потому, что это не программа, а сонет Шекспира.

Таким образом, формально алгоритм – это произвольная последовательность слов данного языка и, все они делятся на работающие и не работающие. Не работать они могут по двум причинам – нештатная (т.е. до признака окончания) остановка (пусть даже и на первом же символе, который оказался неправильным) либо заикливание. Отметим, что путем введения повсюду перехода по типу – «не можешь прочесть корректно символ – попробуй снова» все ошибки можно свести к одной – заикливанию.

Таким образом, основная задача теории алгоритмов – овладеть двумя важными вещами: научиться отличать рабочие алгоритмы от заикливающих и, в случае, рабочих научиться придавать им оптимальный с точки зрения быстродействия вид. Для этих целей было разработано много инструментов, и мы познакомимся здесь с простейшими из них.

Естественно, эта задача нетривиальна – например, в такой постановке вопроса вполне можно написать алгоритм, решающий до сих пор нерешенную проблему Гольдбаха, даже два – один все время говорит «да», второй «нет». Только неясно, какой из них правильный. В данном контексте это не издевательство над разумом, а осознание глубины проблемы.

Назовем вычислимой функцией $f(n)$, где $n, f(n) \in \mathbb{Z}$ функцию, для которой существует алгоритм, вычисляющий ее значение (но не обязательно приводящий к результату, соответственно, вычислимые функции делятся на алгоритмически разрешимые и не разрешимые, но об этом ниже).

Как уже говорилось, алгоритм есть запись на формальном языке. Алгоритм применяется к каким-то словам и возвращает в результате тоже слова, если кодировать входные и выходные данные целыми числами – получим вычислимую функцию. Такой подход позволяет пронумеровать все вычислимые функции: $f_1(n), \dots, f_k(n), \dots$ (т.к. можно пронумеровать все соответствующие им программы-алгоритмы, сначала нумеруются все программы из 1 буквы, потом из 2-х и т.д.).

Здесь возникает определенная методологическая проблема – одной и той же функции может соответствовать бесчисленное множество программ. Формально мы можем, например, изменить алгоритм, добавив умножение и деление на константу – результат не изменится, но алгоритм другой! Аналогично есть и существенно разные программы, вычисляющие одно и то же, например, факториал. В результате, наше определение нуждается в некоторой модификации. Вычислимой функцией мы назовем счетное множество эквивалентных алгоритмов, дающих на выходе одинаковый результат при одинаковых входных данных.

Вопрос о вычислимости тех или иных функций не так прост. Например, функция:

$$f(n) = \begin{cases} 1, & \text{если в десятичной записи } \pi \text{ есть } n \text{ нулей подряд} \\ 0, & \text{в остальных случаях} \end{cases}$$

Эта функция должна выполнять десятичное разложение π , до тех пор, пока в записи не встретится n нулей подряд. Это, если они там встретятся. Однако в случае $f(n) = 0$ для некоторых n , указанная процедура никогда не остановится, и не ясно, существует ли эффективная процедура. Поэтому $f(n)$ невычислима. С другой стороны, очевидно, что либо $f(n) \equiv 1$, либо $f(n > n_0) \equiv 0$. В любом из этих случаев существует тождественный алгоритм, возвращающий либо единицы, либо, после некоторого n_0 , нули без всякой связи с изначальной задачей, значит функция вычислима. Однако с тем же успехом можно возразить, что и в изначальном варианте алгоритм существует, но его невозможно эффективно указать. Вопрос, кто прав не имеет смысла, так как перед нами некорректно поставленная задача (далее будет ясно – почему), но этот пример поясняет, что термином «вычислимость» грамотно пользоваться непросто.

Зато, например, вычислима функция

$$g(n) = \begin{cases} 1, & \text{если в десятичной записи } \pi \text{ есть } n \text{ нулей подряд} \\ 0, & ? \end{cases}$$

на вид похожая, но не обязанная останавливаться в противном случае.

Взяв в качестве алфавита английский и все служебные символы, мы можем перекодировать все буквы в двоичной системе, составить из них бесконечное множество $\{P_n\}$ конечных последовательностей длины n и тем самым получим все возможные алгоритмы-программы. Среди них, конечно, будет все, что угодно – все тексты Толстого, последовательности из одних запятых и вообще все, что было и будет написано, включая все реально работающие и полезные программы.

Основной проблемой теперь станет создание подходящей модели (типа конечных автоматов, рекурсивных функций и т.д.), позволяющей, грубо говоря, отсеять из $\{P_n\}$ не интересующий нас синтаксический мусор и непрофильные тексты. Однако никакие меры не помогут нам предотвратить заикливание на некоторых входах некоторых полезных программ, отобранных на предыдущем этапе. В результате, любая попытка определить произвольную вычислимую функцию для любого n обречена на неудачу.

Вот классический пример:

С помощью упомянутой нумерации определим функцию $g(n) = f_n(n) + 1$. Эта функция невычислима, так как при $n = 1$ ее значение отличается от $f_1(1)$, при $n = 2$ ее значение отличается от $f_2(2)$ и так далее. С другой стороны, она опять-таки вычислима в любом разумном смысле, так как к вычислимому $f_n(n)$ заведомо можно прибавить 1. Этот трюк раскрыть просто – перед нами стандартное т.н. диагональное рассуждение, которое можно использовать, например, при доказательстве бесконечности вычислимых функций. В самом деле, если их число конечно и существует последняя $f_n(n)$, то мы всегда можем определить указанную функцию $g(n)$, которая:

1. не совпадает по построению ни с одной уже имеющейся вычислимой функцией,
2. сама определяется конструктивно и является вычислимой.

Значит, мы перечислили не все функции, следовательно, их бесконечное число. Таким образом, предположение о невычислимости $g(n)$ имеет смысл лишь при ограниченном количестве вычислимых функций, так что с данным примером невычислимости мы слегка промахнулись. Давайте построим хоть одну «честную» невычислимую функцию.

Это делается проще, чем кажется, в самом деле, гибрид функций из предыдущих примеров даст искомое:

$$h(n) = \begin{cases} f_n(n) + 1, & \text{если } f_n(n) \text{ определено} \\ 0, & \text{если } f_n(n) \text{ не определено} \end{cases}$$

Для доказательства предположим противное, т.е. $h(n) = f_p(n)$ при некотором p . Но этого не может быть, т.к. $h(n) \neq f_p(n)$ для любого определенного значения $f_p(n)$ по первому условию и $h(n) \neq f_p(n)$, а равно 0, даже если $f_p(n)$ не определено по второму условию. Отсюда же вытекает и существование недоопределяемых (для любых n) довычисляемых функций.

Ограничив немного поле действий, и уяснив, что вычислимыми бывают не все функции, перейдем к решению классических задач, естественным образом возникающих в связи с вычислимостью. Одной из них является порождение элементов некоторого множества, полученные элементы одновременно нумеруются в порядке появления. Множество называется разрешимым, если существует эффективная процедура выяснения принадлежности любого n этому множеству. Множество перечислимо, если оно есть область значения или область определения вычислимой функции. Эти два понятия связывает теорема Поста:

Для разрешимости множества X необходимо и достаточно, чтобы X и его дополнение $\bar{X}$ были перечислимы. При этом справедливо следующее утверждение:

Существует перечислимое, но неразрешимое множество. Убедиться в этом просто. Пусть $S_1, S_2, \dots$ эффективное перечисление всех перечислимых множеств (его существование следует из того, что это области значения вычислимых функций, чье множество перечислимо). Заметим, что если бы все S_i были разрешимы, то их дополнения входили бы в перечисление. Образует множество D из тех номеров n , которые принадле-

жат S_n . Таким образом, $n \in D \cap S_n$ эквивалентно $n \in D \cup S_n$, значит $D \cap S_n \neq \emptyset$, но при этом $D \cup S_n = \mathbb{N}$. Это означает, что D не совпадает ни с одним S_n и неразрешимо, а значит, по теореме Поста, и неперечислимо.

Аналогом разрешимого множества является понятие эффективно вычислимой функции, которая вычислима и определена при любом n . Наиболее интересной для нас моделью такой функции является конечный автомат, а точнее, его частный случай – машина Тьюринга. Ценность ее заключается в описании простейшей ЭВМ, которая может вычислить любую вычислимую функцию и тем самым реализовать любой рабочий алгоритм. Отметим, что машины Тьюринга, в силу конечности описания каждой, можно эффективно перечислить.

Рекурсивные функции

О машинах мы подробнее поговорим чуть ниже, сейчас же рассмотрим еще один важный (на сей раз уже для следующей теории – λ -исчисления) аппарат рекурсивных функций. Поначалу может показаться, что их введение вместо вычислимых ограничивает нас, урезая правила игры, на самом же деле эти подходы эквивалентны. Итак, на определенном этапе приходит мысль – нельзя ли выразить все вычислимые функции посредством какого-нибудь тривиального набора? Конечно можно, список таких функций, открытый Черчем, очень мал, это:

$0(x) = 0$ обнуление;
 $\sigma(x) = x + 1$ следование;
 $\theta_m^n(x_1, \dots, x_n) = x_m$ проектирование;
 $\omega(x_1, \dots, x_n) = \psi[\varphi_1(x_1, \dots, x_n), \dots, \varphi_m(x_1, \dots, x_n)]$ суперпозиция;
 $\begin{cases} f(x, 0) = g(x) \\ f(x, y + 1) = h(x, y, f(x, y)) \end{cases}$ примитивная рекурсия.

Функция называется примитивно рекурсивной, если она может быть получена с помощью указанных операций. Это функции «хорошие», даже слишком, т.е. всюду вычислимы и определены. Но у нас есть еще класс функций, которые могут и не быть вычислимыми для любого значения переменной. Проблеме помогает добавление оператора, который определен не всегда – минимизация:

$$\mu(\varphi(x, y)) = 0$$

Функции, полученные с добавлением минимизации, называются частично-рекурсивными (иногда их называют μ -функциями, по аналогии с λ -исчислением и π -исчислением, о которых написано ниже), и это в точности все возможные вычислимые функции. Задумайтесь, пройдя от функций обратно к алгоритмам и далее к программам. Это утверждение означает, что любая, вообще любая программа может быть написана на языке, включающем в себя только эти шесть операций, и их реализации в железе достаточно для получения ЭВМ, способной выполнить любую программу! Существуют и более экзотические системы функций, например, система Робинсона (1950 г.):

Любая одноместная общерекурсивная функция может быть получена из функций $\sigma(x) = x + 1$ и $\zeta(x) = x - \lfloor \sqrt{x} \rfloor^2$ с помощью конечного числа сложений, суперпозиций и всюду определенных обращений одноместных функций.

Наконец, тот факт, что вычислимые функции эффективно нумеруются, позволяет рассмотреть вместо $f_1(n), \dots, f_k(n), \dots$ функцию $U(n, x)$, называемую универсальной. Нумерация и функция U называются гёделевскими, если существует примитивно рекурсивная функция $s(n)$, такая, что для любой $f(n, x)$ справедливо $f(n, x) = U(s(n), x)$. Например, упоминавшееся уже упорядочивание программ по их длине дает гёделевскую нумерацию.

Допустим, мы отсеяли упомянутым выше способом, все, что не является программой в традиционном смысле слова и даже выловили все ошибки во всем, что осталось. Возникает вопрос – любая ли задача алгоритмически разрешима в принципе теми средствами, что у нас есть? Универсальные функции позволяют дать ответ на этот вопрос.

Заметим, что диагональная функция $u(n) = U(n, n)$ заведомо не определена при всех n и принципиально недоопределяема. Другими словами, существует n , при котором значение $f_n(n)$ не определено, т.е. n -я программа не применима сама к себе. Еще одна проблема заключается в том, что функция $g(x) = u(x) + 1$ не может быть везде доопределена. Следовательно, среди вычислимых есть функции, область определения которых не может быть расширена до $\mathbb{N}$; одним из следствий этого является неразрешимость проблемы останова. Эта фундаментальная проблема гласит – возможно ли создание алгоритма, который по поданному на его вход алгоритму определит – остановится тот или нет. Иными словами, невозможно написать программу, которая автоматически тестирует любые другие программы на ошибки. Бесконечный список алгоритмически неразрешимых проблем укладывается в одно предложение, называемое теорема Райса:

Ни одно нетривиальное свойство вычислимых функций алгоритмически неразрешимо.

Под нетривиальностью здесь понимают, что свойство не является общезначимым, т.е. есть функции, которые им обладают, и есть – которые не обладают. Например, неразрешимы проблемы закливания произвольного алгоритма, конечности корней уравнения, периодичности, ограниченности, порядка роста и т.п.

Но даже с разрешимыми свойствами иногда возникают колоссальные проблемы, касающиеся т.н. вычислительной сложности. Очевидно, что одни задачи решаются легко, другие сложно. Почему так происходит, и можно ли с этим что-нибудь поделать? Дискретные задачи часто имеют комбинаторную природу, например, в классической задаче коммивояжера даны n городов и r_{ij} расстояний между ними. Всего существует X возможных гамильтоновых циклов, т.е. замкнутых маршрутов, проходящих по одному разу через все города; если дана стоимость перемещения между ними, обычно требуется найти самый дешевый, если нет – просто кратчайший. Уже при n порядка нескольких десятков, возможных маршрутов оказывается больше, чем атомов в видимой Вселенной, и отыскать среди них перебором нужный за разумное время невозможно. Существуют ли алгоритмы, для решения таких задач, кроме перебора? Приближенные – да, точные – не известно и вряд ли будет известно в ближайшем будущем.

Аналогичной сложности задача факторизации чисел – основная теорема теории чисел утверждает, что любое целое число либо простое, либо однозначно разлагается на простые сомножители. Но вот найти это разложение для действительно больших чисел – та еще задачка. Существуют определенные тесты, позволяющие узнавать, простое перед нами число или нет, но точных и быстрых алгоритмов факторизации не найдено, на чем основывается криптостойкость систем с открытым ключом.

Время, затраченное на решение таких задач, с ростом n растет экспоненциально²⁷ и задачи называются, соответственно, экспоненциальными. Конечно, экспоненциально время растет только в случае алгоритма перебора, но вот здесь-то и кроется проблема – существуют ли не экспоненциальные алгоритмы решения таких задач? Пока этого никто не знает.

Не все комбинаторные задачи такие скверные. Скажем, классическая задача поиска минимального остовного дерева (МОД-задача) из теории графов (содержательно, например, соответствует задаче – построить самую дешевую сеть дорог, связывающих n населенных пунктов) тривиально решается т.н. жадным алгоритмом. Выбираем произвольный город и сначала строим из него самую дешевую дорогу из всех возможных, затем из второго города, куда она ведет, строится тоже самая дешевая, при условии, что она не образует цикла, и так повторяется до победы. Если равно дешевых дорог несколько – строим произвольную из них, опять же, при условии отсутствия цикла. Вот и все решение!

В результате, теория сложности алгоритмов рассматривает две проблемы: во-первых, скорее философскую, – выяснение принципиальных ограничений на скорость решения тех или иных классов задач, во-вторых, практическую – поиск оптимальных алгоритмов для уже решенных. Еще один классический пример, напрямую связанный с архитектурой ЭВМ – быстрое умножение. Умножение вообще весьма ресурсоемкая процедура, хуже только деление, к тому же в неоптимальном варианте оно требует большое количество функциональных блоков. Заставить машину работать по оптимальному алгоритму – значит, не только ускорить ресурсоемкие процедуры, но удешевить конструкцию и сделать ее надежнее.

При умножении чисел n_1 и n_2 плохой алгоритм просто $n_1 - 1$ раз «в лоб» выполнит рекурсивную операцию $N(k+1) = N(k) + n_2$, положив $N(1) = n_2$. В итоге $N(n_1) = n_1 \cdot n_2$, но время работы пропорционально n_1 и это уже никуда не годится. Оснатив машину не только сумматором, но и устройством сдвига, можно реализовать умножение в столбик, которое работает за время, пропорциональное $\log n_1 \log n_2$, а это уже намного лучше. Умножение же методом Карацубы работает еще быстрее и т.д.

В качестве стандартных терминов теории сложности приняты – длина входа, или длина описания задачи в битах, т.е. количество бит, необходимое для записи условия и эталонная сложность – полиномиальная. Если работа алгоритма с длиной входа x характеризуется необходимостью выполнения $f(x)$ элементарных операций, то алгоритм считается полиномиальным (а задача – хорошо решаемой), если $f(x) = O(x^k)$ при некотором $k > 0$. Иначе говоря, для большого x существует такая $C = const$, что $f(x) < C \cdot x^k$. Естественно, любая комбинация таких алгоритмов дает тоже полиномиальный алгоритм.

Задачи теории сложности обычно принимают две формы – задачи оптимизации, для которых нужно подобрать решение, и т.н. задачи распознавания – которые по предлагаемым решениям возвращают ответ – оптимально оно или нет. Если полиномиальный алгоритм существует для одной такой задачи, то он существует и для второй, что формулируется в виде теоремы:

Если целевая функция может принимать не более N значений, то существует дихотомическая процедура решения оптимизационной задачи за $\log_2 N$ шагов решения соответствующей комбинаторной задачи распознавания.

Однако, это тоже помогает не всегда. Например, в задаче «простое число – составное число» обе половинки полиномиально разрешимы, как задача распознавания, но задача факторизации – экспоненциальна. Проблема заключается в том, что при поиске сомножителей не ясно пока, как можно контролируемо уменьшать размер задачи, что бы это помогало решению исходной.

²⁷ На самом деле в комбинаторике число комбинаций определяется через факториал, соответственно, количество вариантов (вместе со временем перебора) растет факториально, но т.к. нас интересуют большие n , то факториал по формуле Стирлинга обычно приближают экспоненциальной функцией для простоты работы: для больших n , $n! \approx \sqrt{n} \cdot \left(\frac{n}{e}\right)^n$.

Совокупность задач распознавания, которые могут быть решены полиномиальным алгоритмом носит название класса P или просто P -задач. Класс NP -задач – это задачи распознавания, полиномиально проверяемые, они описываются несколько сложнее.

Класс NP определяется для множества языков, то есть множеств слов над конечным алфавитом Σ . Язык $L \in NP$, если существуют двуместный предикат²⁸ $R(x, y) \in P$ (то есть вычислимый за полиномиальное время) и константа $C > 0$, такие, что для всякого слова x условие $x \in L$ равносильно условию $\exists y, |y| < |x|^C$, такой, что $R(x, y) = 1$. Слово y называется сертификатом принадлежности x к L .

Таким образом, если у нас есть слово x , относительно принадлежности которого к L мы не уверены, и слово-сертификат полиномиальной длины, принадлежащее L (его бывает трудно найти), то мы за полиномиальное время можем проверить принадлежит ли x L или нет.

В переводе на человеческий язык это означает, что если у нас есть решение, относительно которого мы не знаем, правильно оно или нет, и есть, в каком-то смысле, «частное решение» (полиномиально не длиннее данного), которое точно правильно, то, используя его, мы за полиномиальное время проверим – правильно данное решение или нет.

Таким образом, класс P – это задачи распознавания, которые, например, машина Тьюринга может решить за полиномиальное время, а класс NP – решение которых, при наличии некоторых дополнительных сведений, можно полиномиально проверить на машине Тьюринга²⁹. Отметим, что недетерминированная машина (иначе говоря, вероятностный алгоритм, который не дает точного решения, но может быть к нему сколь угодно близок) решает NP полиномиально без сертификатов. Всякую NP задачу можно решить точно, полным перебором за экспоненциальное время.

Основная ценность квантов компьютера заключается в том, что сложность квантов алгоритмов является корнем квадратным из классической сложности. Соответственно на такой машине (применив т.н. алгоритм Гровера, GSA) можно решать полным перебором NP задачи за время $t_Q = \sqrt{T}$, где T – время классического решения, что является невероятным ускорением. GSA есть пример т.н. массовой задачи, зависящей от оракула. Для более частных задач удастся получить большее квантовое ускорение. Например, алгоритм факторизации Шора дает экспоненциальный выигрыш по сравнению с соответствующими классическими алгоритмами.

Класс языков, дополнения которых принадлежат NP , называется классом $co - NP$, хотя и не доказано, что этот класс отличен от класса NP . Пересечение классов NP и $co - NP$ содержит класс P . В частности, класс NP включает в себя P , однако не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов. Если они равны, то любую задачу из класса NP можно будет решить быстро (за полиномиальное время). Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос. Существует более узкий класс NP -полных задач. Задача NP полна, если к ней за полиномиальное время сводится любая другая NP -задача. Для многих задач доказано, что они NP -полны, соответственно, если удастся решить хоть одну из них за полиномиальное время, удастся решить их все. Сами NP задачи еще не самые сложные, они включаются в иные, более широкие, например, в класс PH . Существуют также открытые вопросы о строгости его включения в другие классы.

1.5.3. Конечные автоматы

Одна из наиболее ценных абстракций теории алгоритмов носит название конечный автомат. Он определяется следующим образом. Пусть даны два множества – $A = \{a_1, \dots, a_n\}$ и $B = \{b_1, \dots, b_n\}$, называемые, соответственно, входной и выходной алфавит. Определим множества A^∞ и B^∞ как множество всех последовательностей в этих алфавитах.

Отображение $\varphi: A^\infty \rightarrow B^\infty$ назовем детерминированной функцией, если для любого $t \in \mathbb{N}$, $b(t)$ однозначно определяется по $a(1), \dots, a(t)$. Обозначают такие функции так:

$$\begin{array}{l} \varphi: \begin{array}{l} a_1(1), \dots, a_1(t) \rightarrow b_1(1), \dots, b_1(t) \\ \vdots \\ a_n(1), \dots, a_n(t) \rightarrow b_n(1), \dots, b_n(t) \end{array} \end{array}$$

причем если $a_1(1) = a_2(1)$, то $b_1(1) = b_2(1)$ и если $a_1(i) = a_2(i) \forall i \in (1, \dots, t)$, то $b_1(t) = b_2(t)$.

²⁸ n -местный предикат R – это функция с множеством значений $\{0,1\}$, определенная на множестве $M = M_1 \times \dots \times M_n$. Таким образом, $R(m_1, \dots, m_n) = \{0,1\}$. Например: пусть $M = \mathbb{N}^n$ и дано отношение упорядоченности по возрастанию. Тогда предикат, проверяющий это отношение, вернет 0, если оно не выполняется, и 1, если выполняется – $R(5,1,2,3,4, \dots, n) = 0$, а $R(1,2,3,4,5, \dots, n) = 1$.

²⁹ Что не избавляет нас от необходимости откуда-то его взять сначала, что, в общем случае, может быть той еще проблемой!

Пусть у нас имеется функция $\varphi: A^\infty \rightarrow B^\infty$. Рассмотрим произвольное слово $a = a_1 a_2 \dots a_k \in A^*$. Определим функцию φ_a следующим образом. Пусть $a(1), \dots, a(t), \dots$ – произвольная входная последовательность. Рассмотрим

$$\varphi(a_1 a_2, \dots, a_k a(1) a(2) \dots a(t) \dots) = b_1 b_2, \dots, b_k b(1) b(2) \dots b(t) \dots$$

Положим $\varphi_a(a(1) a(2) \dots a(t) \dots) = b(1) b(2) \dots b(t) \dots$ и назовем ее остаточной функцией для φ по слову a . Детерминированная функция φ называется ограниченно детерминированной, если у нее имеется лишь конечное число различных остаточных функций.

Автоматом, точнее инициальным автоматом, называется любая шестерка (A, B, Q, G, F, q_0) , где A, B, Q – конечные алфавиты, $G: A \times Q \rightarrow Q$, $F: A \times Q \rightarrow B$, $q_0 \in Q$ – начальное состояние. Неинициальные автоматы могут начинать свою работу с произвольного состояния и произвольным заканчивать.

Входом такого автомата служит последовательность $a_1 a_2 \dots a_k \in A^*$, не обязательно конечная, выходом автомата служит последовательность $z(t)$, при этом автомат задается системой уравнений:

$$\begin{cases} z(t) = F(x(t), q(t-1)) \\ q(t) = G(x(t), q(t-1)) \\ q(0) = q_0 \end{cases}$$

Отображение $\varphi: A^\infty \rightarrow B^\infty$ называется автоматной функцией, если существует автомат, реализующий это отображение. Функция является автоматной тогда и только тогда, когда она ограниченно детерминирована. Например, пусть $A = B = Q = \{0, 1\}$ и система канонических уравнений выглядит так:

$$\begin{cases} z(t) = q(t-1) \\ q(t) = x(t) \\ q(0) = 0 \end{cases}$$

Такой автомат осуществляет отображение $a(1) a(2) \dots \rightarrow 0 a(1) a(2) \dots$ и называется единичной задержкой. Очевидно, таким образом можно описать формально, с одной стороны, все вычислимые функции, т.е. любой реализуемый алгоритм, с другой – конкретный преобразователь, который его выполняет. Любой процессор является сложным конечным автоматом.

Диаграммой Мура для автомата называется ориентированный граф с множеством вершин Q , у которой каждой паре (a, q) сопоставляется дуга, идущая из вершины q в соответствующую $G(a, q)$. Этой дуге приписывается метка $(a, F(a, q))$. Особо помечается вершина, соответствующая начальному состоянию. Диаграмма Мура однозначно задает автомат.

Абстрактный автомат с выделенным начальным состоянием называется инициальным. Если функция переходов и/или функция выходов являются случайными, то автомат называют вероятностным. Ограничение числа параметров абстрактного автомата определило такое понятие как конечный автомат.

Классификация автоматов по логическим свойствам функций переходов и выходов. В специальной литературе автомат, все действия которого привязаны к переходам, называется машиной Мили (Mealy), а автомат, все действия которого привязаны к состояниям, машиной Мура (Moore).

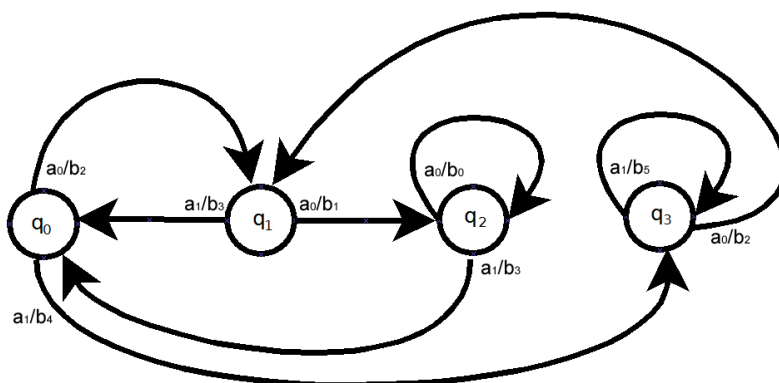
В автомате Мили функция выходов определяет значение выходного символа по классической схеме абстрактного автомата. Математическая модель автомата Мили и схема рекуррентных соотношений не отличаются от математической модели и схемы рекуррентных соотношений абстрактного автомата. Автомат Мура описывается немного модифицированным способом:

$$\begin{cases} z(t) = F(x(t), q(t-1)) \\ q(t) = G(x(t)) \\ q(0) = q_0 \end{cases}$$

Как видим, выходной сигнал автомата Мура зависит только от состояния и не зависит явно от входного сигнала. Работа автомата состоит в том, что он анализирует состояния своих входов и, в зависимости от значений входов и своего внутреннего состояния, изменяет значения выходов и внутреннее состояние. Правила, в соответствии с которыми происходит изменение, описываются таблицей или диаграммой переходов. Диаграмма переходов представляет собой граф, вершины которого соответствуют допустимым состояниям внутренних переменных автомата, а ребра – допустимым переходам между ними. Переходы между вершинами в общем случае некоммутативны: наличие перехода из A в B не означает, что существует переход из B в A . Наличие перехода в обоих направлениях символизируется двумя ребрами, соединяющими одну пару вершин. Таблица переходов может рассматриваться как матричное представление диаграммы переходов.

У конечных автоматов с ограниченным числом допустимых значений входов граф переходов всегда конечен, хотя и может содержать циклы (замкнутые пути) и контуры (совокупности различных путей, приводящих к одной и той же вершине). Автомат заканчивает свою работу, если достигнуто одно из конечных состояний, прочитан символ, не принадлежащий A^* , или входные данные исчерпаны.

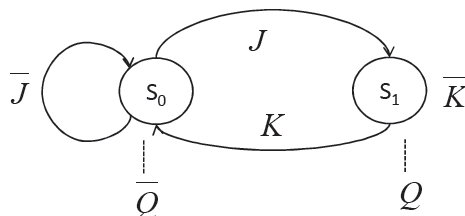
Классический пример работы автомата – допустим, методы выставления оценок, когда текущая оценка ставится исходя из предыстории событий.



Например, у нас есть входной алфавит с 2 событиями: $A = \{a_0, a_1\}$, где a_0 соответствует прогулу, а a_1 соответствует посещению студентом занятия, и выходной алфавит $B = \{b_0, b_1, b_2, b_3, b_4, b_5\}$ с соответствующими реакциями:

- b_0 – отправить к декану;
- b_1 – назначить отработку прогула;
- b_2 – уточнить причины отсутствия;
- b_3 – не реагировать;
- b_4 – похвалить за посещаемость;
- b_5 – поставить зачет автоматом.

В результате, студента, прогулявшего пару, ожидает различная реакция, в зависимости от предыстории его поступков. Далее приведен автомат, реализующий такую простую вещь, как JK-триггер, одну из разновидностей простых ячеек памяти:



Здесь (для соответствия традиционным обозначениям триггеров) внутренние состояния (записано 0 или 1) обозначаются, соответственно, s_0 и s_1 , выходы Q и $\bar{Q}$, а управляющие сигналы – $J, \bar{J}, K, \bar{K}$.

Как уже упоминалось в разделе секвенциальной логики, триггер можно задать соответствующей формулой или таблицей. RS-триггер (два входа – Set/Установка, при подаче на него 1 на выходе возникает 1 и Reset/Сброс, при подаче на него 1 на выходе возникает 0, в результате триггер никак не реагирует на нулевые входы, сохраняя последнее записанное состояние, или меняет записанное на противоположное при подаче на один из входов единицы). Такое устройство описывается формулой $\bar{x} \vee x \wedge \bar{y}$ или, в аналитической записи, состояние выхода Q есть функция входов R и S : $Q = \bar{S} \vee S \wedge \bar{R}$. Состояние, при котором на оба входа R и S одновременно поданы логические единицы, в простейших реализациях является запрещённым, так как вводит схему в режим генерации.

JK-триггер работает так же, как RS-триггер, с одним лишь исключением: при подаче логической единицы на оба входа J и K состояние выхода триггера изменяется на противоположное. Вход J (Jump) аналогичен входу S . Вход K (Kill) аналогичен входу R . При подаче единицы на вход J и нуля на вход K выходное состояние триггера становится равным логической единице, а при подаче единицы на вход K и нуля на вход J выходное состояние триггера становится равным логическому нулю. JK-триггер в отличие от RS-триггера не имеет запрещённых состояний на основных входах. Такой триггер описывается формулой

$$Q(t+1) = \bar{Q}(t) \cdot J + Q(t) \cdot \bar{K}$$

1.5.4. Машины Тьюринга, Поста, фон Неймана

Самый знаменитый из всех абстрактных автоматов был предложен Аланом Тьюрингом в 1936 г. в качестве одного из первых формальных описаний алгоритма. Модель этой машины при желании даже можно построить или запрограммировать, что приближает ее к реальной ЭВМ. В этом смысле машина Тьюринга очень важна, так как отвечает на вопрос – реализуема ли на данной архитектуре данная программа? Так как конечный автомат в состоянии выполнить любой рабочий алгоритм, достаточно показать, что данная ЭВМ формально эквивалентна машине Тьюринга. Из этого сразу следует, что она может выполнить любую выполнимую программу, а значит, является универсальной. Критерий равносильности конкретной ЭВМ данной машины носит название полноты по Тьюрингу.

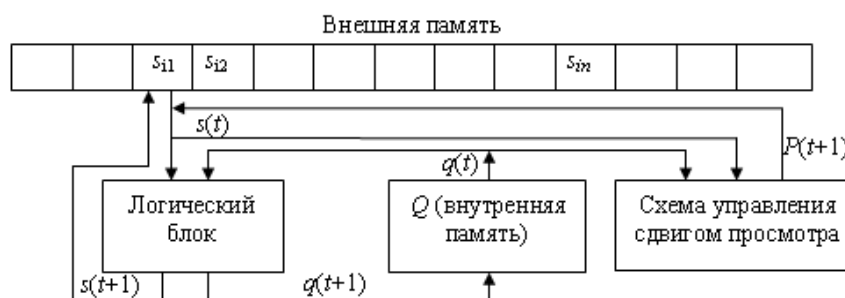
В общем случае не любая ЭВМ обладает свойством полноты, например, один из первых компьютеров – ABC таковым не являлся, полнота по Тьюрингу аналитической машины Беббиджа была доказана лишь в 1970-е годы.

Итак, как работает этот автомат?

В каждой машине Тьюринга есть две основные части:

- Внешняя память – неограниченная в обе стороны лента (в реальных машинах частично эмулируется теоретически неограниченным расширением памяти через добавление новых блоков), разделенная на ячейки.
- Читающая головка (и блок управления с внутренней памятью), которая в дискретный момент времени может обозревать одну ячейку и записывать/считывать информацию из нее, в зависимости от управляющих сигналов.

Вот одна из схем такой машины:



С каждой машиной Тьюринга связаны два конечных алфавита: алфавит входных символов $S = \{s_{i1}, \dots, s_{in}\}$ и алфавит состояний $Q = \{q_1, \dots, q_n\}$. Среди символов имеется пустой, обозначаемый s_0 , запись которого стирает содержимое текущей ячейки. Среди состояний есть два выделенных – q_B и q_E , соответственно начальное и конечное.

В зависимости от поданной начальной информации a (слово, содержащееся на ленте внешней памяти знаков) возможны два случая:

- после конечного числа тактов машина останавливается (имея информацию b), подавая сигнал об остановке. В этом случае МТ применима к информации a и перерабатывает ее в информацию b ;
- остановка никогда не наступает. В этом случае МТ не применима к начальной информации a .

В каждый момент обозревается лишь одна ячейка ленты. Переход может осуществляться лишь к соседней ячейке (R – вправо, L – влево, N – остановится). Переход к произвольной ячейке производится путем последовательного перебора всех ячеек, разделяющих текущую и необходимую ячейки. На каждом отдельном такте t команда предписывает только замену единственного знака s_i , хранящегося в обозреваемой ячейке, каким-либо другим знаком s_j . Знаки (R, L, N, Q) образуют внутренний алфавит машины.

Переработанный знак s_j , записываемый в просматриваемую ячейку, состояние, которое примет машина Тьюринга в следующем такте $q(t+1)$, и выполняемая в данном такте операция перехода к следующей ячейке $P(t+1)$ являются функцией анализируемого в данном такте символа и текущего состояния машины s_i и $q(t)$:

$$\begin{cases} s_i(t+1) = F_1(s_i, q(t)) \\ q(t+1) = F_2(s_i, q(t)) \\ P(t+1) = F_3(s_i, q(t)) \end{cases}$$

Любая программа для МТ определяется тройкой $\{s_i, P, q\}_t$.

Вот пример вычисления логической функции $y = y \oplus x$ для машины Тьюринга:

Символ S_i	Состояние			
	q_1	q_2	q_3	q_4
0	$\{0, R, q_2\}$	$\{0, N, q_4\}$	$\{1, N, q_4\}$	$\{0, N, q_4\}$
1	$\{1, R, q_3\}$	$\{1, N, q_4\}$	$\{0, N, q_4\}$	$\{1, N, q_4\}$

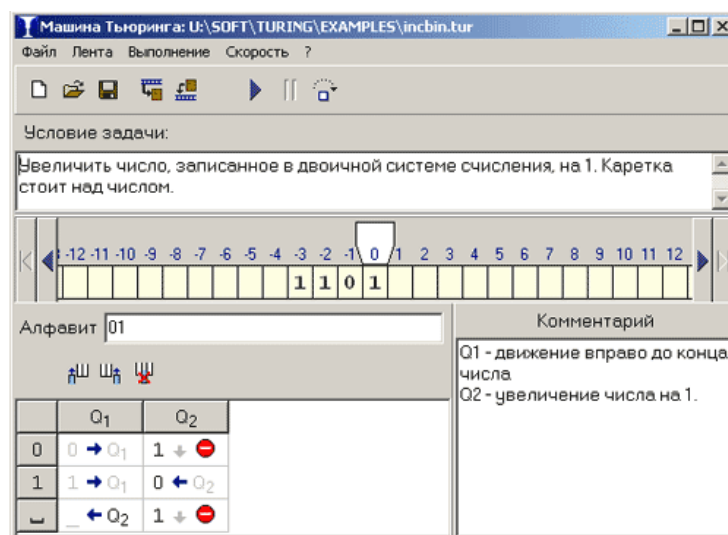
Перед началом работы машина Тьюринга находится в состоянии q_1 считывания первого операнда. Данная МТ применима к исходной информации. Останов – состояние q_4 . Значение S_j в ячейке y не меняется (сохраняется результат).

Если программа для МТ будет определена таблицей переходов:

Символ S_i	Состояние			
	q_1	q_2	q_3	q_4
0	$\{0, R, q_2\}$	$\{0, N, q_4\}$	$\{1, N, q_4\}$	$\{1, N, q_4\}$
1	$\{1, R, q_3\}$	$\{1, N, q_4\}$	$\{0, N, q_4\}$	$\{0, N, q_4\}$

то данная МТ будет не применима к исходной информации, поскольку в состоянии q_4 значение S_j в ячейке y постоянно меняется на противоположное.

Вот пример одного из простых программных эмуляторов МТ:



Модель машины Тьюринга допускает расширения. Можно рассматривать машины Тьюринга с произвольным числом лент и многомерными лентами с различными ограничениями. Однако все эти машины являются полными по Тьюрингу и моделируются обычной машиной.

В случае недетерминированной машины Тьюринга (НМТ), комбинация текущего состояния автомата и символа на ленте может допускать несколько переходов. Например, X на ленте и состояние 3 допускает как состояние 4 с записью на ленту символа Y и смещением головки вправо, так и состояние 5 с записью на ленту символа Z и смещением головки влево.

Как НМТ «узнаёт», какой из возможных путей приведёт в допускающее состояние? Есть два способа это представить.

- Можно считать, что НМТ — «чрезвычайно удачлива»; то есть всегда выбирает переход, который, в конечном счёте, приводит к допускающему состоянию, если такой переход вообще есть.
- Можно представить, что в случае неоднозначности перехода (текущая комбинация состояния и символа на ленте допускает несколько переходов) НМТ делится на копии, каждая из которых следует за одним из возможных переходов.

То есть, в отличие от ДМТ, которая имеет единственный путь вычислений, НМТ имеет дерево вычислений (в общем случае — экспоненциальное число путей). Говорят, что НМТ допускает входные данные, если какая-нибудь ветвь этого дерева останавливается в допускающем состоянии, иначе НМТ входные данные не допускает. (Таким образом, ответы «ДА» и «НЕТ» в случае недетерминированных вычислений не симметричны).

Интуитивно кажется, что НМТ более мощные, чем ДМТ, так как они выполняют несколько возможных вычислений сразу, требуя только, чтобы хотя бы одно из них заканчивалось в допускающем состоянии. Однако любой язык, допускающийся НМТ, также допускается ДМТ: ДМТ может моделировать любой переход НМТ, делая многократные копии состояния, если встречается неоднозначность.

Очевидно, что это моделирование требует значительно больше времени. Насколько больше — не известно. В частном случае ограничения по времени в виде полинома от длины входа этот вопрос представляет собой классическую задачу $P = NP$. Класс алгоритмов, выполняемых за полиномиальное время на недетерминированных машинах Тьюринга, является классом NP.

Универсальной машиной Тьюринга называют МТ, которая может заменить собой любую машину Тьюринга. Получив на вход программу и входные данные, она вычисляет ответ, который вычислила бы по входным данным машина Тьюринга, чья программа была дана на вход.

Теорема об универсальной машине Тьюринга утверждает, что такая машина существует и моделирует другие машины с не более чем квадратичным замедлением.

Машина Поста

Существует более простой вариант МТ, предложенный независимо Эмилем Постом для тех же целей. Машина Поста состоит из каретки (или считывающей и записывающей головки) и разбитой на секции бесконечной в обе стороны ленты. Каждая секция ленты может быть либо пустой — 0, либо помеченной меткой 1. За один шаг каретка может сдвинуться на одну позицию влево или вправо, считать, поставить или стереть символ в том месте, где она стоит. Работа машины Поста определяется программой, состоящей из конечного числа строк. Для работы машины нужно задать программу и ее начальное состояние (т.е. состояние ленты и позицию каретки). Кареткой управляет программа, состоящая из строк команд. Каждая команда имеет следующий синтаксис:

$$\langle i K j \rangle$$

где i — номер команды, K — действие, j — номер следующей команды. Всего для машины Поста существует шесть типов команд:

$V j$	поставить метку, перейти к j -й строке
$X j$	стереть метку, перейти к j -й строке
$\leftarrow j$	сдвинуться влево, перейти к j -й строке
$\rightarrow j$	сдвинуться вправо, перейти к j -й строке
$? j_1 j_2$	если в ячейке нет метки, то перейти к j_1 -й строке, иначе перейти к j_2 -й строке
!	стоп

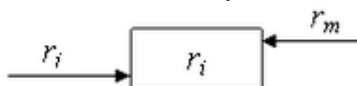
У команды «стоп» отсылки нет. После запуска возможны варианты:

- работа может закончиться невыполнимой командой (стирание несуществующей метки или запись в помеченное поле);
- работа может закончиться командой Stop;
- работа никогда не закончится.

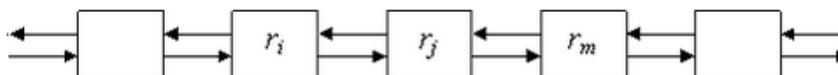
Машина фон Неймана

По принципу обработки информации вычислительное устройство, предложенное Нейманом (автомат Неймана — АН), существенно отличается от машины Тьюринга. Важная особенность машины Тьюринга — преобразование информации на каждом такте происходит лишь в одной ячейке, остальные ждут посещения головки, хотя часто имеется возможность работать параллельно. Простейшее решение — использование нескольких машин Тьюринга с общей для них внешней памятью (лентой) — не всегда допустимо из-за возможных конфликтов при обращении к одной и той же ячейке памяти.

В автомате Неймана число одновременно обрабатываемых ячеек может неограниченно расти, оставаясь в каждый момент конечным. Элемент Неймана (ЭН) — это устройство, которое на каждом такте пребывает в одном из конечного числа состояний $r_i \in R$, образующих его алфавит. ЭН имеет два входных канала: левый и правый; по каждому из них на такте t также поступает по одному состоянию из R :



Элемент реализует функцию $z_{t+1} = \psi(r_i, r_j, r_m)_t$, то есть в такте $t + 1$ переходит в состояние z , определяемое его состоянием в текущий момент времени и значениями, поступившими по входным каналам. Состояния элементов Неймана в момент времени t определяют конфигурацию автомата Неймана $K(t)$ в момент $t + 1$:



Функционирование АН — это переход от состояния $K(t)$ к состояниям $K(t + 1)$, $K(t + 2)$... За один такт свое состояние может менять большое число элементов Неймана, что фактически приводит к параллельной обработке информации.

Регистровые машины

Регистровые машины были разработаны для того, чтобы описывать алгоритмы более близкими к реальным ЭВМ способами. Среди них известны РАМ-машина, машина Минского, машина Шепардсона и Стургиса и многие другие.

РАМ-машина (равнодоступная адресная машина) – модель машины с одним сумматором, команды программы не могут изменять сами себя. Она состоит из входной ленты, с которой она может только считывать, и исходной ленты, на которую она может только записывать, и памяти. Входная лента состоит из последовательности ячеек, в которых записаны целые числа (возможно отрицательные). Каждый раз, когда машина считывает число с входной ленты, головка передвигается на следующую ячейку вправо.

На исходную ленту машина может только записывать; она разбита на ячейки, которые первоначально пусты. При выполнении команды записи в ячейку, на которую указывает записывающая головка, сохраняется целое число, а головка передвигается на следующую ячейку вправо. Записанное исходное число изменить уже невозможно.

Память состоит из последовательности регистров $r_0, \dots, r_i, \dots$, каждый из которых может хранить произвольное целое число. Программа РАМ-машины хранится не в её памяти, поэтому предполагают, что программа не способна изменять саму себя. Программа состоит из последовательности (возможно) помеченных команд. Список команд зависит от постановки задачи, но похож на типичный язык ассемблера. Вычисления осуществляют в первом регистре — r_0 , который называют сумматором. Каждая команда состоит из двух частей: кода операции и адреса.

Вот типичный набор команд РАМ-машины:

- HALT самая простая инструкция, приказывающая машине остановить выполнение по ее достижении.
- READ число вводится в регистр r_0 .
- WRITE выводит число, хранящееся в r_0 .
- LOAD $x / [x] / [[x]]$ записывает в r_0 произвольное значение. Значение может быть задано константно: LOAD 5, например. Или скопировано из другой ячейки: LOAD [3] — здесь значение из третьего регистра копируется в нулевой, или же с помощью двойной адресации: LOAD [[3]] загружается число из регистра, на который ссылается третий регистр. Таким образом, видно, что машина поддерживает косвенную адресацию.

- STORE $x / [x] / [[x]]$ записывает значение из r_0 в заданную ячейку. Доступна либо прямая адресация: STORE [1], либо косвенная: STORE [[1]].

- NEG обращает знак числа в r_0 .
- LSHIFT $x / [x] / [[x]]$ выполняет побитовый сдвиг влево числа в r_0 на указанный аргумент; доступна константа, прямой адрес, косвенный адрес. Например, LSHIFT [[3]] умножит значение в [0] на двойку в степени значения регистра, на который ссылается [3].

- RSHIFT $x / [x] / [[x]]$ выполняет побитовый правый сдвиг, аргументы и смысл аналогичны LSHIFT. Классическая модель не допускает использование знаковых чисел в качестве аргументов LSHIFT/RSHIFT.

- ADD $x / [x] / [[x]]$ прибавляет к r_0 значение, указанное аргументом функции. Доступна константа, прямая адресация, косвенная.

- JUMP метка безусловный переход на указанную строку кода или метку.

- JG метка переход на указанную метку, если значение в нулевом регистре строго положительное.

Машина Минского работает почти так же, но имеет другой набор команд. Все инструкции машины снабжены метками и бывают трех типов:

- $Sk: Rl++; Si$ увеличить значение в регистре Rl на 1, перейти к команде с меткой Si ;
- $Sk: Rl--; Si; Sj$, если в регистре Rl не 0, уменьшить его значение на 1 и перейти к команде с меткой Si , иначе перейти к команде с меткой Sj ;
- $Sk: STOP$ конец работы.

1.5.5. Сети Петри и клеточные автоматы

Сети Петри

На стыке конечных автоматов и процессных алгебр (о них ниже) находится такая замечательная вещь, как сети Петри. Эта абстракция, в отличие от вышеописанных, была разработана специально для описания параллельных процессов, для которых машины Тьюринга и фон Неймана не очень удобно использовать.

Данный математический аппарат моделирования динамических дискретных систем появился совсем недавно – сети Петри были предложены в докторской диссертации «Kommunikation mit Automaten»³⁰ немецкого математика Карла Петри в 1962 г., что забавно – общую идею он придумал еще в 13 лет для описания процессов, протекающих в домашних химических опытах. Сети Петри на самом деле можно применять, как и любое другое описание конечного автомата, не только для описания работы ЭВМ, но и для описания логики работы программ, потока работ в менеджменте, химических реакций и др.

Математически сеть Петри определяется как четверка $\{P, T, I, O\}$, где P и T – конечные множества позиций и переходов, I и O – множества входных и выходных функций. Этот набор естественно представить в виде двудольного ориентированного мультиграфа. На графе Петри позициям соответствуют вершины, изображаемые кружками, а переходам – вершины, изображаемые утолщенными черточками; функциям I соответствуют дуги, направленные от позиций к переходам, а функциям O – дуги, направленные от переходов к позициям. Вершины одного типа не могут быть соединены непосредственно.

Как и в системах массового обслуживания, в сетях Петри вводятся объекты двух типов: динамические, которые изображаются метками (маркерами) внутри позиций, и статические, которым соответствуют вершины сети Петри.

Распределение маркеров по позициям называют маркировкой. Маркеры могут перемещаться в сети. Каждое изменение маркировки называют событием, причем каждое событие связано с определенным переходом. Считается, что события происходят мгновенно и одновременно при выполнении некоторых условий.

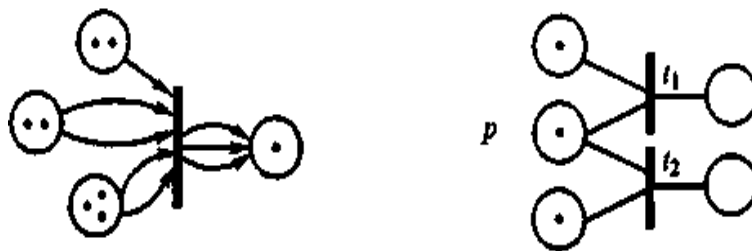
Каждому условию в сети Петри соответствует определенная позиция. Совершению события соответствует срабатывание (возбуждение или запуск) перехода, при котором маркеры из входных позиций этого перехода перемещаются в выходные позиции. Последовательность событий образует моделируемый процесс.

Правила срабатывания переходов конкретизируют следующим образом: переход срабатывает, если для каждой из его входных позиций выполняется условие

$$N_i > K_i,$$

где N_i – число маркеров в i -й входной позиции, K_i – число дуг, идущих от i -й позиции к переходу. При срабатывании перехода число маркеров в i -й входной позиции уменьшается на K_i , а в j -й выходной позиции увеличивается на M_j , где M_j – число дуг, связывающих переход с j -й позицией.

На рис. показан пример распределения маркеров по позициям перед срабатыванием, эту маркировку записывают в виде $(2, 2, 3, 1)$ или (2231) . После срабатывания перехода маркировка принимает вид $(1, 0, 1, 4)$.



Фрагмент сети Петри и конфликтная ситуация

Можно вводить ряд дополнительных правил и условий в алгоритмы моделирования, получая ту или иную разновидность сетей Петри. Так, прежде всего полезно ввести модельное время, чтобы моделировать не только последовательность событий, но и их привязку ко времени. Это осуществляется приданием переходам веса – продолжительности (задержки) срабатывания, которую можно определять, используя задаваемый при этом алгоритм. Полученную модель называют временной сетью Петри.

Если задержки являются случайными величинами, то сеть называют стохастической. В стохастических сетях возможно введение вероятностей срабатывания возбужденных переходов. Так, выше представлен фрагмент сети Петри, иллюстрирующий конфликтную ситуацию – маркер в позиции p может запустить либо переход t_1 , либо переход t_2 . В стохастической сети предусматривается вероятностный выбор срабатывающего перехода в таких ситуациях.

³⁰ «Связь автоматов», в ней он сформулировал основные понятия теории связи асинхронных компонент вычислительной системы.

Если задержки определяются как функции некоторых аргументов, которыми могут быть количество маркеров в каких-либо позициях, состояния некоторых переходов и т.п., то сеть называют функциональной.

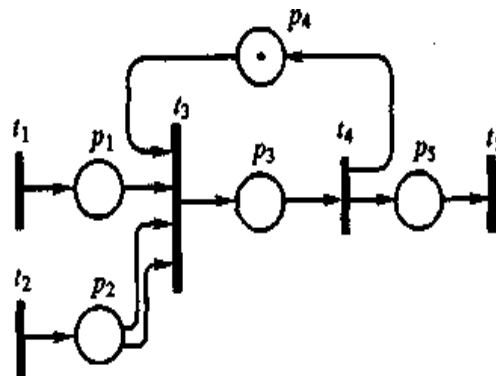
Во многих задачах динамические объекты могут быть нескольких типов, и для каждого типа нужно вводить свои алгоритмы поведения в сети. В этом случае каждый маркер должен иметь хотя бы один параметр, обозначающий тип маркера. Такой параметр обычно называют цветом; цвет можно использовать как аргумент в функциональных сетях. Сеть Петри при этом называют цветной.

Среди других разновидностей сетей Петри следует упомянуть ингибиторные сети, характеризующиеся тем, что в них возможны запрещающие (ингибиторные) дуги. Наличие маркера во входной позиции, связанной с переходом ингибиторной дугой, означает запрещение срабатывания перехода.

Как уже упоминалось, сети Петри применяют не только для описания работы ЭВМ, но и для более тривиальных вещей. Например.

Допустим, требуется описать с помощью сети Петри функционирование системы из предприятий A , B и C . Предприятия A и B поставляют узлы X_1 и X_2 , соответственно, а на предприятии C происходит сборка, в каждый сборочный узел входит один узел X_1 и два узла X_2 .

На рисунке предприятиям A , B и C соответствуют переходы t_1 , t_2 и t_3 .



Срабатывание перехода t_3 происходит только в том случае, если:

во-первых, в позиции p_1 имеется метка, а в позиции p_2 - не менее двух меток, что означает поступление от A и B соответствующих комплектующих;

во-вторых, имеется метка в позиции p_4 , что означает, что предприятие C закончило сборку предыдущего изделия и готово приступить к сборке следующего.

Пока очередное изделие не будет собрано, метки в p_4 не будет, следовательно, запросы, пришедшие во входные позиции p_1 и p_2 , вынуждены ожидать срабатывания перехода t_4 . Переходам t_1 , t_2 и t_3 поставлены в соответствие процедуры вычисления задержек срабатывания. Задержки в первых двух переходах равны интервалам времени между появлениями готовых узлов, задержка в t_3 равна времени сборки изделия.

Особыми видами сетей Петри являются ингибиторная сеть (в ней возможны ингибиторные дуги, запрещающие срабатывания перехода, если во входной позиции, связанной с переходом ингибиторной дугой, находится метка), а также иерархическая сеть. Иерархическая сеть содержит не мгновенные переходы, в которые вложены другие, возможно, также иерархические, сети. Срабатывание такого перехода характеризует выполнение полного жизненного цикла вложенной сети.

Еще одним случаем сетей Петри являются т.н. WF-сети (сети потоков работ). Этот формализм был разработан голландским математиком Вил ван дер Аальстом для моделирования потоков работ в workflow-системах. Абстракция такой системы с равным успехом применяется для описания потоков работ, например, при сложном строительстве или проектировании или же разработки планировщика задач операционных систем.

Сеть Петри PN называется сетью потоков работ WFN , если выполняются следующие условия:

- существует только одна исходная позиция I , такая что отсутствуют переходы, входящие в I ;
- существует только одна конечная позиция O , такая что отсутствуют переходы, выходящие из O ;
- каждый узел данной сети расположен на пути от I к O .

WF-сети используются для проверки графов потоков работ на наличие таких структурных конфликтов, как тупики (deadlocks) и недостатки синхронизации (lack of synchronization). Структурные конфликты отсутствуют, если WF-сеть является бездефектной.

Свойство бездефектности или правильной завершаемости соответствует следующим требованиям:

- конечная позиция O достижима при любой последовательности переходов от позиции I ;
- WF-сеть не содержит лишних позиций (которые никогда не будут выполнены);
- при достижении конечной позиции данной сети не должно оставаться маркеров в промежуточных позициях.

Свойство бездефектности соответствует двум хорошо известным свойствам сетей Петри – живости и ограниченности. Ограниченность означает, что число меток в любой позиции сети не может превысить неко-

торого критического значения K , живость – возможность срабатывания любого перехода при функционировании моделируемого объекта.

Помимо этих свойств, в сетях Петри также изучают их безопасность – частный случай ограниченности, $K = 1$; достижимость – возможность перехода сети из одного заданного состояния (характеризуемого распределением меток) в другое и сохраняемость – постоянство загрузки ресурсов, выражаемое соотношением

$$\sum A_i N_i = const,$$

где N_i – число маркеров в i -й позиции, A_i – весовой коэффициент.

В 1974 году Тилак Аджервала показал, что ингибиторная сеть Петри является универсальной алгоритмической системой. В монографии В. Е. Котова приведен набросок доказательства, указывающий правила кодирования ингибиторной сетью программы счетчикового автомата Минского. Дж. Питерсон приводит примеры других расширенных классов сетей Петри, являющихся универсальной алгоритмической системой: синхронных и приоритетных. Построенная в явном виде универсальная сеть Петри насчитывала несколько тысяч вершин и недавно была уменьшена до 56 вершин. Такая сеть Петри является полной, по Тьюрингу, вычислительной машиной.

Помимо конечных, в настоящее время активно изучаются бесконечные сети Петри. Они были введены для верификации вычислительных решеток и позволяют определять свойства сетей Петри для регулярных структур (линейная, древовидная, квадратная, треугольная, шестиугольная и гиперкуб) произвольного размера, полученных путем композиции типовых фрагментов.

Клеточные автоматы

Одной из самых знаменитых вычислительных абстракций, служащей и игрушкой и головоломкой и серьезной моделью, является клеточный автомат. Эта модель изучается не только в математике, но и в физике, теоретической биологии и микромеханике.

Основное направление исследования клеточных автоматов — алгоритмическая разрешимость тех или иных проблем, также рассматриваются вопросы построения начальных состояний, при которых клеточный автомат будет решать заданную задачу.

Теория таких автоматов имеет богатую историю. Станислав Улам, работая в знаменитой создании атомной бомбы Лос-Аламосской национальной лаборатории в 1930-е годы, изучал рост кристаллов, используя простую решеточную модель. В это же время Джон фон Нейман, коллега Улама, работал над проблемой самовоспроизводящихся систем³¹.

Первоначальная концепция фон Неймана основывалась на идее робота, собирающего другого робота, однако он быстро осознал, что с практической стороны такого процесса будет много проблем – например, где взять столько запчастей? Тем не менее, смоделировать что-то самовоспроизводящееся было слишком большим искушением, и тогда Улам посоветовал ему более абстрактную математическую модель, подобную той, что он сам использовал для изучения роста кристаллов. Таким образом, возникла первая клеточно-автоматная система. Подобно решетке Улама, клеточный автомат фон Неймана двухмерный, а самовоспроизводящийся робот описан алгоритмически. В результате появился универсальный конструктор, работающий «внутри» клеточного автомата с окрестностью, включающей непосредственно прилегающие ячейки и имеющего 29 состояний. Фон Нейман доказал, что для такой модели существует паттерн, который будет бесконечно копировать самого себя.

Затем, в 1940-е годы, Норберт Винер и нейрофизиолог Артуро Розенблют (тот самый отец нейроинформатики), используя эти идеи, разработали клеточно-автоматную модель возбудимой среды. В 1960-е годы клеточные автоматы изучались как частный тип динамических систем, и впервые была установлена их связь с областью символьной динамики.

В 1970-е была разработана знаменитая двухмерная клеточно-автоматная модель с двумя состояниями, известная как игра «Жизнь». Изобретенная Джоном Конвеем и популяризованная Мартином Гарднером, она использует следующие правила: если клетка имеет двух «живых» соседей, она остаётся в прежнем состоянии. Если клетка имеет трёх «живых» соседей, она переходит в «живое» состояние. В остальных случаях клетка «умирает». Несмотря на свою простоту, система демонстрирует огромное разнообразие поведения, колеблясь между очевидным хаосом и удивительным порядком. Одним из феноменов игры «Жизнь» являются глайдеры — сочетания клеток, движущиеся по сетке как единое целое. Возможно построить автомат, в котором глайдеры будут выполнять некоторые вычисления, и впоследствии было показано, что игра «Жизнь» может эмулировать универсальную машину Тьюринга.

В 1969 году немецкий инженер Конрад Цузе опубликовал книгу «Вычислимый космос», где выдвинул предположение, что физические законы дискретны по своей природе, и что вся Вселенная является гигантским клеточным автоматом. Это была первая книга из области, называемой сейчас цифровой физикой.

В 1983 г. Стивен Вольфрам (основатель компании Wolfram Research, прославившейся знаменитым программным пакетом Mathematics) опубликовал первую из серии статей, исследующих очень простой, но

³¹ И заодно теоретически предсказал появление компьютерных вирусов – классического примера саморазмножающейся системы.

малоизученный класс моделей, называемых элементарными клеточными автоматами. привела С. Вольфрама предположил, что сложность естественных систем обусловлена сходным механизмом поведения этих простых автоматов.

Кроме того, в течение этого периода Вольфрам формулирует концепцию истинной случайности и вычислительной неприводимости, и выдвигает предположение, что разновидность «Жизни», известная как «Правило 110» (Rule 110), может быть универсальным автоматом — факт, доказанный в 1990 году его ассистентом Мэтью Куком. Результатом их исследований стала публикация в 2002 г. 1280-страничной монографии по клеточным автоматам «A New Kind of Science».

В том же году Пауль Чепмен построил образец «Жизни», которая моделирует регистровую машину Минского, описанную выше. Первая версия образца была большой (268096 живых ячеек на площади 4558x21469 клеток) и медленной (20 поколений/сек при использовании клиента Life32 Иогана Бонтеса на 400 MHz AMD K6-II), в дальнейшем она была неоднократно усовершенствована. Таким образом, было конструктивно доказано, что игра «Жизнь» является полной по Тьюрингу машиной — нетривиальный и поразительный факт.

Что же представляет собой клеточный автомат? Он включает регулярную решётку ячеек, каждая из которых может находиться в одном из конечного множества состояний, например, 1 и 0. Решетка может быть любой размерности. Для каждой ячейки определено множество ячеек, называемых окрестностью.

Чаще всего используют две естественных окрестности — фон Неймана и Мура. Окрестность первого порядка по фон Нейману — совокупность 4 клеток, имеющих с данной общую сторону, второго порядка, соответственно, имеющих общую сторону с первыми четырьмя и т.д. Окрестность Мура первого порядка — совокупность восьми точек, касающейся данной.

Окрестность фон Неймана порядка r — множество клеток, манхэттенское расстояние³² до которых от данной клетки не превышает r . Окрестность фон Неймана порядка r имеет форму ромба.

Окрестность Мура порядка r — множество клеток, расстояние Чебышева³³ до которых от данной клетки не превышает r . Окрестность Мура порядка r представляет собой квадрат со стороной $2r + 1$.

Для работы клеточного автомата требуется задание начального состояния всех ячеек и правил перехода из одного состояния в другое. На каждой итерации, используя правила перехода и состояния соседних ячеек, определяется новое состояние каждой ячейки. Обычно правила перехода одинаковы для всех ячеек и применяются сразу ко всей решётке.

Формально это выглядит так. Клеточный автомат можно определить как множество конечных автоматов, каждый из которых может находиться в одном из состояний

$$\sigma \in \Sigma = \{0, 1, 2, \dots, k\}$$

изменения состояний автоматов происходит согласно правилу перехода

$$\sigma_{i,j}(t+1) = \varphi(\sigma_{k,l}(t) | \sigma_{k,l}(t) \in \mathcal{N}),$$

где $\mathcal{N}$ - множество автоматов, составляющих окрестность. Для окрестности фон Неймана, например:

$$\mathcal{N}_N^1(i, j) = \{\sigma_{k,l} | |i - k| + |j - l| \leq 1\},$$

а для окрестности Мура

$$\mathcal{N}_M^1(i, j) = \{\sigma_{k,l} | |i - k| \leq 1, |j - l| \leq 1\}$$

Число всех возможных правил перехода определяется числом состояний σ и количеством соседей n и составляет

$$N_r = \sigma^n.$$

Клеточный автомат называется обратимым, если для каждой текущей конфигурации существует только одна предшествующая конфигурация. Если рассматривать клеточный автомат как функцию, отображающую

³² Так же известно, как расстояние городских кварталов, норма l_1 (так как она является также одной из норм в лебеговых пространствах — пространствах функций, измеримых по Лебегу, там ее называют 4-метрикой), метрика такси, прямоугольная метрика — метрика, введенная Г. Минковским. Согласно этой метрике, расстояние между двумя векторами равно сумме модулей разностей их координат. Более формально, для двух векторов p и q

$$l_1(p, q) = \sum_{i=1}^n |p_i - q_i|$$

³³ Так же известно, как метрика решетки, равномерная метрика, метрика l_∞ (она также является частным случаем лебеговских метрик, в функциональном анализе она известна как 8-метрика). Согласно ей, расстоянием между векторами называется максимум модуля разности компонент этих векторов: $l_1(p, q) = |p_i - q_i|$. Расстояние между полями шахматной доски для виэри (или ладьи, если расстояние считать в клетках) равно манхэттенскому расстоянию; король и ферзь пользуются расстоянием Чебышева.

щую одну конфигурацию в другую, то обратимость предполагает биективность этой функции. Если клеточный автомат обратим, то его обратная эволюция также может быть описана клеточным автоматом. Конфигурации, не имеющие предшественников, т.е. недостижимые в данном клеточном автомате, носят название «Сады Эдема»³⁴.

Для одномерных клеточных автоматов существуют алгоритмы определения обратимости или необратимости. Однако для клеточных автоматов с двумя и более измерениями таких алгоритмов нет.

Обратимые клеточные автоматы часто используют для моделирования таких физических феноменов, как динамика жидкости и газа, поскольку они подчиняются законам термодинамики. Такие автоматы специально создаются обратимыми. Такие системы изучались Томасо Тоффоли (Tommaso Toffoli) и Норманом Марголюсом (Norman Margolus). Существует несколько типов обратимых конечных автоматов. Наиболее известными являются клеточный автомат второго порядка и блочный клеточный автомат. Обе эти модели следуют несколько модифицированному варианту определения клеточного автомата, однако доказано, что они могут быть эмулированы традиционным клеточным автоматом со значительно большим размером окрестности и числом состояний. Также было доказано, что любой обратимый клеточный автомат может быть эмулирован блочным клеточным автоматом.

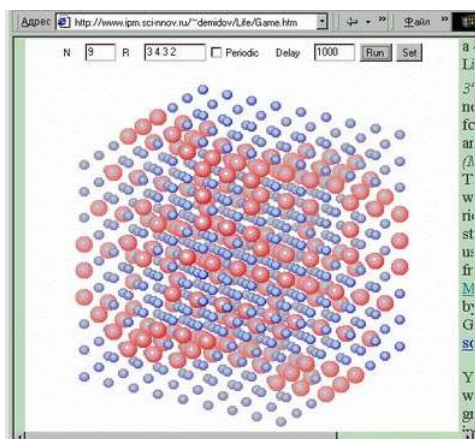
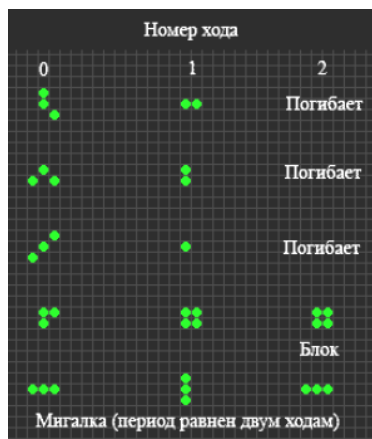
Наиболее известным клеточным автоматом является игра «Жизнь». Ее правила очень просты. Рассматривается ограниченный или неограниченный клетчатый лист, в начале игры на нем хаотично (или по правилам) распределяется несколько «живых» клеток (математически, например, квадрату присваивается цифра 1). Это первое поколение. Далее начинается сама игра. Каждое следующее поколение рассчитывается на основе предыдущего по таким правилам:

- в пустой (мёртвой, цифра 0) клетке, рядом с которой ровно три живые клетки, зарождается жизнь;
- если у живой клетки есть две или три живые соседки, то эта клетка продолжает жить; в противном случае (если соседей меньше двух или больше трёх) клетка умирает («от одиночества» или «от перенаселённости»).

Игра прекращается, если на поле не останется ни одной «живой» клетки, если при очередном шаге ни одна из клеток не меняет своего состояния (складывается стабильная конфигурация) или если конфигурация на очередном шаге в точности (без сдвигов и поворотов) повторит себя же на одном из более ранних шагов (складывается периодическая конфигурация, глайдер).

Эти простые правила приводят к огромному разнообразию форм, которые могут возникнуть в игре. Игрок не принимает прямого участия в игре, а лишь расставляет или генерирует начальную конфигурацию «живых» клеток, которые затем взаимодействуют согласно правилам уже без его участия. Как уже упоминалось, определенная конфигурация «Жизни» может эмулировать работу машины Тьюринга.

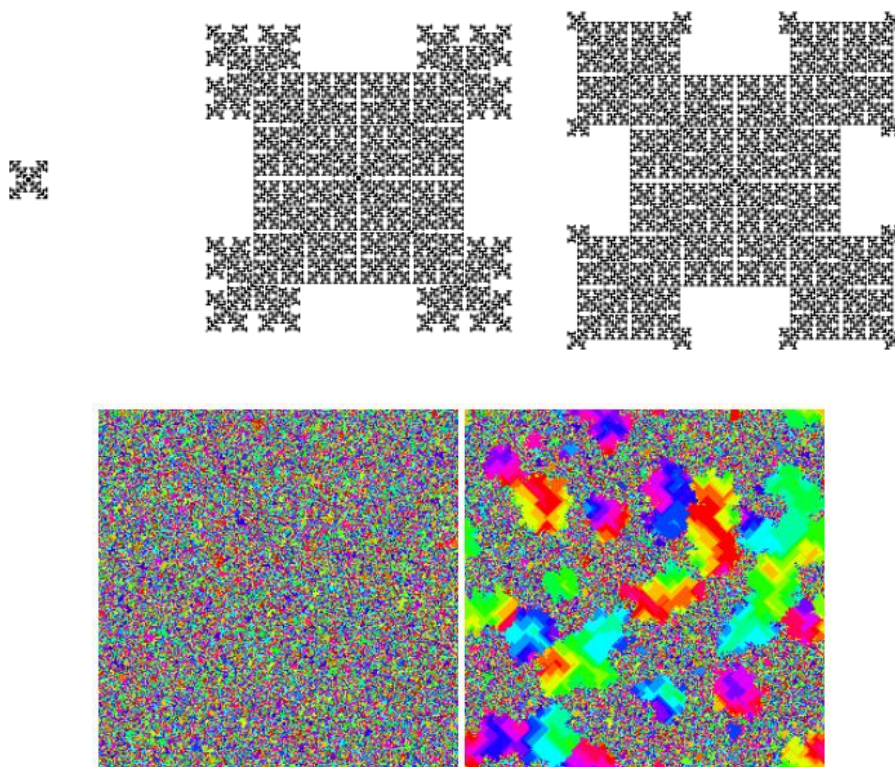
Вот пример самой простой вариации в такой игре:



Существуют, конечно, конфигурации на порядок сложнее, в т.ч. трехмерные.

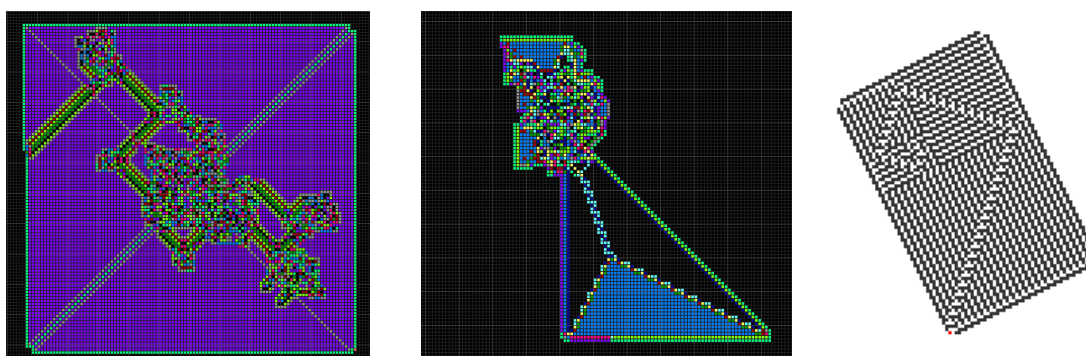
³⁴ Две конечные конфигурации клеточного автомата называются близнецами, если их эволюции, начиная со следующего поколения, полностью совпадают. Клеточный автомат называется инъективным, если в этом автомате нет близнецов. Клеточный автомат называется сюръективным в том и только в том случае, если у каждой конфигурации есть родитель, то есть если садов Эдема не существует. Автомат, одновременно являющийся инъективным и сюръективным, называется обратимым клеточным автоматом. Теорема сада Эдема утверждает, что клеточный автомат в евклидовой вселенной является локально инъективным тогда и только тогда, когда он сюръективен. Другими словами, теорема утверждает, что сады Эдема существуют только в тех автоматах, в которых существуют близнецы. Теорема применима к «Жизни», поскольку легко найти две различные конфигурации, которые эволюционируют в следующем поколении в одну и ту же конфигурацию. «Мёртвая вселенная» и одинокая живая клетка в «мёртвой вселенной» эволюционируют в одну и ту же конфигурацию, все клетки которой мёртвые. Следовательно, в «Жизни» существуют сады Эдема. Теорема сада Эдема была выдвинута Эдвардом Муром и доказана Муром и Джоном Майхиллом. Относительно эффективный метод отыскания таких садов основан на теории формальных языков; временная сложность этого подхода зависит экспоненциально не от площади, а от ширины ограничивающего прямоугольника. Первый известный сад Эдема в «Жизни», размещающийся в прямоугольнике 9×33 , был найден Роджером Бэнком в 1971 году. В 1973–74 гг. были построены сады Эдема в прямоугольниках 6×122 и 6×117 . В декабре 2011 года был найден сад Эдема, состоящий из 56 живых клеток и умещающийся в квадрате 10×10 ; также было выяснено, что сады Эдема в прямоугольниках меньше 6×6 не существуют.

Если модифицировать «Жизнь», то можно получить не менее интересные автоматы, генерирующие что угодно, от фракталов до моделирования вычислений:



Еще одним вариантом клеточного автомата является муравей Лэнгтона — двумерная машина Тьюринга с очень простыми правилами, изобретенная Крисом Лэнгтоном.

Муравей Лэнгтона может быть описан как клеточный автомат, в котором почти все поле покрашено в черно-белый цвет, а клетка с «муравьем» имеет один из восьми различных цветов, кодирующих соответственно все возможные комбинации черного/белого цвета клетки и направления движения муравья.



1.5.6. Вычисления на графах и потоковые ЭВМ

Последняя треть прошлого века была чрезвычайно плодотворным временем для развития ЭВМ. В 1970-е годы появилось невероятное количество идей (волна от многих из них докатилась даже до 90-х гг.), посвященных увеличению производительности компьютеров. Несколько из них были даже больше чем идеями – скорее философскими принципами организации вычислений. Это потоковые (Data Flow Computers или DFC, в противоположность классической архитектуре Control Flow – потока команд) и редуционные машины³⁵.

³⁵ Подробное описание DF-машин можно найти по адресу <http://habrahabr.ru/post/122479/> и <http://habrahabr.ru/post/122530/>

В чем же их революционность? Начиная со времен фон Неймана, считалось само-собой разумеющимся, что вводимая в компьютер информация делится на команды и данные, при этом данные играют подчиненную роль, т.е. они не определяют логику поведения машины. Команды, помимо обработки данных, управляют всем вычислительным процессом в целом. Это простая и логичная идея, и никто не думал, что возможно иначе, так как большая часть содержания задач тех лет была сосредоточена в алгоритме их решения, а не в конкретных данных. Любой программист легко узнает, например, стандартные численные алгоритмы решения систем линейных уравнений, но ничего не сможет сказать о задаче, если ему дать от нее одни данные – пару матриц.

К концу 70 гг. ситуация стала меняться, так как компьютеры стали использовать не только для математики и инженерных расчетов. Это было время зарождения автоматизации менеджмента и появления первых СУБД – сложных программных комплексов, в которых огромные базы данных полностью описывали логику работы предприятий, а простые программы, их обрабатывающие, играли подчиненную роль. Узнать все о задаче теперь было можно, изучив одни лишь данные, напротив, алгоритм их обработки ничем бы не помог. В результате родилась концепция передать управляющую роль потокам данных, зафиксировав статичные команды, которые их обрабатывают. Не команды выбирают для обработки поток данных, а данные выбирают, какой командой быть обработанными. Так родилась потоковая архитектура.

Архитектура control flow сама по себе имеет ряд врожденных недостатков, полностью избавиться от которых невозможно, так как они проистекают из самой организации вычислительного процесса, можно только уменьшить негативный эффект с помощью различных технических решений. Вот ее основные проблемы:

- Перед выполнением инструкции ее операнды необходимо загрузить из памяти в регистры процессора, а после выполнения — выгрузить результат обратно в память. Шина процессор-память становится узким местом: процессор простаивает часть времени, ожидая загрузки данных. Проблема с переменным успехом решается при помощи упреждающей выборки и нескольких уровней кэш-памяти.
- Построение многопроцессорных систем сопряжено с рядом трудностей. Существуют две основные концепции таких систем: с общей и с распределенной памятью. В первом случае сложно физически обеспечить совместный доступ многих процессоров к одному ОЗУ. Во втором случае возникают проблемы когерентности данных и синхронизации. С ростом числа процессоров в системе все больше ресурсов тратится на обеспечение синхронизации и все меньше — на собственно вычисления.
- Никто не гарантирует, что на момент выполнения какой-либо инструкции ее операнды будут находиться в памяти по указанным адресам. Инструкция, которая должна записать эти данные, может оказаться, еще не выполнена. В многопоточных приложениях существенная доля ресурсов тратится на обеспечение синхронизации потоков.

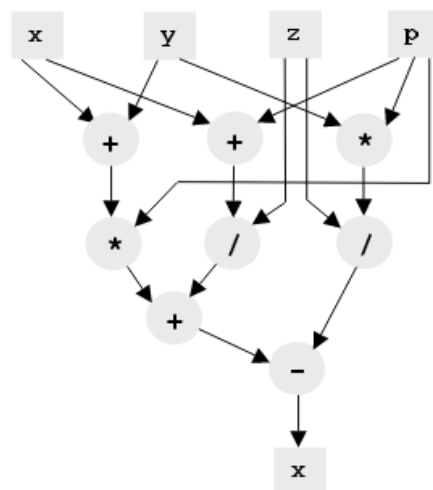
Все конечные автоматы³⁶, по сути, потоковые машины. В машине Тьюринга, например, именно данные, записанные на ленту, определяют, какое состояние примет машина! Однако, как обычно, просто было на бумаге – архитектура настоящих Data Flow-компьютеров оказалась чрезвычайно сложной, а их программирование еще сложнее, не говоря уже о концептуальной непривычности – никаких дешифраторов и счетчиков команд, никаких программ, хранимых в памяти, и т.п.

Рассмотрим классический пример. Допустим, надо вычислить простое выражение, типа

$$x := (x + y) \cdot p + (x + p)/z - p \cdot y/z$$

Потоковая машина, прежде всего, должна представить это выражение в виде ориентированного графа потоков данных. В нем есть 2 вида узлов – сами данные, изображенные квадратами (т.е. значение переменных), и обрабатывающие элементы (статичные монозадачные блоки, реализующие обычные операции сложения, деления и т.п.).

Идея потоковых вычислений состоит в следующем. Все действия над данными проводят обрабатывающие элементы (операторы, собираемые из обычных вентилях). Каждый элемент начинает обработку, когда ему на вход поступают готовые и подходящие данные. В нашем примере на первом такте автоматически и параллельно начинают работу 2 цепи сложения и одна умножения, затем, на втором такте, умножение и 2 деления, на третьем такте снова происходит умножение, на четвертом остается одно вычитание и расчет закончен. Вообще это похоже на внутренности логического вентиля – в определенном смысле каждый такой – маленькая монозадачная потоковая машина – он получает на входе произвольные данные и обрабатывает их фиксированными пересылками от транзистора к транзистору.



³⁶ в т.ч. и даже прочие алгоритмические абстракции, цепи Маркова, например.

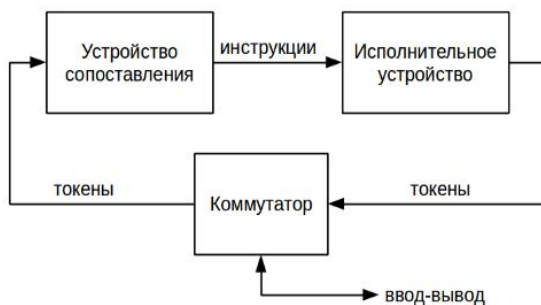
Естественно, данные необходимо снабдить какими-то меняющимися тегами, на которые будут реагировать обрабатывающие элементы, это необходимо для задания пути их прохождения через АЛУ.

Граф потока данных и является программой для DF-машины, а программы в традиционном смысле здесь не существует. Конечно, описать это все на обычном языке не представляется возможным, так что параллельно с разработкой железа продвигалась и работа над DF-языками (Data Flow Language). Помимо концептуальной изысканности, DF-машины имеют колоссальное преимущество – естественный врожденный параллелизм – любые готовые данные тут же поступают на соответствующие обработчики. Ясно, что при наличии оптимального графа решить задачу быстрее, чем на DF-компьютере, невозможно в принципе.

Сам термин dataflow впервые был предложен в 1969 г. Адамсом из Стэнфордского университета. Вскоре в Массачусетском технологическом институте появился бумажный проект такого компьютера; потом появился язык Id (Irvine dataflow). В 1979 г. во Франции был построен первый многопроцессорный DFC-компьютер, называвшийся CERT-ONERA. Примерно в то же время компания Texas Instruments разрабатывала свою версию системы DDP (Data-Driven Processor). Первым коммерческим, отличным от фон-неймановского, проектом стал процессор Data-Driven Media Processor (DDMP), предложенный в 1995 году компанией Sharp для обработки мультимедийных данных.

Ни одна из этих попыток не получила серьезного признания и не была успешной с коммерческой точки зрения. Тем не менее, считать потоковые подходы к созданию компьютеров тупиковой веткой эволюции было бы неверно. Идеи, родившиеся в качестве альтернативы фон-неймановской схеме, продолжают жить.

В результате появились разнообразные многопотоковые модели (single-token-per-arc dataflow, tagged-token dataflow, explicit token store, threaded dataflow, large-grain dataflow, RISC dataflow, cycle-by-cycle interleaved multithreading, block interleaved multithreading, simultaneous multithreading), развитие которых с переменным успехом и не очень бурно отчасти идет и сейчас.

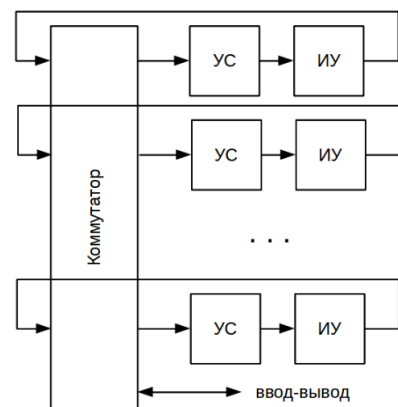


Вот как должна работать аппаратная часть таких машин. В потоковых машинах данные передаются и хранятся в виде т.н. токенов (token). Токен – это структура, содержащая собственно передаваемое значение и метку – указатель узла назначения. Простейшая потоковая вычислительная система состоит из двух устройств: исполнительного (execution unit) и устройства сопоставления (matching unit).

Исполнительное устройство служит для выполнения инструкций и формирования токенов с результатами операций. Как правило, оно включает в себя память команд, доступную только для чтения. Готовность входных данных узла определяется по наличию

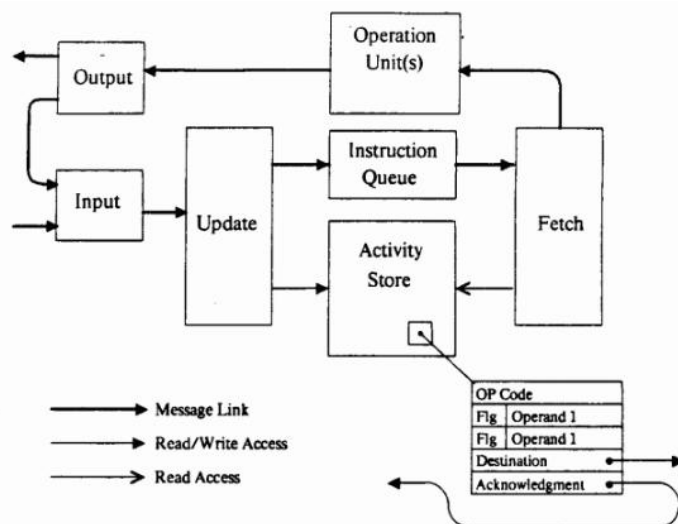
набора токенов с одинаковыми метками. Для поиска таких наборов и служит устройство сопоставления. Обычно оно реализуется на базе ассоциативной памяти. Используется либо «настоящая», аппаратная ассоциативная память (CAM — content-addressable memory), либо структуры, работающие аналогично, например, хэш-таблицы.

Одним из основных достоинств DF -архитектуры является ее масштабируемость: не составляет труда собрать систему, содержащую множество устройств сопоставления и исполнительных устройств. Устройства объединяются простейшим коммутатором, причем для адресации токенов служат их метки. Весь диапазон номеров узлов просто распределяется равномерно между устройствами. Никаких дополнительных мер для синхронизации вычислительного процесса, в отличие от многопроцессорной controlflow-архитектуры, не требуется:



Описанная выше схема называется статической (static dataflow, SDFC). В ней каждый вычислительный узел представлен в единственном экземпляре, число узлов заранее известно, также заранее известно число токенов, циркулирующих в системе. В качестве примера реализации статической архитектуры можно привести MIT Static Dataflow Machine — потоковый компьютер, созданный в Массачусетском технологическом институте в 1974 году. Машина состояла из множества обрабатывающих элементов (Processing Element), связанных коммуникационной сетью.

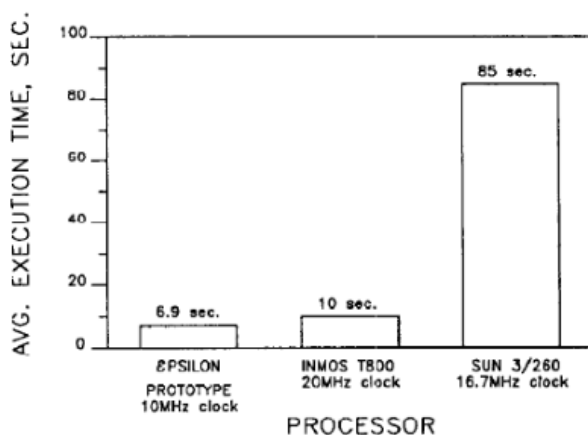
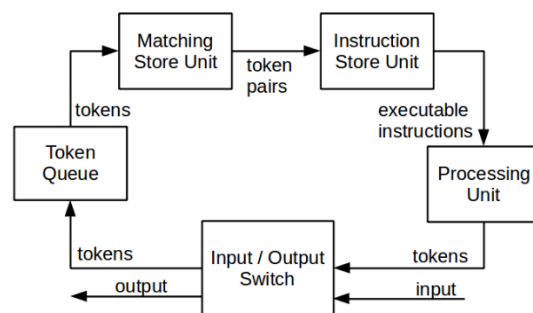
Схема одного элемента показана на рисунке.



связанных между собой и имеющая входы и выходы. Можно одновременно вызывать несколько экземпляров одной и той же процедуры, которые будут отличаться контекстом.

- Распараллеливание циклов. Если между итерациями цикла нет зависимости по данным, можно обрабатывать сразу все итерации одновременно. Номер итерации будет содержаться в поле контекста.

Одной из первых реализаций динамической потоковой архитектуры была система Manchester Dataflow Machine (1980). Машина содержала аппаратные средства для организации рекурсий, вызова процедур, раскрытия циклов, копирования и объединения ветвей вычислительного графа. Также в отдельный модуль была вынесена память команд (instruction store unit). На рисунке показана схема одного элемента машины.



Динамическая dataflow-архитектура, по сравнению со статической, демонстрирует более высокую производительность за счет лучшего параллелизма вычислений. Кроме того, она дает больше возможностей для программиста. С другой стороны, динамическая система сложнее по аппаратной реализации, особенно это касается устройств сопоставления и блоков формирования контекста токенов. Вот некоторые данные о сравнительной производительности в виде времени, затраченного на выполнение одного и того же кода на SUN 3 и двух прототипов DF-машин.

Dataflow-системы очень эффективны для обработки разреженных данных, так как фактически пересылаются и обрабатываются только значимые элементы. Чистые потоковые архитектуры, подобные описанным MIT Static Dataflow Machine и Manchester Dataflow Machine, к сожалению, имели много слабых мест:

- Dataflow-машины давали огромные возможности для параллелизма выполнения. Обратной стороной этого преимущества было то, что на последовательных участках вычислительного графа они показывали резкое падение производительности.
- Загрузка исполнительных устройств была далека от максимально возможной. Большая часть машинного времени тратилась на поиск соответствия операндов, выборку инструкций, а исполнительное устройство все это время простаивало, выполняя лишь по одной инструкции на каждую пару токенов.

- Трудным было конструирование устройств сопоставления. Ассоциативная память сложнее, дороже, медленнее, занимает больше места и потребляет больше энергии, по сравнению с обычной оперативной памятью такого же объема.

- Сам принцип управления потоком данных не позволял организовать эффективный конвейер. Почти все устройства работали асинхронно, требовались буферы и очереди в линиях связи.

- По сравнению с классической многопроцессорной архитектурой, в dataflow-машинах значительно выше была нагрузка на коммутационную сеть. Ведь фактически каждая операция требовала пересылки двух токенов.

В попытках решить перечисленные проблемы стали появляться гибридные архитектуры, сочетающие в себе элементы как архитектур потока данных, так и потока управления. Так родилась идея Threaded dataflow, этому термину нет вообще никакого адекватного перевода на русский язык.

Суть данного подхода состоит в том, чтобы последовательные участки вычислительного графа, которые невозможно распараллелить, заменить тредами (threads) — наборами последовательно выполняемых инструкций. Сразу исчезают лишние промежуточные токены, и растет загрузка исполнительных устройств. Принципы threaded dataflow были воплощены в процессоре Epsilon в 1989 году. Дальнейшим развитием threaded dataflow стали так называемые крупнозернистые потоковые архитектуры (large-grain dataflow). Когда стало ясно, что параллелизм чистого dataflow во многих случаях избыточен, возникло решение строить потоковый граф не из отдельных операторов, а из блоков. Таким образом, в крупнозернистой архитектуре каждый узел представляет собой не одну инструкцию, а классическую последовательную программу. Взаимодействие между узлами по-прежнему осуществляется по принципу потока данных. Одним из преимуществ такого подхода стала возможность использовать в качестве исполнительных устройств обычные фон-неймановские процессоры. Следует отметить, что несмотря на название «крупнозернистая», блоки, на которые разбивается задача, все равно намного мельче, чем, скажем, в кластерных системах. Типичный размер блока составляет 10-100 инструкций или 16-1К байт данных.

В СССР работой над DF-машинами занимались В.С. Бурцев и В.М. Глушков. По их терминологии, принцип Threaded dataflow получил название мелкозернистого параллелизма и был успешно реализован В.М. Глушковым и В.А. Торгашевым в мультипроцессорах с динамической архитектурой (МДА). В основу МДА была положена теория динамических автоматных сетей, где программа представляет собой не линейную последовательность команд, а совокупность связанных между собой линиями обмена данных автономных вычислительных узлов (конечных автоматов).

Начальное состояние алгоритма задается перечнем вычислительных узлов и структурой сети (совокупность информационных связей между узлами) и алгоритмом работы каждого узла. Структура сети может быть описана с помощью специального языка программирования, либо с помощью графического языка. Во время вычислительного процесса меняются не только состояния узлов, но может трансформироваться и вся сеть: узлы порождают узлов-потомков, уничтожают их и меняют информационные связи. Решением поставленной перед автоматной сетью задачи является такое состояние сети, в котором она теряет способность к автотрансформации. В МДА распараллеливание алгоритма происходит автоматически без участия программиста: каждому узлу системы выделяется свой виртуальный процессор, а операционная система сама принимает решение о том, какому узлу и когда выделять аппаратные ресурсы (процессор и память). Однако данная архитектура, несмотря на довольно удачные экспериментальные реализации, распространения не получила.

Глушков активно продвигал не-фон-Неймановские машины, в т.ч. на конференциях за рубежом, много публиковался. К сожалению, в те годы в СССР развитие чего либо, кроме ЕС ЭВМ, было невозможно иначе, как в виде нескольких прототипов, а сложности DF-машин позволили бы серьезно продвинуться в их создании только совместными усилиями ведущих конструкторов и при полной государственной поддержке. Для сравнения на Западе ведущие конструкторы многих серьезных фирм стали получать практический результат не совсем в том направлении, на которое рассчитывали изначально, лишь после 15-20 лет разработки DF-архитектуры.

Настоящая судьба потоковой архитектуры в России, как и многое другое, бессмысленна.

На Западе дальнейшим развитием Threaded dataflow стали векторные потоковые системы 1980-х годов. В таких системах токен содержит не одно значение, а сразу множество. Соответственно, и операции выполняются не над парами операндов, а над парами векторов. В качестве примера такой системы можно привести машину SIGMA-1 (1988). Иногда векторный режим поддерживается только частью исполнительных устройств системы. Часто также применяются гибридные архитектуры, объединяющие сразу несколько подходов, например крупнозернистая архитектура с возможностью выполнения векторных операций.

Развитие технологий ПЛИС (FPGA) сделало возможным принципиально новый подход к архитектуре dataflow. Что, если собрать машину, ориентированную на решение одной конкретной задачи? Если реализовать непосредственно на схемотехническом уровне нужный вычислительный граф, можно добиться потрясающих результатов. Вместо сложных и медленных устройств сопоставления можно использовать безусловное перенаправление данных от одного функционального модуля к другому.

Сами исполнительные устройства тоже можно собрать под нужную задачу: выбрать тип арифметики, разрядность, нужный набор поддерживаемых операций. Разумеется, подобная машина будет очень узкоспециализированной, но ведь достоинство ПЛИС как раз в возможности неоднократного перепрограммирования. Таким образом, под каждую отдельную задачу собирается нужная архитектура. Некоторые системы позволяют осуществлять перенастройку прямо в процессе работы. Реконфигурируемые системы на базе FPGA-микросхем в настоящее время выпускаются серийно в самых разных форматах — от блока ускорителя для ПК до системы производительностью порядка нескольких Тфлопс.

Из недостатков реконфигурируемых архитектур можно выделить следующие:

- Принципиальная однозадачность. Для запуска новой задачи требуется остановка системы и перепрограммирование ПЛИС, входящих в ее состав.
- Сложность программирования. Программирование каждой задачи включает в себя синтез всей вычислительной архитектуры под данную задачу.
- Избыточная аппаратная сложность. Обратной стороной гибкости ПЛИС является наличие на кристалле большого процента элементов, которые непосредственно не участвуют в вычислениях, а служат только для реконфигурации. Тем не менее, эти элементы потребляют энергию и выделяют тепло во время работы, что ухудшает энергетические показатели эффективности системы (Гфлопс/Вт).

Программирование в парадигме dataflow существенно отличается от привычного структурного, а ближе, скорее, к функциональному. Здесь нет переменных, массивов и других именованных областей памяти, нет вызываемых процедур и функций в привычном смысле этого слова. Потокное программирование использует принцип однократного присваивания. Каждая единица данных (как правило, это токен) получает значение только один раз, при создании, и в дальнейшем может быть только прочитана. Этот принцип отражает однонаправленность ветвей вычислительного графа. В dataflow, к сожалению, так и не выработался какой-либо стандарт программирования, обычно под каждую архитектуру разрабатывается свой язык.

Вот несколько наиболее известных:

VAL (Value-oriented Algorithmic Language) был разработан в MIT в 1979 году специально для MIT Dataflow Machine. Вот как выглядит, к примеру, вычисление факториала на VAL:

```
for Y:integer := 1; P:integer := N;
do      if P ~= 1 then iter Y := Y*P; P := P-1; enditer;
  else Y
  endif
endfor
```

SISAL (Streams and Iteration in a Single Assignment Language), появившийся в 1983 году, является дальнейшим развитием VAL. В отличие от VAL, позволяет использовать рекурсию. Вычисление факториала:

```
function factorial( n : integer returns integer )
  if n <= 1 then 1
  else n * factorial( n - 1 )
  end if
end function
```

Id — параллельный язык общего назначения, создан в конце 1970 – начале 1980 гг. в MIT. Дальнейшая работа над Id привела к развитию языка pH — параллельного диалекта Haskell'a. Факториал на Id:

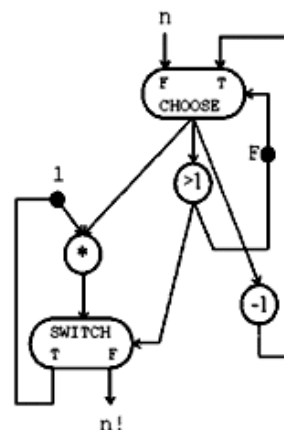
```
( initial j <- n; k <- 1
  while j > 1 do
    new j <- j - 1;
    new k <- k * j;
  return k )
```

А граф вычислений для данной программы будет выглядеть так:

Lucid. Разработан в 1976 Биллом Уэйджем (Bill Wadge) и Эдом Эшкрофтом (Ed Ashcroft). Опирается на понятия потоков значений (аналог переменных) и фильтров, или преобразователей (аналог функций). Факториал на Lucid:

```
fac
where
  n = 0 fby (n + 1);
  fac = 1 fby ( fac * (n + 1) );
end
```

Quil. Язык, основанный на Lucid, разработан в 2010, в настоящее время работа продолжается. На сайте языка есть онлайн-интерпретатор.



Бум потоковых архитектур пришелся на 1970-80-е годы, потом интерес к dataflow постепенно угасал. В наши дни элементы потоковых вычислителей можно встретить в архитектуре DSP (сигнальных процессоров), сетевых маршрутизаторов и графических процессоров. Таким образом, потоковые машины в чистом виде оказались красивой утопией, тем не менее, идеи, лежащие в их основе, вполне работоспособны при разумных ограничениях, например, отказе от детерминизма вычислений.

В результате DF-архитектура породила систолические массивы, Wawefront-архитектуру и нейропроцессоры — они активно применяются там, где задача одна, но сложноформализуема, полносвязность не важна, и результат надо получить в тысячи раз быстрее, чем на это способна универсальная ЭВМ. Классический пример таких проблем — распознавание изображений и звука. Также упоминалась еще одна важная область, где потоковая архитектура не только приносит дивиденды, но и фактически вся отрасль выросла из нее — ПЛИС или FPGA, на которых реализовано огромное количество архитектур микроконтроллеров, в т.ч. и знаменитые Forth-машины.

Интересной разновидностью DF-машин были т.н. вычислительные системы с управлением по запросу или редукционные машины (Demand-driven control computer, DDCC). В традиционных потоковых машинах каждая команда, для которой имеются данные, немедленно выполняется вне зависимости от того, нужна она сейчас реально или нет, и узлы DF-графа запускаются сверху вниз.

Модифицировать эту схему можно, введя обратную связь снизу-вверх, когда запускаться будут только необходимые в данный момент вычисления, это т.н. нисходящая обработка, получившая название редукция графа. Соответственно, такая модификация DF-машин получила название редукционной архитектуры.

Математическую основу таких ЭВМ составляет λ -исчисление Чёрча и соответствующие ему функциональные языки, такие, как Erlang, Haskell, FP и др. На функциональном языке программы представляются в виде выражений, а процесс выполнения программы заключается в получении последовательных значений этих выражений (оценке выражений). Оценка производится посредством повторения операций выбора и упрощения тех частей выражения, для которых упрощение возможно (такая часть называется редексом), операция упрощения называется редукцией. Процесс завершается, когда в выражении не остается редексов. Такое выражение называется нормальной формой.

1.5.7. λ -исчисление и редукционные ЭВМ

Элементарная синтаксическая конструкция, как один из способов формализовать понятие вычислимо-сти приобрела в глазах рядового программиста стойкую ауру мистического знания.

Во-первых, λ -исчисление изначально не задумывалось как язык программирования. В нем не заложено никаких облегчающих написание программ и изучение их методических принципов. Во-вторых, и это главное, это математический инструмент, прежде всего, для самих математиков. Наконец, в-третьих, существует очень мало курсов, направленных на изучение программирования, начиная с функционального, а не императивного.

Нет ячеек памяти, нет адресов, нет циклов и условных переходов, даже команд нет и вообще нет никакой последовательности вычислений, как процесса изменения состояния памяти под управлением программы! Книги по функциональному программированию обычно написаны без углубления в теорию, без которой там сложно что-нибудь понять, а книги по теории алгоритмов, наоборот, чрезмерно перегружены чистой математикой.

Неудивительно, что в таких условиях, какой-нибудь Haskell³⁷ считается среди программистов эзотерическим языком, наподобие легендарного Barainfuck, людей на нем реально пишущих мало и вызывают они ощущение гуру, достигших просветления.

Как и все пособие, эта глава не несет цели дать исчерпывающие знания по освещаемому вопросу. Она направлена, скорее, на облегчение подготовки к чтению такой литературы и объясняет базовые принципы, лежащие в основе функционального программирования.

Предположим, у нас есть функции от одного аргумента, причем и их аргумент, и возвращаемое значение — тоже функции от одного аргумента³⁸. Что мы можем сделать, имея под рукой лишь такой абстрактный набор странных функций? Для начала неплохо бы определиться, как эти функции записывать. В λ -исчислении приняты такие правила:

- Пусть f и a — некоторые функции. Тогда $f\ a$ обозначает применение функции f к функции a , или, в терминологии λ -исчисления, *аппликация*. Аналогичная операция могла бы быть записана как $f(a)$, как это принято в традиционной математике, но так не делают, чтобы подчеркнуть, что f и a — объекты одной природы, это функции, и они обе от чего-то зависят, а не функция и ее аргумент. Т.е. если создать грубую анало-

³⁷ <http://habrahabr.ru/post/152889/> и продолжение здесь <http://www.pvsm.ru/programirovanie/16451> - неплохое введение в этот легендарный язык, также рекомендуется доступная по уровню книга <http://akoub.narod.ru/funchook/index.htm>

³⁸ Здесь математики возразят: у нас получается множество, совпадающее с множеством функций из себя в себя, чего в теории множеств быть не может (даже в случае бесконечных множеств, множество функций строго больше). Но это не помешало американцу Дана Скотту построить декартово-замкнутую категорию, введя особую топологию на частично упорядоченных множествах, т.е. представленный объект действительно существует, но это уже совсем другая история.

гию, то в λ -исчислении запись *sin cos* означает применение функции *sin* к функции *cos*, то есть то, что по-человечески записалось бы, как *sin (cos (...))*. Как же все-таки указать, от чего зависит функция?

- Пусть F — выражение, свободно содержащее переменную x (свободно означает, что x ни с чем больше не связана и нигде не входит), либо вообще её не содержащее. Тогда $\lambda x.F$ обозначает функцию от x , значение которой задается выражением F , и называется это λ -абстракцией. По-человечески это выглядело бы как $y = F(x)$.

- Тогда $\lambda x.f\ x = f$, так как эта функция принимает аргумент x и возвращает значение $f\ x$, что и делает сама функция f . Это называется η -преобразованием.

- Например, $\lambda x.x^2$ — функция возведения в квадрат, а $\lambda x.x^2\ 3$ — применение этой функции к аргументу 3. Интуиция подсказывает, что значение этого выражения — 9, что и утверждает единственная аксиома λ -исчисления, называемая β -редукцией: $\lambda x.F\ a = F(x := a)$ (иногда пишут со скобками, $(\lambda x.F)a$ где $F(x := a)$ означает выражение F , в котором вместо x подставлено a . $\lambda x.F\ a$ называется *редексом*, $F(x := a)$ его *контрактом*. Любое выражение λ -исчисления принято называть, как в формальной грамматике *термом*. Подстановка также может обозначаться стрелкой, а контракт вот так: $[y/x]$.

- Поскольку результат β -редукции определяется подстановкой, возникает опасность *коллизии имён переменных*. Рассмотрим терм $\lambda xy.x\ y = \lambda x.(\lambda y.x)y$. Выполним редукцию:

$$\lambda x.(\lambda y.x)y \rightarrow \lambda x.(\lambda y.x)[y/x] = \lambda y.y$$

У нас получилась тождественная функция, в то время как $\lambda y.x$ описывает константу x . Поэтому необходимо переименование связанной переменной y в терме $\lambda y.x$:

$$\lambda x.(\lambda y.x)y \rightarrow \lambda x.(\lambda y.x)[z/y][y/x] = \lambda z.y$$

Выражения, отличающиеся только именами связанных переменных, полностью эквивалентны (*алфавитно эквивалентны*), например, $\lambda x.x\ a = \lambda y.y\ a = a$. Замену имен связанных переменных называют α -преобразованием, или α -редукцией.

Вместе с аксиомой β -редукции λ -исчисление становится Тьюринг-полным языком программирования, т.е. на нем можно реализовать любой алгоритм (в разумном понимании того, что же такое алгоритм).

Несложно набросать тождественную функцию, просто возвращающую свой аргумент: $\lambda x.x$. Функция-константа тоже не вызывает осложнений: $\lambda x.c$, где c не зависит от x . Действительно, при любом аргументе функция возвратит c .

Рассмотрим такой пример: $\lambda x.\lambda y.x + y$. Для большей наглядности расставим скобки: $\lambda x.(\lambda y.(x + y))$ — это функция, возвращающая другую функцию, возвращающую сумму. Как это работает? Давайте применим ее к некоторому аргументу: $\lambda x.(\lambda y.(x + y))\ a = \lambda y.(a + y)$.

Получившуюся функцию применим к другому аргументу: $\lambda y.(a + y)\ b = a + b$. Такой прием называется *карринг*, или *каррирование* (в честь Хаскелла Карри), и работает для любого числа аргументов. Договоримся, если функция имеет вид $\lambda x.(\lambda y.(\lambda z.(F)))$, то мы будем записывать её как $\lambda xyz.F$. Ещё договоримся, что $(f\ a\ b\ c\ d)$ обозначает $((f\ a)b)c$. Теперь функции многих аргументов записываются очень просто: $\lambda xy.x + y$, а их применение — $f\ a\ b$.

Заметим, что мы уже не раз использовали числа и арифметические операции, несмотря на то, что говорим о бестиповом исчислении, и формально ни о каких числах и операциях пока не знаем. Это сделано для улучшения понимания, и вскоре мы узнаем, что и то, и другое в λ -исчислении можно реализовать.

Управляющие конструкции

Теперь ответим на вопрос, как в λ -исчислении сделать цикл. Так как мы имеем дело только с функциями, то очевидно с помощью механизма рекурсии. Но в λ -исчислении мы не можем вызвать функцию изнутри себя, поэтому применяется несколько иной подход — функция передается сама себе в качестве аргумента.

Рассмотрим такую функцию: $w = \lambda x.x\ x$. Она применяет свой единственный аргумент к самому себе. Попробуем применить эту функцию к самой себе:

$$w\ w = \lambda x.x\ x\ w = x\ x\ (x := w) = w\ w$$

Воспользовавшись β -редукцией, мы пришли к тому, с чего начали. Это — один из простейших примеров бесконечного цикла в λ -исчислении, примечательный тем, что не делает абсолютно ничего.

Еще один важный вопрос: как реализовать условный оператор? Здесь важно понять, что *else-ветвь* в нем не может отсутствовать. У нас все объекты — функции, которые обязаны вернуть значение, и мы не можем «ничего не делать». Определим такие объекты:

- $true = \lambda xy.x$
- $false = \lambda xy.y$

Теперь, если `bool` – логическое значение, в смысле данных определений, то `(bool if else)` – в точности нужная нам конструкция. Если `bool = true`, то результатом будет выражение `if`, иначе выражение `else`.

Попробуем решить классическую задачу на циклы и рекурсию — вычисление факториала. Пусть у нас уже реализованы числа, арифметические операции (которые мы теперь будем записывать в префиксной нотации, в стиле λ -исчисления), функция `dec`, уменьшающая свой аргумент на 1, а также функция `zero`, с помощью которой мы можем проверить число на равенство нулю.

$$g = \lambda r n. (zero\ n)1\ (\cdot n(r(dec\ n)))$$

Посмотрим внимательно: функция проверяет n на равенство нулю, возвращает 1 в случае успеха, иначе возвращает $(\cdot n(r(dec\ n)))$ — произведение n и значения некоей функции r от $n - 1$. Что такое функция r ? Это своего рода трюк, чтобы сделать рекурсию — в качестве r мы будем передавать саму функцию g , а значит, написанное выражение действительно вычисляет факториал: $g(n) = n \cdot g(n - 1)$.

Так как неудобно каждый раз передавать функции саму себя, сделаем такую окончательную реализацию: $f = g\ g$. Таким образом, g «запомнит» свой первый аргумент и останется только передать ей нужное число для вычислений.

Если мы заглянем дальше, то окажется, что в лямбда-исчислении для любой функции f существует такой аргумент x , что $f\ x = x$. Более того, этот x всегда можно найти! Эту работу выполняет так называемый комбинатор неподвижной точки. Вот один из его вариантов, предложенный Хаскеллом Карри:

$$Y = \lambda f. (\lambda x. f\ x\ x)(\lambda x. f\ (x\ x))$$

Посмотрим, как это работает:

$$Y\ f = (\lambda x. f\ x\ x)(\lambda x. f\ (x\ x)) = f(\lambda x. f\ (x\ x))(\lambda x. f\ (x\ x)) = f\ (Y\ f)$$

Таким образом, $Y\ f$ — действительно неподвижная точка функции f , так как $f\ (Y\ f) = Y\ f$.

Все приведённые выше реализации необходимых объектов и структур не единственны — прелесть λ -исчисления в том, что вы сами решаете, как реализовать базовые необходимые вещи. Но если в отношении циклов и условий вроде как выявлены самые простые реализации, то практических реализаций чисел noticeably много, для каждой задачи можно придумать свой, специфичный вариант. Тем не менее, существует общепринятая реализация чисел — «нумералы Чёрча», придуманные Алонзо Чёрчем.

Определим неотрицательное число N как функцию, принимающую некоторую функцию f , аргумент x и N раз вычисляющую значение f от x :

- $0: x$
- $1: f(x)$
- $2: f(f(x))$
- и т.д.

Таким образом, в λ -исчислении наши числа выглядят так:

- $0: \lambda f x. x$
- $1: \lambda f x. f\ x$
- $3: \lambda f x. f\ (f\ x)$
- и т.д.

Вот как, например, реализуется функция `inc`:

$$inc = \lambda n f x. f\ (n\ f\ x)$$

n раз применили функцию и применим еще раз, чтобы получить $n + 1$. Как сложить m и n ? Нужно применить f к x сначала m , а затем n раз:

$$add = \lambda m n f x. m\ f\ (n\ f\ x)$$

Умножение делается еще проще, нужно m раз применять $(n\ f)$ к x :

$$mult = \lambda m n f. m\ (n\ f)$$

Возведение в степень и того проще:

$$pow = \lambda m n. n\ m$$

Напишем функцию, проверяющую, ноль ли наш аргумент:

$$\lambda n. n\ (\lambda x. false)\ true$$

Если аргумент — ноль, то он (как вы помните из определения) просто возвратит второй аргумент, т.е. `true`, иначе он будет много (ну, как минимум, один) раз применять функцию $\lambda x. false$ к чему-то там. Вот и пригодились те самые константы, о которых мы говорили — возвращаемое значение будет `false`.

Теперь вычитание. Для начала сделаем функцию, возвращающую число на 1 меньше (или что-нибудь, если аргумент 0). Вот она:

$$\text{dec} = \lambda n f x. n (\lambda g h. h (g f)) (\lambda u. x) (\lambda u. u)$$

Вот из-за таких конструкций λ -исчисление и называют эзотерическим. Теперь даже у самых стойких сдадут нервы в связи с вопросом «как это работает?». Попробуем понять.

Для начала посмотрим на кусок $n (\lambda g h. h (g f)) (\lambda u. x)$ — нечто неведомое n раз применяется к $\lambda u. x$. Давайте уподобимся нумералам Чёрча и применим:

- Первый раз: $(\lambda g h. h (g f)) (\lambda u. x) = \lambda h. h ((\lambda u. x) f) = \lambda h. h x = \lambda h. h (0 f x)$ по определению нуля.
- Второй раз: $(\lambda g h. h (g f)) (\lambda h. h x) = \lambda h. h ((\lambda h. h. x) f) = \lambda h. h (f x) = \lambda h. h (1 f x)$ здесь нужно понимать, что h внутри скобок и h внешняя — разные переменные!
- Третий раз: $(\lambda g h. h (g f)) (\lambda h. h (f x)) = \lambda h. h ((\lambda h. h. (f x)) f) = \lambda h. h (f (f x)) = \lambda h. h (2 f x)$

Итак, мы видим, что после n применений наше выражение имеет вид:

$$\begin{aligned} \text{dec } n &= \lambda f x. (\lambda h. h ((n-1) f x)) (\lambda u. u) = \lambda f x. (\lambda u. u) ((n-1) f x) \\ &= \lambda f x. (n-1) f x = n-1 \end{aligned}$$

Но что функция вернет в случае нуля?

$$\text{dec } 0 = \lambda f x. (\lambda u. x) (\lambda u. u) = \lambda f x. x = 0$$

Заметим, что 0 и false в наших определениях совпадают.

Теперь несложно написать и вычитание:

$$\text{sub} = \lambda m n. (n \text{ dec}) m$$

В заключение поговорим о комбинаторах. Комбинаторы — это λ -функции, не содержащие свободных переменных, т.е. любая переменная в них связана λ -абстракцией. Примеры не комбинаторов:

- $\lambda x. y$
- $\lambda x y z. y z t$

Примеры комбинаторов:

- $I = \lambda x. x$ — функция, возвращающая свой аргумент
- $K = \lambda x y. x$ — генератор констант
- $S = \lambda x y z. (x y) (y z)$
- $Y = \lambda f. (\lambda x. f x x) (\lambda x. f x x)$ — комбинатор Хаскелла
- $B = \lambda x y z. x (y z)$
- $C = \lambda x y z. x y z$
- $W = \lambda x y. x y y$

И еще один интересный факт: возможно выбрать конечный набор комбинаторов, через которые можно выразить любую другую λ -функцию! Такими наборами являются, например, SKI и BCKW.

Перейдем теперь к редукционным ЭВМ. В таких машинах вычисления производятся по запросу на результат операции. Допустим, мы вычисляем $a = (b + 1) \cdot c - d/c$. В случае DF-машин, процесс начнется с внутренних операций, то есть с параллельного вычисления $b + 1$ и d/c , затем выполнится умножение и, наконец, вычитание. Такие вычисления часто называют энергичными (eager evaluation), строгими или жадными.

При обратном порядке все начинается с запроса на получение a , который последовательно приводит нас к запросу на $b + 1$, который и выполняется. Результат поступает в порядке, обратном поступлению запроса. Такой подход называют ленивыми вычислениями (lazy evaluation), отложенные или нестрогие.

Отложенные вычисления позволяют сократить общий объем вычислений за счёт тех вычислений, результаты которых не будут использованы. Программист может просто описывать зависимости функций друг от друга и не следить за тем, чтобы не осуществлялось «лишних вычислений».

В обычных, не ленивых, языках программирования аргументы функции всегда вычисляются перед выполнением самой функции³⁹. Например, в каком-то абстрактном строгом языке программирования:

```
max(5+3, 4*4) ->
```

```
max(8, 4*4) ->
```

```
max(8, 16)
```

```
16
```

А в ленивых языках, в том числе Haskell, вычисления отложенные. Все вычисления (кроме некоторых функций ввода-вывода), не выполняются сразу, а как бы откладываются до реальной надобности. Тот же пример в Haskell:

```
max (5+3) (4*4) ->
```

```
let x = 5+3; y = 4*4 in if x >= y then x else y ->
```

³⁹ Подробнее об этом, например, здесь <http://habrahabr.ru/post/131910/>

```

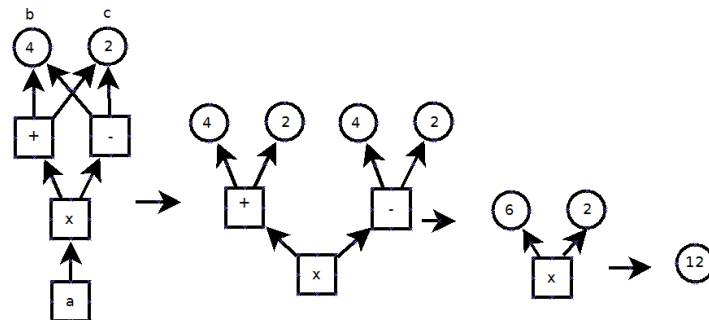
let x = 8; y = 4*4 in if 8 >= y then 8 else y ->
let x = 8; y = 16 in if 8 >= 16 then 8 else 16 ->
16

```

На этом примере хорошо видно, что вычисление подвыражений $(5 + 3)$ и $(4 \cdot 4)$ откладывалось до последнего, а именно, до момента, когда их пришлось сравнить. Здесь силой можно считать вывод на консоль, т.е. IO.

Программа, таким образом, состоит из распознавания редексов и их редуцирования до результата. Известны два типа редукционных систем – строчная и графовая, отличающиеся тем, что передается в функцию, скопированные значения данных или указатели на их положение в памяти. В строчной редукционной машине каждый узел получает отдельную копию выражения для собственной оценки. Длинные строки естественным образом редуцируются до единственного значения, каждый шаг редукции содержит оператор, сопровождаемый ссылкой на требуемые операнды. Пока входные параметры оцениваются, работа приостанавливается. В такой модели для получения результата загружается базовый граф и запускается ближайшая к требуемому значению операция. Если она невозможна из-за нехватки данных, выделяется подграф, производящий эти вычисления, и запускается уже он. Исходный граф, таким образом, редуцируется.

В графовой модели задача изначально представлена как ориентированный граф, сокращаемый по результатам оценки ветвей и подграфов. В зависимости от запросов возможна параллельная редукция. Запрошившему узлу возвращается указатель на результат редукции, обход графа продолжается, пока не будет получено значение результата.



В графовой модели при требовании рассчитать значение переменной, определение ее вычисления не копируется, а передается указатель определяющей программы. При достижении нужного узла направление переданного указателя меняется на противоположное, для запоминания места, из которого будет возвращаться результат вычисления. Таким образом, граф редуцируется, и цикл повторяется до завершения вычислений.

Несмотря на значительные усилия, не удалось установить, были ли прототипы чистых редукционных машин построены и испытаны. В принципе, к ним можно отнести процессоры, использующие в качестве внутреннего функциональные языки, типа Lisp, Haskell и подобные. Такие проекты существовали, например, инженеры EPSRC (Совет по физическим и техническим наукам Великобритании) в 2011 г. разработали Reduceron – FPGA чип, оптимизированный для выполнения функционального кода.

При тактовой частоте 96 МГц на специальных задачах он характеризуется высоким параллелизмом и показывает производительность 25% от Core 2 Duo с частотой 3 ГГц, тогда как в обычных процедурных задачах Core 2 Duo на порядок быстрее FPGA. Если сравнить с Pentium 4 2,8 ГГц, то очевидно, что код Haskell, например, быстрее выполняется на Reduceron.

Сам по себе проект носил скорее учебно-методический характер и неизвестно, имеет ли он какие-нибудь коммерческие перспективы. В принципе, Reduceron доказал возможность построения такой машины, но FPGA-матрицы тем и замечательны, что воплотить на них можно любую архитектуру, под любой язык, при этом на специфических задачах они будут показывать отличную производительность, во всех остальных же весьма печальную. Сомнительно, чтобы судьба редукционной машины в этом смысле отличалась от десятков подобных проектов.

1.5.8. Процессные алгебры и параллельные вычисления

До этого момента мы рассматривали, преимущественно, последовательные вычисления. Конечно, редукционные вычисления параллельны по своей природе, но хотелось бы рассмотреть более традиционную архитектуру. Ничто не мешает машинам Тьюринга работать параллельно, но тогда встает вопрос об удобном языке описания такой работы – включая разделение памяти, переключение контекста и прочие специфические вещи. Такой подход был разработан совсем недавно и носит название процессных алгебр.

В первой половине XX века были предложены различные способы формализации описания понятия вычислимой функции: μ -функции, машина Тьюринга, λ -исчисление. Однако для явного описания параллелизма и связей между параллельно функционирующими компонентами их либо недостаточно, либо они чрезвычайно неудобны.

Первыми, чисто параллельными моделями стали уже описанные нами сети Петри в 1962 и модель «Деятель» в 1973 году. Исследованиями процессных алгебр (ПА) начал всерьез заниматься Робин Милнер, предложив процессную алгебру «Взаимодействующих параллельных систем» (CCS – Communication concurrent processes) в период с 1973 по 1980 год. Параллельно с ним работал известный ученый Чарльз Хоар (Charles Hoare), предложивший концепцию, известную под названием «Взаимодействующие последовательные процессы» (CSP – Communicating Sequential Processes), которая впервые появилась в 1978 году и впоследствии превратилась в полноценную процессную алгебру в начале 1980-х.

Эти базовые концепции развивали и дополняли друг друга. В 1982 году Ян Бегстра и Ян Виллем Клоп начали работу над тем, что стало впоследствии известно как «Алгебра взаимодействующих процессов» (ACP), и предложили термин «процессная алгебра» для описания сути их работы. CCS, CSP, и ACP представляют собой три основные ветви в семье процессных алгебр: все остальные теории берут свои начало у этих основополагающих концепций.

В настоящее время исследования в области процессных алгебр сосредоточены в основном на следующих проблемах:

- Разработка новых процессных алгебр для более точного описания имитационного моделирования (aprrpi и π -calculus, «пи-исчисление», по аналогии с λ -исчислением).
- Конкретизация алгебр для решения определенных узкоспециализированных задач. Это может быть полезным, потому как многие из алгебр сами по себе являются достаточно обширными и общими для описывания конкретных специфичных процессов, являясь вместе с тем избыточными.
- Решение базовых задач поведенческих теорий, таких как установление эквивалентности двух процессов. Как правило, процессы могут считаться идентичными вне контекста параллельности, однако с учетом таковой могут обнаружиться несоответствия. Понятие «бисимуляция» (bisimulations) является техническим инструментом, позволяющим рассуждать об эквивалентности.
- Усовершенствование выразительности алгебр. Опыт исследований показывает, что определенные проблемы в одних алгебрах решать легче, чем в других. Этот факт требует более точной выразительности ПА, используемой для моделирования определенных процессов.

При определении процессной алгебры обычно начинают с набора каналов, которые используются для обеспечения средств коммуникации параллельных вычислителей между собой. В дополнение к каналам нужны средства для создания новых процессов.

Основные операторы, представленные в той или иной форме, позволят реализовать:

- композицию параллельных процессов;
- спецификацию, какие каналы использовать для приема и передачи данных;
- описание последовательности взаимодействий;
- сокрытие точек взаимодействия;
- рекурсию (репликацию) процессов.

Рассмотрим кратко основные термины процессных алгебр. Они тесно пересекаются в теории операционных систем, т.к. практически все современные ОС являются многозадачными и их архитектура описывается практически теми же понятиями, что и архитектура параллельных вычислителей. Более того, эти теории развивались параллельно, и многое позаимствовали друг у друга⁴⁰.

Процесс

Рассмотрим следующий пример. Два человека запускают программу извлечения квадратного корня. Один хочет вычислить $\sqrt{5}$, а второй – из $\sqrt{4}$. С их точки зрения, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. В предках современных ОС – системах пакетной обработки, существовало понятие «задание» как совокупность кода программы, набора команд внешнего языка управления заданиями, необходимых для ее выполнения, и входных данных, так что здесь мы имеем дело с двумя разными заданиями.

Пусть оба человека пытаются вычислить $\sqrt{4}$ со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно.

⁴⁰ К концу 1960-х гг. отраслью и научно-образовательным сообществом был создан целый ряд ОС, реализующих полностью или частично многозадачность. К ним относятся Atlas (Манчестерский университет), CTTS и ITSS (MIT), THE (Эйндховенский технологический университет), RS4000 (Университет Орхуса) и др. всего эксплуатировалось более сотни различных ОС. Наиболее продвинутые, такие как OS/360 (IBM), SCOPE (CDC) и частично завершённый к 1969 г. легендарный MULTICS (MIT и Bell Labs), предусматривали возможность полноценного исполнения на многопроцессорных компьютерах. Неудивительно, что к этому времени потребовалась развитая математическая теория такого исполнения.

Это происходит потому, что термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть, зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов «программа» и «задание» используется термин «процесс».

В результате под процессом понимают совокупность статичных (код программы и входные данные) и динамических (выделенный объем памяти, выделенное число ресурсов процессора и выделенные для обработки вспомогательные устройства) переменных, а также все внешние вспомогательные параметры – требующийся для запуска код ядра ОС, прерывания, исключения, привилегии и т.д.

Состояния процесса

Аксиомой теории ОС и процессных алгебр является следующее утверждение:

Все, что выполняется в вычислительных системах (не только программы пользователей, но и части операционных систем), организовано как набор процессов⁴¹.

Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди. В результате, каждый процесс может находиться как минимум в двух состояниях: **процесс исполняется, и процесс не исполняется**. Для многопроцессорных систем это тоже справедливо, ведь процессы никогда не бывают полностью изолированы – в результате одно ядро вполне может простаивать, ожидая информацию от второго, хотя такая ситуация и нежелательна.

Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние «процесс не исполняется» на два новых состояния: **готовность и ожидание**.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимание на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: **рождение и уничтожение**.

В конкретных операционных системах состояния процесса могут быть еще более детализованы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9. Тем не менее, все операционные системы подчиняются изложенной выше модели.

Операции над процессами

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними одноразовые (приводят к изменению количества процессов, находящихся под управлением ОС, и всегда связаны с выделением или освобождением ресурсов, это пара создание процесса – завершение процесса) и многократные (приостановка/запуск блокирование/разблокирование повышение/понижение приоритета) операции.

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура (Process Control Block) содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблица открытых файлов).

⁴¹ С точки зрения грамотной организации архитектуры операционной системы допускаются лишь две абстракции – процесс и файл. Все, что из себя представляет ОС, все ее механизмы и внутренние части должны описываться схемой «процесс применен к файлу». Так, например, была организована MULTICS и также организованы ее упрощенные и облегченные наследники – ОС семейства UNIX, включая Linux.

Состав PCB зависит от конкретной ОС. Информация, для хранения которой он предназначен, называется **контекстом процесса**. Обычно она делится на две неравные части:

- содержимое всех регистров процессора (включая значение программного счетчика) будем называть **регистровым контекстом процесса**,
- все остальное – **системным контекстом процесса**.

Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т.е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты.

Код и данные, находящиеся в адресном пространстве процесса, будем называть его **пользовательским контекстом**. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

Помимо этих основных сведений, существует подробно разработанная теория создания процессов (например, если процесс инициировал другой процесс, т.н. наследник, исчезнет ли он с завершением родительского и т.п.⁴²); теория переключения процессов и их планирования (например, способы уменьшения накладных расходов на переключение контекста и т.п. вопросы); теория связи процессов между собой (каналы связи, разделяемая память, легковесные процессы-нити с общим системным контекстом и т.п.) и, наконец, методы борьбы с тупиками, дедлоками и прочими неприятностями (семафоры Дейкстры, мониторы и т.п.). Тем не менее, этой информации уже достаточно, чтобы продолжить наш разговор о математическом описании всего этого хозяйства.

Наибольшее распространение получили следующие ПА:

ПА «взаимодействующих параллельных систем» (CCS) – представляет собой процессную алгебру, разработанную Робинот Милнером около 1980 г. Официальный язык включает примитивы для описания параллельных компонент и взаимодействия между ними, а также средства оценки правильности качественных свойств системы, таких как deadlock или livelock.

ПА «взаимодействующих последовательных процессов» (CSP) – была впервые описана в статье 1978 Р. Хоара, но с того времени существенно развивалась. CSP нашла практическое применение в промышленности в качестве инструмента для определения и подтверждения параллельных аспектов различных систем – таких как T9000 Transputer и безопасность системы электронной торговли. Теория CSP все еще является объектом для исследования, включая работы по расширению сферы ее применения (например, увеличение масштабов систем, которые можно проанализировать).

ПА «взаимодействующих процессов» (ACP) – это также алгебраический подход к описанию параллельных систем, разработанный Яном Бергстрой и Яном Вильямом Клопом в 1982 г. Термин «процессная алгебра» был взят именно из этого подхода, предполагающего более четкое аксиоматическое описание понятия процесса и операций над ним.

Сравним эти три базовые концепции в плане выразительности алгебр. Пусть задано множество имен процессов P : P_1 и P_2 . Тогда сравнительная характеристика процессных алгебр CCS, CSP и ACT будет выглядеть следующим образом:

Оператор	CCS	CSP	ACT
Действие: P_1 выполняет действие a и продолжает быть тем же P_1	$a.P_1$	$a \rightarrow P_1$	$a \cdot P_1$
Выбор: процесс может вести себя, как P_1 или P_2	$P_1 + P_2$		
Параллельная композиция: P_1 и P_2 существуют одновременно	$P_1 P_2$	$P_1 P_2$	$P_1 \parallel P_2$
Взаимодействие: обмен сообщениями между P_1 и P_2	$x(y) \cdot P x(v) \cdot Q \rightarrow P Q[y/v]$	$r(d) \cdot P w(d) \cdot Q$	-----
Ограничение: процесс P_1 не может осуществить действие a	$P_1 \backslash a$		$\tau_{\{a\}} P_1$
Переименование: действие a в P_1 переименовано в b	$P_1[b/a]$	-----	
Определение дедлока δ:	$\begin{cases} \delta + P = P \\ \delta.P = \delta \end{cases}$	$\begin{cases} \delta + P = P \\ \delta \rightarrow P = \delta \end{cases}$	$\begin{cases} \delta + P = P \\ \delta \cdot P = \delta \end{cases}$

⁴² Казалось бы, не обязательно, но в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его наследников.

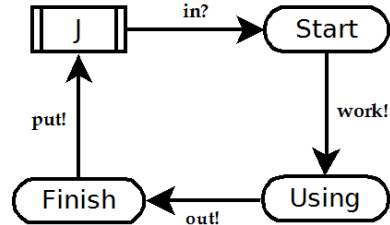
Как видим, практически все заданные ПА поддерживают базовый набор операторов для описания параллельных процессов со средствами коммуникации между ними. Многообразие существующих процессных алгебр велико (например, варианты, включающие средства описания стохастического поведения, временной информации и специализации для изучения молекулярных взаимодействий), но все они имеют определенный набор схожих черт:

- представление взаимодействия между независимыми процессами с помощью передачи сообщений, а не с помощью модификации общих переменных;
- описания процессов и систем с помощью небольшой коллекции примитивов и операторов для объединения этих примитивов;
- определение алгебраических законов для операторов процесса, которые позволяют манипулировать процессными выражениями, используя принцип эквивалентности.

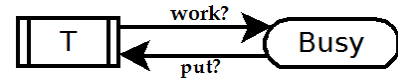
Отметим, что для формального описания процессов также хорошо подходят графы, как в теории абстрактных автоматов. Далее можно определить семантику процессов, достижимые и недостижимые состояния для процесса, механизмы их взаимодействия и т.д.

Вот пример элементарной абстракции, описанной алгеброй CCS. Допустим, у нас имеется столярная мастерская с несколькими рабочими, каждый из которых работает каким-либо инструментом над каким-нибудь изделием.

Поведение рабочего можно описать процессом J (от Jobber – рабочий):



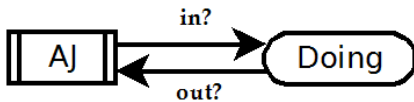
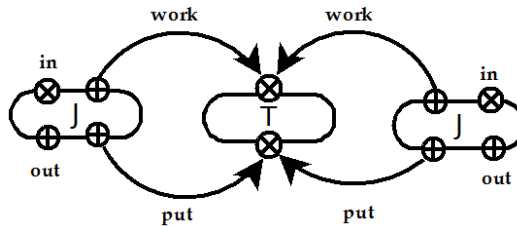
Здесь символ «?» употребляется по отношению к входящему действию, символ «!» - по отношению к исходящему. Поведение инструмента мы представим с помощью процесса T (Tool):



Отметим, что объект T и процесс T - разные понятия. Функционирование мастерской определяется при помощи процесса W (workshop):

$$W = (J|J|M)\backslash L,$$

где $L = \{work, put\}$. Потокный граф процесса W имеет вид:



Введем понятие «абстрактного рабочего», про которого известно, что он циклически принимает материал и выдает готовые изделия. Его поведение задается процессом AJ (Abstract Jobber):

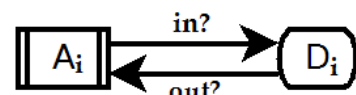
Поведение абстрактной мастерской зададим процессом AW :

$$AW = AJ|AJ$$

Абстрактную мастерскую мы будем использовать как спецификацию мастерской. Этот процесс передает поведение мастерской без учета деталей ее реализации⁴³. Теперь мы можем делать различные выводы о нашей мастерской, интересные с точки зрения анализа ее работы в будущем. Для начала докажем соответствие мастерской ее спецификации, или, в терминах процессных алгебр, конгруэнцию

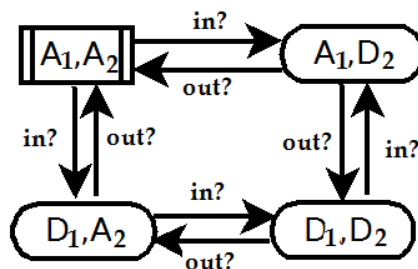
$$W \approx AW$$

Процесс AW является параллельной композицией двух процессов AJ . Для предотвращения коллизий выберем разные идентификаторы для обозначения состояний этих процессов, пусть, например:



⁴³ Такой подход очень характерен для ООП.

где $i = 1, 2$. Параллельная композиция процессов имеет вид:



Для дальнейшего сравнения процессов проводят т.н. минимизацию, т.е. находят процесс, эквивалентный данному, но с наименьшим числом состояний. Не углубляясь в детали этой процедуры⁴⁴, отметим, что минимальный процесс для вышеуказанного имеет вид:



Процесс W имеет $4 \cdot 4 \cdot 2 = 32$ состояния, поэтому здесь мы его не приводим. Однако после минимизации мы получим процесс, изоморфный минимальному для AW , что и требовалось доказать. Исследование эквивалентности процессов и их спецификаций чрезвычайно важно для отыскания ответов на вопросы по типу, достигнута ли заданная функциональность системы, или исследования отсутствия тупиков, дедлоков и т.п., анализа эффективности использования вычислительных ресурсов и прочего, что роднит процессные алгебры с доказательным программированием.

На этом мы закончим краткое введение в вопрос и в заключение приведем небольшую классификацию параллельных архитектур вычислительных машин. Желая углубиться в вопрос, рекомендуется поискать информацию также по сетям Петри, темпоральной логике и проверке моделей, теории частичных порядков, теории трасс и взаимодействующим абстрактным автоматам, семантике языков параллельного программирования.

Отметим также, что теория процессов и процессные алгебры нашли не только широкое применение в проектировании и анализе параллельных архитектур, но и в создании языков для них, таких, как Оссам (являлся фактически языком ассемблера для систолических матриц на транспьютерах INMOS, популярных в 1980-е г.); Unified Parallel C (расширение C, применяемое для эффективного программирования NUMA-машин, таких как станции SGI в середине и конце 1990-х г.); X10 (разработан IBM в 2004 г., как параллельное расширение Java для эффективного использования в мейнфреймах); Fortress (его конкурент от Sun); Chapel (язык суперкомпьютеров Cray); Intel Cilk Plus (новое расширение C++ от Intel, 2011 г.); знаменитый Erlang и др.

Параллельные же машины исторически развивались примерно так:

1. Попытки ввести псевдопараллельность (один процессор с различными комбинациями нескольких АЛУ и памяти):

1) Дискретные процессоры на интегральных элементах. Простейший вид ВМ, развивавшийся в 1950-1960 гг., некоторые устаревшие модели выпускались вплоть до 1970-х:

1.1) Классические последовательные машины с разрядно-параллельной памятью, умеющие последовательно обрабатывать слова (IBM 701, 1952 г.; серия DEC PDP, первый компьютер PDP-1 выпущен в 1960-м г., ее преемник DEC VAX11/780, 1977 г.);

1.2) Машины с классическим конвейером (БЭСМ-6, 1957 г., Ferranti ATLAS, 1962 г.);

1.3) Конвейерные скалярные машины с несколькими функциональными устройствами в конвейере (IBM 360/91, 1964 г.; Control Data Corporation CDC 6600, 1964 г.; Эльбрус-1, 1979 г.).

2) Машины на процессорных платах. Развивались в 1970-1980 гг., пришли на смену дискретным ЭВМ, в основном относились к суперкомпьютерам:

2.1) Конвейерно-векторные машины с независимыми конвейерными функциональными устройствами, последовательно обрабатывающими слова при параллельной обработке разрядов. К этой категории относились CDC-7600 (1971 г.), CDC CYBER 205 (1981 г.), CRAY-1 (1982 г.), Denelcor Heterogeneous Element Processor (1982 г.), Fujitsu FACOM VP-400 (1985 г.). Все они являлись суперкомпьютерами, рекордными для своего времени;

2.2) Ассоциативные SIMD-машины – последовательная обработка битовых слоев при параллельной обработке множества слов (Goodyear STARAN, 1972 г.; ILC Distributed Array Processor, 1979 г.; Goodyear

⁴⁴ Подробности можно найти, например, в откуда и позаимствован, с незначительными изменениями, данный пример.

Aerospace MPP, 1983 г.). Эти машины были поменьше размером и занимали нишу высокопроизводительных коммерческих ЭВМ или были экспериментальными разработками.

2. Попытки увеличить число полных вычислительных блоков (процессорные матрицы и машины на микропроцессорах):

1) Многопроцессорные системы на дискретных элементах. Исторически первые попытки собрать матричные вычислители относятся к 1960-1970 гг.:

1.1) Ортогональные машины (Sanders Associates OMEN-60, 1970-е г.) В серии OMEN-60 использовалась машина PDP-11 в качестве обычного горизонтального арифметического устройства и матрица на 64 процессорных элементах в качестве вертикального ассоциативного арифметического устройства, которое работало с байтовыми, а не разрядными секциями. Между процессорными элементами встраивалась логика для изменения последовательности байтов в секции на обратную или для выполнения полной тасовки или циклического сдвига;

1.2) Машины на параллельных процессорных элементах (Burroughs Corporation PEPE, начало 1970-х г.). Эти машины послужили прототипом для следующих в нашем списке;

1.3) Классические процессорные матрицы (Burroughs Corporation ILLIAC IV, 1972 г.; Texas Instruments ASC, 1973 г.);

1.4) Системные массивы, например, INMOS, iWarp и пр. Возможно, первой систолической машиной был Colossus Mk II 1944 г.

2) Многопроцессорные системы на микропроцессорах. Современные версии многопроцессорных систем собираются на микропроцессорах и имеют 4 основных архитектуры, в зависимости от организации разделяемой памяти:

2.1) SMP (Symmetric Multiprocessing), архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессоров подключаются к общей памяти. Большинство бюджетных многопроцессорных систем (такие как все сервера начального уровня – Dell, Tyan, Supermicro, Intel, HP, IBM, Apple и пр.) сегодня используют архитектуру SMP. SMP-системы позволяют любому процессору работать над любой задачей независимо от того, где в памяти хранятся данные для этой задачи; при должной поддержке операционной системой SMP-системы могут легко перемещать задачи между процессорами, эффективно распределяя нагрузку. С другой стороны, память гораздо медленнее процессоров, которые к ней обращаются; даже однопроцессорным машинам приходится тратить значительное время на получение данных из памяти. В SMP ситуация ещё более усугубляется, так как только один процессор может обращаться к памяти в данный момент времени. В результате масштабируемость SMP-систем невелика: 2-16 процессоров, чаще всего 4;

2.2) UMA (Uniform Memory Access) Все микропроцессоры в UMA-архитектуре используют физическую память одновременно. При этом время запроса к данным из памяти не зависит от того, какой именно процессор обращается к памяти, или какой именно чип памяти содержит нужные данные. Однако каждый микропроцессор может использовать свой собственный кэш. Примером являются SPARC-серверы, например, Sun Enterprise 10000;

2.3) NUMA (Non-Uniform Memory Access) — неравномерный доступ к памяти или Non-Uniform Memory Architecture, архитектура с неравномерной памятью. В этой схеме коммутации вычислительных элементов время доступа к памяти определяется её расположением по отношению к процессору. NUMA с когерентностью кэш-памяти. Существуют различные разновидности NUMA-машин, масштабируемость их достигает 1000 и более процессоров. Классическим примером являлись графические станции на MIPS-процессорах ныне почившей корпорации SGI, например, Origin2000;

2.4) MPP (massive parallel processing), массивно-параллельная архитектура с физически разделенной памятью. Система строится из отдельных узлов, содержащих процессор, локальный банк оперативной памяти, коммуникационные процессоры или сетевые адаптеры, иногда — жёсткие диски и другие устройства ввода-вывода. Доступ к банку оперативной памяти данного узла имеют только процессоры из этого же узла. Узлы соединяются специальными коммуникационными каналами. Одним из первых представителей был C.mpp, содержащий до 16 машин типа DEC PDP-11, связанных с 16 модулями памяти через перекрестный переключатель размерности 16x16 и Cray T3E 1996 года. Большая часть современных суперкомпьютеров, например, IBM серии Z, строится именно так.

2.5) Системы с большим числом ЭВМ – сети, кластеры и миникластеры: RenderCube, Orion Multi-systems, T-Forge Mini, T-Edge Mini, Cray CX1, Thinking Machines CM-1, Beowulf, Ломоносов и др.

3. Микропроцессоры

1) Псевдопараллельные (с одним полнофункциональным элементом):

3.1) Суперскалярные процессоры (Intel I960, 1985 г.; Sun SPARC, 1992 г.; Motorola m88k, 1992 г.; Intel Pentium, 1993 г.; MIPS R800, 1994 г.). Здесь не упоминаются все суперскалярные процессоры – очевидно, что эта технология сейчас присутствует повсеместно, упомянуты только родоначальники, те, в которых она отвечала за большую часть производительности;

3.2) Конвейерно-векторные процессоры (CRAY EL90, 1993 г.; CRAY T90, 1995 г.; NEC SX-6, 2001 г.; HPC Server SX-8i, 2006 г.; CRAY X1, 2003 г.; Fujitsu-VPP5000, 2000 г.). Здесь речь идет не о чистых процессорах, т.к. они никогда не поставлялись на свободный рынок, а о суперкомпьютерах, на них собранных;

3.3) Многопоточные процессоры (Intel Pentium 4, 2000 г.) Конечно, технология Hyper Threading применялась не только в нем, но это, аналогично суперскалярности, наиболее чистый представитель семейства;

3.4) VLIW и EPIC процессоры (Transmeta Crusoe, 1999 г.; Intel Itanium, 2001 г.; Эльбрус E2k, 2012).

2) Полностью параллельные (многоядерные):

2.1) Many-Core System – системы с небольшим числом полнофункциональных мощных ядер, такие, как Intel Core и Xeon, AMD, IBM Power, ARM11 MPCore, Cortex-A9 MPCore, MIPS Computer Systems MIPS32 1004K, Sun Microsystems Ultra Sparc T2;

2.2) Multi-Core System – системы с большим числом параллельных микроядер, такие как NVIDIA G80, AMD ATI, ClearSpeed Advance X620, Tiler Tile64Pro, ClearSpeed CSX700, STMicroelectronics AsAP-II, Rapport KC-256, IBM Cell, Intel Xeon Phi.



БЭСМ-6



CDC 6600



Процессорные платы Cray-1 в частной коллекции, США



ILLIAC IV



Titan, суперкомпьютер по типу MPP, построенный на серверах Cray XK7 для Oak Ridge National Laboratory в 2012 г.

Подробнее архитектуру параллельных машин мы рассмотрим далее, в следующей части нашей работы.

II. ФИЗИКА

2.1. Аналог и цифра

2.1.1. Сигналы и их представление

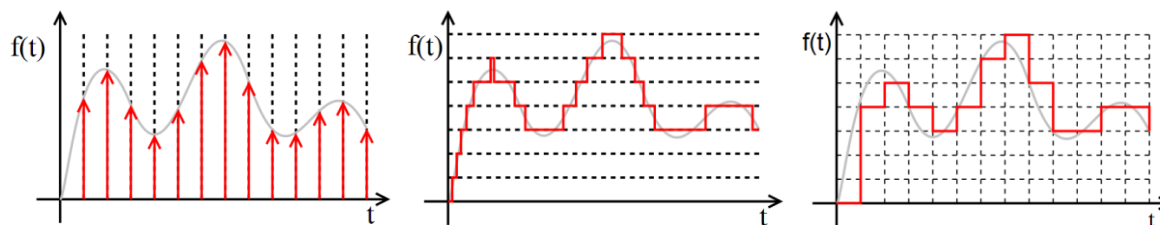
С точки зрения физики, сигнал – любая физическая величина, изменяющаяся со временем. С точки зрения информатики, все, что изменяется со временем, переносит информацию. Сигнал может генерироваться, но его приём не обязателен, в отличие от сообщения, которое рассчитано на принятие принимающей стороной, иначе оно не является сообщением.

Сигнал, детерминированный, или случайный, описывают математической моделью, характеризующей изменение его параметров. В технике противоположностью сигналу, который несёт полезную информацию, является шум – обычно случайная функция времени, взаимодействующая с сигналом и искажающая его.

Понятие сигнал позволяет абстрагироваться от конкретной физической величины и рассматривать вне физического контекста явления, связанные с кодированием информации и извлечением её из сигналов. В исследованиях сигнал часто представляется функцией времени, параметры которой могут нести нужную информацию. Способ записи этой функции, а также способ записи мешающих шумов называют математической моделью сигнала.

С понятием сигнала формулируются такие базовые принципы кибернетики, как пропускная способность канала связи или оптимальный прием. В зависимости от функции, описывающей параметры сигнала, выделяют аналоговые, дискретные, квантованные и цифровые сигналы:

- непрерывные (аналоговые), описываемые непрерывной функцией;
- дискретные, описываемые функцией отсчётов, взятых в определённые моменты времени;
- квантованные по уровню;
- дискретные сигналы, квантованные по уровню, называются цифровыми сигналами.



Сигналы дискретный, квантованный и цифровой

Аналоговый сигнал — сигнал данных, у которого каждый из представляющих его параметров описывается функцией времени и непрерывным множеством возможных значений. Различают два пространства сигналов – пространство L (непрерывные сигналы) и пространство l – пространство последовательностей.

Пространство l есть пространство коэффициентов Фурье (счётного набора чисел, определяющих непрерывную функцию на конечном интервале области определения), пространство L есть пространство непрерывных по области определения (аналоговых) сигналов. При некоторых условиях, пространство L однозначно отображается в пространство l (например, первые две теоремы дискретизации Котельникова).

Аналоговые сигналы описываются непрерывными функциями времени, поэтому аналоговый сигнал иногда называют континуальным. Свойства аналоговых сигналов в значительной мере являются противоположностью свойств цифровых сигналов.

- Отсутствие чётко отличимых друг от друга дискретных уровней сигнала не позволяет применить для его описания понятие информации в том виде, как она понимается в цифровых технологиях. Содержащееся в одном отсчёте «количество информации» будет ограничено лишь динамическим диапазоном средства измерения.

- Отсутствие избыточности. Из непрерывности пространства значений следует, что любая помеха, внесенная в сигнал, не отличима от самого сигнала и, следовательно, исходная амплитуда не может быть восстановлена. В действительности фильтрация возможна, например, частотными методами, если известна какая-либо дополнительная информация о свойствах этого сигнала (в частности, полоса частот).

Цифровой сигнал — сигнал данных, у которого каждый из представляющих параметров описывается функцией дискретного времени и конечным множеством возможных значений. Сигналы представляют собой дискретные электрические или световые импульсы. Узкополосные системы (baseband) передают данные в виде цифрового сигнала одной частоты.

Дискретный цифровой сигнал сложнее передавать на большие расстояния, чем аналоговый, но использование в цифровых системах алгоритмов проверки и восстановления цифровой информации решает эту проблему. Из-за шумов и изменения параметров линий передачи цифровой сигнал имеет флуктуации по амплитуде, фазе/частоте (джиттер), поляризации.

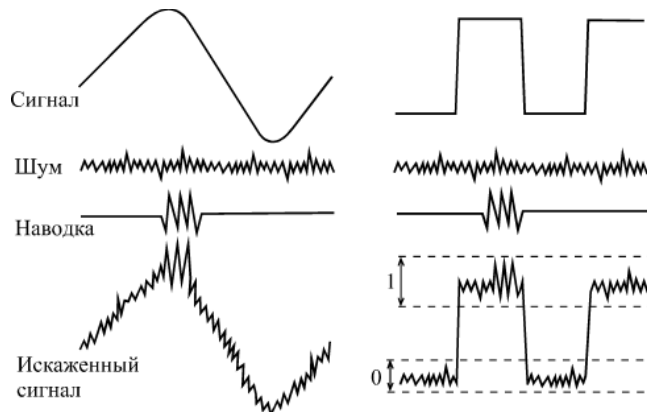
Важным свойством цифрового сигнала, определившим его доминирование в современных системах связи, является его способность к полной регенерации вплоть до некоторого порогового отношения сигнал/шум, в то время как аналоговый сигнал удаётся лишь усилить вместе с наложившимися на него шумами. Здесь же кроется и недостаток цифрового сигнала: если цифровой сигнал утонет в шумах, восстановить его невозможно (эффект крутой скалы), в то время как человек (не машина) может усвоить информацию из сильно зашумлённого сигнала на аналоговом радиоприёмнике, хотя и с трудом.

Если, например, сравнивать сотовую связь аналогового формата (AMPS, NMT) с цифровой связью (GSM, CDMA), то при помехах на цифровой линии из разговора выпадают порой целые слова, а на аналоговой можно вести разговор, хотя и с помехами.

Квантованием в информатике называют разбиение диапазона значений непрерывной или дискретной величины на конечное число интервалов. Существует также векторное квантование – разбиение пространства возможных значений векторной величины на конечное число областей. Простейшим видом квантования является деление целочисленного значения на натуральное число, называемое коэффициентом квантования.

Не следует путать квантование с дискретизацией (и, соответственно, шаг квантования с частотой дискретизации). При дискретизации изменяющаяся во времени величина (сигнал) замеряется с заданной частотой (частотой дискретизации), таким образом дискретизация разбивает сигнал по временной составляющей. Квантование же приводит сигнал к заданным значениям, то есть, разбивает по уровню сигнала. Сигнал, к которому применены и дискретизация, и квантование, называется цифровым. Квантование часто используется при обработке сигналов, в том числе при сжатии звука и изображений.

При оцифровке сигнала количество уровней квантования называют также глубиной дискретизации или разрядностью. Глубина дискретизации измеряется в битах и обозначает количество бит в двоичном слове, выражающих амплитуду сигнала. Чем больше глубина дискретизации и чем больше частота дискретизации, тем точнее цифровой сигнал соответствует аналоговому. В случае однородного квантования глубину дискретизации называют также динамическим диапазоном и измеряют в децибелах (1 бит $\approx$ 6 дБ).



Все операции, производимые электронными устройствами над сигналами, можно условно разделить на три большие группы:

- обработка (или преобразование);
- передача;
- хранение.

Во всех этих трех случаях полезные сигналы искажаются паразитными – шумами, помехами, наводками. Кроме того, при обработке сигналов (например, при усилении, фильтрации) еще и искажается их форма. А при передаче на большие расстояния и при хранении сигналы к тому же ослабляются.

Надо учесть, что все шумы, помехи и наводки принципиально не поддаются точному расчету, поэтому точно описать поведение любых аналоговых устройств абсолютно невозможно. К тому же со временем параметры всех аналоговых устройств изменяются из-за старения элементов, поэтому характеристики этих устройств не остаются постоянными.

Поведение цифровых устройств всегда можно абсолютно точно рассчитать и предсказать. Цифровые устройства гораздо меньше подвержены старению, так как небольшое изменение их параметров никак не отражается на их функционировании, кроме того, цифровые устройства проще проектировать и отлаживать.

Однако у цифровых сигналов есть и крупный недостаток. Дело в том, что на каждом из своих разрешенных уровней цифровой сигнал должен оставаться хотя бы в течение какого-то минимального временного интервала, иначе его невозможно будет распознать, аналоговый же сигнал может принимать любое свое значение бесконечно малое время.

Благодаря этому максимально достижимое быстродействие аналоговых устройств всегда принципиально больше, чем цифровых. Кроме того, цифровой сигнал передает информацию только двумя уровнями и изменением одного своего уровня на другой, а аналоговый – еще и каждым текущим значением своего уровня, то есть он более емкий с точки зрения передачи информации. Поэтому для передачи того объема информации, который содержится в одном аналоговом сигнале, чаще всего приходится использовать несколько цифровых (чаще всего от 4 до 16).

К тому же в природе все сигналы – аналоговые и для преобразования их в цифровые и обратно требуется применение специальной аппаратуры (АЦП и ЦАП).

2.1.2. Аналоговые компьютеры

Уникальные и массовые, быстрые, но неточные, моделирующие плотины ГЭС и рассчитывающие первую атомную бомбу – это все о них. Аналоговые компьютеры отправляли в космос Гагарина, управляли первыми ядерными реакторами и создавали звук в музыкальных синтезаторах. Что представляют собой эти самые необычные и практически забытые⁴⁵ вычислительные машины?

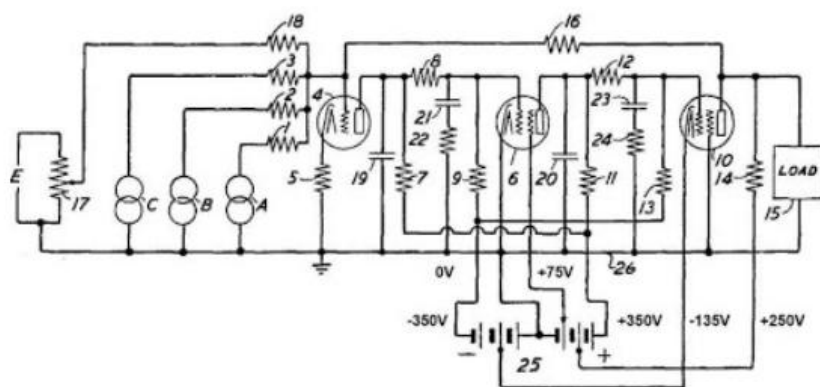
Если кратко, то АВМ – устройство, представляющее данные напрямую при помощи соответствующей модели с помощью аналоговых физических переменных (скорость, длина, напряжение, ток, давление и т.п.).

Развитие аналоговых компьютеров в США и СССР шло параллельно, и главные шаги были сделаны перед Второй мировой войной. Одну из первых и самую знаменитую АВМ построил в 1930 г. Ванневар Буш – это была механическая интегрирующая машина, применявшая при расчётах траектории стрельбы корабельных орудий. В 1942 г. была создана её электромеханическая версия, участвовавшая в Манхэттенском проекте и долгое время работавшая вместе с ENIAC.

В 1935 г. в СССР выпустили электродинамическую счётно-аналитическую машину САМ Т-1, затем были разработаны механический интегратор и электрический расчётный стол для определения стационарных режимов энергетических систем.

В 1942-44 гг. в США был разработан операционный или т.н. «решающий» усилитель – усилитель постоянного тока, имеющий весьма высокий коэффициент усиления. Это позволило создавать аналоговые компьютеры без движущихся частей на постоянном токе. В СССР в 1945-46 гг. под руководством Гутенмахера были созданы первые электронные аналоговые машины с повторением решения. В 1949 году в СССР был построен целый ряд АВМ на постоянном токе – для создания советской атомной бомбы требовалось огромное количество вычислений. Эти работы положили начало развитию аналоговой вычислительной техники в СССР.

В 60-х годах аналоговые компьютеры уже могли совершать расчеты с точностью до одной десятичной. Это было невероятным успехом, и они на какое-то время стали повседневным инструментом ученых для решения множества специфических задач (таких, как расчет запасов нефти и газа и прогнозирование погоды). В Советском Союзе серийно выпускалось 9 типов электронных аналоговых вычислительных машин, среди них были такие знаменитые АВМ, как серия ИГ (гидравлических интеграторов), так же, по традиции, как и во всех странных архитектурах, отличились поляки.



A) Schematic diagram

R1, R2, R3, R13, R16	1megΩ
R5	1.5kΩ
R7	0.5megΩ
R8	0.75megΩ
R9	2megΩ
R11	0.25megΩ
R12	1.25megΩ
R14, R15 (load)	6kΩ
R18	3megΩ
R22, R24	100kΩ
C19	0.25μF
C20	100pF
C21	1000pF
C23	0.03μF
V4	6SC7
V6	6SJ7
V10	6Y6G

B) Component values

Суммирующий усилитель, Bell Labs, 1941 г., схема и список компонентов

АВМ могут выполнять такие операции, как сложение, вычитание, умножение, деление, а также напрямую аппаратно производить дифференцирование, интегрирование и инвертирование, на что не способна в принципе ни одна ЭВМ. При работе аналоговый компьютер строит физическую модель задачи из стан-

⁴⁵Не совсем, конечно, например, автомобильная автоматическая трансмиссия является примером гидромеханического аналогового вычислителя, в котором при изменении вращающего момента жидкость в гидроприводе меняет давление, что позволяет получить изменение коэффициента передачи, маленькие АВМ окружают нас везде – аналоговая обработка электрических сигналов занимает важное место в промышленной электронике. Большинство типов первичных преобразователей физических величин являются источниками аналоговых сигналов, а многие исполнительные элементы в объектах управления управляются непрерывно изменяющимся электрическим током. Даже системы управления, основой которых являются цифровые вычислительные комплексы, не могут отказаться от аналоговой обработки сигналов и сопрягаются с объектами управления и датчиками с помощью аналоговых и аналого-цифровых устройств. Однако, все эти машины монозадачны, малы и очень отдалены от конечного пользователя, строго говоря, их даже компьютерами не назовешь – скорее вспомогательными механизмами, а АВМ на столах ученых, заменяющие традиционный компьютер, уже давно отошли в прошлое.

дартных блоков, а затем запускает соответствующий процесс – протекание жидкости, электрического тока и т.п., после чего на выходах сразу появляется решение также в аналоговой форме – в виде показаний приборов, сигналов осциллографа, распечаток или даже интегральных кривых решенных уравнений, форма которых задается столбиками воды в специальных сосудах.

Эти компьютеры идеально приспособлены для осуществления автоматического контроля над производственными процессами, потому что они моментально реагируют на различные изменения во входных данных. В ряде случаев с помощью аналоговых компьютеров можно решать задачи, намного меньше заботясь о точности вычислений, чем при написании программы для цифровой ЭВМ. Например, для электронных аналоговых компьютеров без проблем реализуются задачи, требующие решения дифференциальных уравнений, интегрирования или дифференцирования. Для каждой из этих операций применяются специализированные схемы и узлы, обычно с применением операционных усилителей. Также интегрирование легко реализуется и на гидравлических аналоговых машинах.

Все типовые АВМ также строятся на стандартных блоках, которые можно разделить на ряд групп:

- линейные — выполняют такие математические операции, как интегрирование, суммирование, перемена знака, умножение на константу;
- нелинейные (функциональные преобразователи) – соответствуют нелинейной зависимости функции от нескольких переменных;
- логические – устройства непрерывной, дискретной логики, релейные переключающие схемы. Вместе эти устройства образуют устройство параллельной логики.

Универсальные АВМ, как правило, содержат в своем составе:

- источник питания,
- контрольно-измерительную аппаратуру,
- управляющее устройство,
- наборное поле,
- блоки суммирования (сумматор),
- блоки интегрирования (интегратор),
- блоки дифференцирования (дифференциатор),
- множително-делительное устройство,
- блоки нелинейности (функциональный преобразователь).

Также используются:

- потенциометр функциональный,
- блок переменных коэффициентов,
- вычислитель индукционный,
- тахогенератор,
- масштабное звено — аналоговый функциональный блок в АВМ структурного типа, в котором выходная и входная величина связаны линейно. Применяется, когда в АВМ при реализации структурной схемы модели необходимо произвести умножение на постоянный коэффициент. В качестве звена масштабирования может применяться блок суммирования.

В качестве запоминающих устройств используют:

- Ёмкостные запоминающие устройства — динамические запоминающие устройства, основанные на свойстве конденсаторов хранить поданное на них напряжение. Ячейка ёмкостного ЗУ формируется на обычном интеграторе с различными коммутаторами. Иногда в интегратор для уменьшения времени процесса запоминания вводится операционный усилитель – повторитель. Время хранения информации в таких устройствах ограничено – это ОЗУ аналоговых машин;
- Делитель напряжения – электромеханическое запоминающее устройство, в котором запоминаемым величинам соответствуют углы поворота реостатов. Подобные устройства могут неограниченное время хранить информацию;
- Запоминающая пара – устройство, формирующее задержанную во времени последовательность выбранных уровней входного сигнала. В качестве запоминающей пары часто применяют каскадно-соединённые операционные усилители, один из которых работает в режиме отслеживания входного сигнала, а другой – в режиме хранения;
- Запоминающее устройство на ферритовых сердечниках – основано на свойстве ферромагнетиков сохранять намагниченность. Ячейки таких ЗУ выполняются на ферритовых сердечниках либо на трансфлюксорах и тороидальных сердечниках. Использование трансфлюксоров и тороидальных сердечников уменьшает погрешности, одновременно снижая быстродействие.

В качестве основной характеристики АВМ применяли т.н. добротность. Это величина, рассчитывалась так:

$$d = (E_{max} - E_{min}) / E_{min},$$

где E_{max} – максимально возможное значение машинной переменной, E_{min} – нижний предел возможного значения машинной переменной. Пределы, как правило, определяются экспериментально. Числовое значение зависит от уровня помех, ошибок аналоговых функциональных блоков, точности применяемой измерительной аппаратуры. Добротность мощных АВМ превышала 10^3 .

По принципу работы выделяли:

АВМ механическая. В такой машине переменные воспроизводятся механическими перемещениями. При решении задач на АВМ данного типа необходимо, кроме масштабирования переменных, производить силовой расчет конструкции и расчет мертвых ходов. Достоинством механических АВМ является обратимость, позволяющая воспроизводить прямые и обратные математические операции. Недостатки АВМ такого типа — высокая стоимость, сложность изготовления, большие габариты и вес, а также низкий коэффициент эффективности использования отдельных вычислительных блоков. Простейший вариант — известная всем логарифмическая линейка.

АВМ пневматическая. Один из немногих типов аналоговых машин, ограниченно применяющийся сейчас. В такой машине переменные представлены в виде величин давления газа в разных точках сети, элементами машины являются дроссели, мембраны и емкости. Дроссели играют роль сопротивлений, они могут быть постоянными, переменными, нелинейными, регулируемые. Емкости представляют собой глухие или проточные камеры, мембраны используют для преобразования давления. Скорость работы таких АВМ достигала неплохих для механики показателей 10кГц. Такие АВМ отличались большой погрешностью и потому применялись и ограниченно используются сейчас в управлении устройствами там, где нельзя допускать, даже теоретически, возможность замыкания или искр — взрывоопасных и агрессивных средах, на химических производствах, в металлургии и энергетике.

В 1960-х годах они разрабатывались для получения средства дискретных вычислений с высокой радиационной стойкостью. Были разработаны элементы, выполняющие основные логические операции и элементы памяти вообще без механических подвижных частей. Такие машины очень долговечны, поскольку в них практически отсутствуют подвижные части и, как следствие, нечему ломаться. В случае засорения каналов логические матрицы легко разбираются и промываются. Работает пневмокомпьютер от промышленной пневмосети. Логические матрицы легко штампуются на термопласт-автоматах из пластика. Для особых случаев матрица может быть изготовлена из тугоплавкой керамики, отлита из чугуна или другого сплава.

Сейчас пневмокомпьютеры используются в отраслях промышленности, где требуется повышенная вибрационная стойкость, работоспособность в очень широком диапазоне температур или требуется управление пневматическими силовыми устройствами. В последнем случае устраняется необходимость в преобразователях электрического сигнала в перемещение (электро-пнеумопреобразователь + позиционер). Это роботы и автоматика, работающие в металлургии, в горнорудной промышленности. Известны случаи управления элементами авиационных двигателей, автоматикой ракетных систем, силовыми приводами вертолетов и самолетов.

Существует также целая категория производств, агрегатов и установок, где применение электричества, даже самых низких напряжений, очень нежелательно. Это химия органических соединений, нефтеперегонные заводы, подземная добыча угля и руды. Они до сих пор широко используют пневматическую автоматику.

Про гидравлические машины можно сказать все то же, что и про пневматику. Наиболее известными из них являются интеграторы ИГ, созданные в СССР инженером Лукьяновым для решения сугубо практической задачи анализа изменения температуры в бетонной кладке в зависимости от состава бетона, технологии его заливки и внешних условий.

Все это описывалось дифференциальными уравнениями, и Лукьянов предложил новый способ механизации трудоёмких расчётов. Обнаружив сходство между движением потока жидкости и распространением тепла в твёрдых телах, Лукьянов разработал устройство, в котором вода выступала в роли аналога термодинамического процесса. Прибор Лукьянова представлял собой систему сосудов с водой и трубок с изменяемым гидравлическим сопротивлением — пьезометров. Подбирая величины гидравлических сопротивлений трубок и схему соединения сосудов, Лукьянов добился возможности решения системы уравнений с частными производными. Результат решения фиксировался на графиках вручную путём замера уровня воды в пьезометрах. Свое устройство Лукьянов назвал гидроинтегратором и постоянно продолжал его совершенствовать. В пятидесятые годы состав блоков гидро-

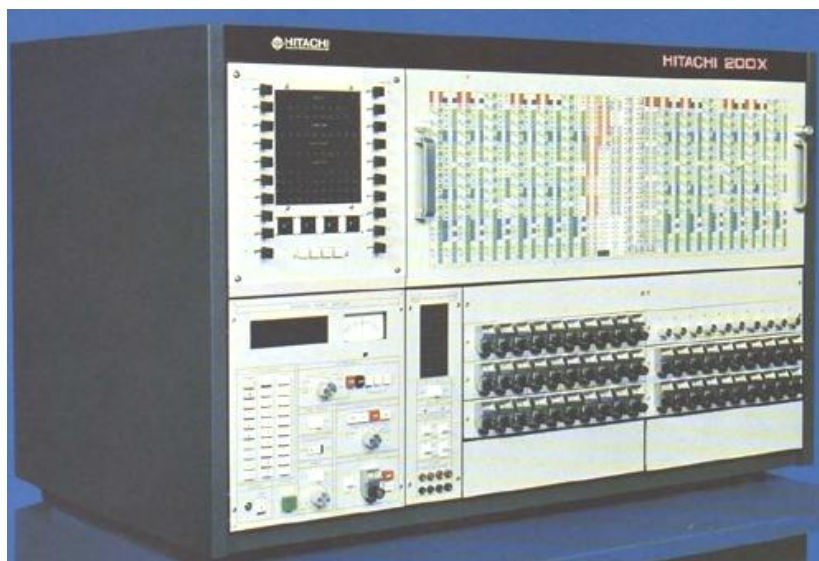


The diagram illustrates the signaling pathways that regulate the cell cycle, organized into a grid with columns 1-10 and rows A-K. Key components and interactions include:

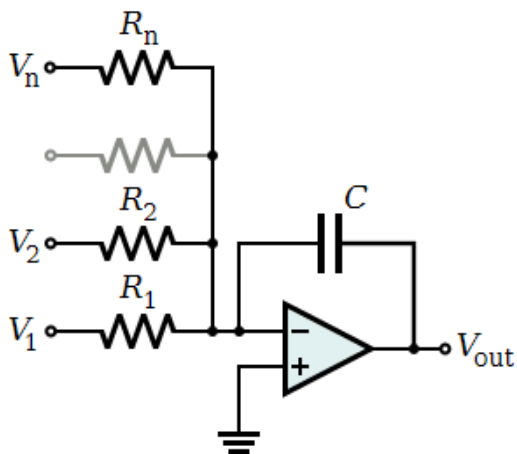
- Transcription:** SL1, Cdk7, Cdk9, and other transcription factors.
- Cyclin box:** p16, cdc25A, cdk4/6, cyclin D, p21, p27, p57, p107, p130, E2F1-6, DP1/2, Sp1, and other cyclin-related proteins.
- Replication:** PCNA, RPA, p53, Mdm2, APC, Wee1, Myt1, and other replication-related proteins.
- E2F box:** E2F1-6, DP1/2, Sp1, and other E2F-related proteins.
- Myc box:** Myc, Max, AP2, and other Myc-related proteins.
- Chromatin & acetylase box:** P/CAF, p300, HDAC1, and other chromatin-related proteins.

The diagram shows the integration of growth factor signals (Ras, Raf1, c-Abl) into the cell cycle regulation network, involving various kinases and phosphatases, and the resulting effects on transcription, replication, and cell cycle progression.

АВМ электрическая. Тут открывается широчайший простор для экспериментов. Правильно подобранными электрическими блоками можно смоделировать почти любой процесс. Такие машины были самые быстрые, удобные и универсальные. Они имели малые размеры (относительно) и применялись практически везде.



111



Интегратор

помогло распределение вычислительной задачи между аналоговым и цифровым компьютером. Машинные комплексы, представляющие собой связанные с помощью АЦП-ЦАП преобразователей аналоговые и цифровые вычислители, назывались гибридными вычислительными системами.

Наибольшую известность в шестидесятые годы прошлого столетия приобрели гибридные ЭВМ производства компании Packard-Bell. Их компьютерная система HYCOMP, состоящая из аналогового компьютера MARK III и цифровой ЭВМ PB-440, использовалась для решения расчётных задач всех миссий лунной программы «Аполлон». В СССР подобные гибридные вычислительные комплексы «Сатурн» разрабатывались Пензенским заводом САМ на базе электроинтеграторов на резистивной сетке и цифровых ЭВМ семейства «Урал».

Электромеханические АВМ

Примером комбинированной АВМ могут служить электромеханические АВМ, в которых машинными переменными являются механические (обычно угол поворота) и электрические (напряжение) величины. Специфическими для данного типа АВМ являются вращающиеся трансформаторы и тахогенераторы. АВМ данного типа менее надёжны, чем механические, из-за наличия скользящих контактов.

Гибридные АЭВМ

Несмотря на весьма небольшие погрешности отдельных операционных блоков, общая погрешность аналоговых компьютеров оставалась значительной. В сложных схемах решения ряда задач из-за накопления относительных погрешностей в их элементах суммарная погрешность схемы достигала пяти процентов. Решить эту проблему

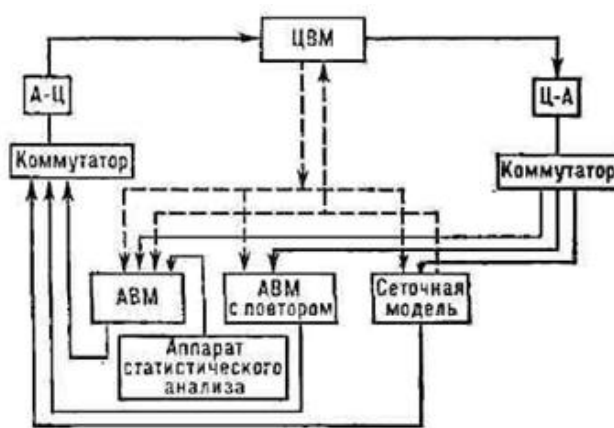


Схема АЭВМ «Сатурн»



Packard-Bell HYCOMP

В дальнейшем успехи в области разработки интегральных цифровых схем позволили реализовать принципиально новый вид гибридности. В так называемых гибридных ЭВМ операционные блоки создавались на базе аналоговых схем лишь частично. Часть из них была реализована на цифровых схемах. Такое схемотехническое решение позволило наряду с аналоговыми вычислениями реализовать: аналого-цифровое моделирование, конечно-разностное цифровое моделирование и цифровой вычислительный процесс на основе неалгоритмического потокового программирования, в ходе которого решение задачи организуется путём структурной перестройки процессора специального типа, именуемого FPAA (Field-programmable Analog Array). В FPAA в корпусе обычной интегральной микросхемы реализованы микроминиатюрные операционные блоки на основе традиционных для аналогового компьютера операционных усилителей.

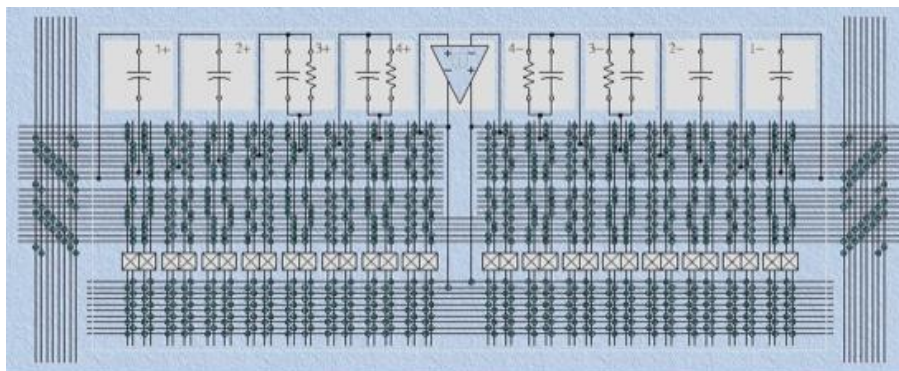


Схема программируемой аналоговой матрицы FPAА

Являясь особым видом ПЛИС, интегральные схемы FPAА легко перепрограммируются под решение конкретных вычислительных задач, обеспечивая при этом минимально возможные для аналоговых операционных элементов погрешности вычислений. В отличие от своих цифровых собратьев FPGA, содержащих значительное количество логических элементов и соединительных связей, интегральные схемы FPAА состоят из относительно небольшого числа САВ-модулей, каждый из которых содержит либо схемы на основе операционных усилителей, либо массивы ёмкостей и резисторов. Фактически микросхемы FPAА реализуют конструкцию обычного аналогового компьютера с кросс-панелью в миниатюре, что позволяет применять их для задач, в которых аналоговые компьютеры традиционно сильны. Так, современные FPAА в специальном исполнении, защищающем их от космической радиации, работают в качестве вычислителей орбит и траекторий полёта современных спутников и пилотируемых космических аппаратов.

По конструктивным признакам АВМ делятся на матричные и структурные.

АВМ матричного типа (групповая аналоговая машина) – аналоговая машина, в которой отдельные простейшие вычислительные блоки жестко соединяются в одинаковые типовые группы. В основном используется для моделирования дифференциальных уравнений. Задачу при этом предварительно необходимо свести к равносильной ей системе дифференциальных уравнений первого порядка. Каждая типовая группа вычислительных элементов используется для моделирования одного уравнения. АВМ матричного типа нуждается в определенном процессе масштабирования, при котором значения коэффициентов одного столбца матрицы должны иметь одинаковый порядок. Набор задач на таких АВМ сводится к установке коэффициентов и начальных условий. Недостатком АВМ этого типа является низкая эффективность использования отдельных блоков. К этому типу АВМ в основном относятся механические АВМ.

Структурная операционная аналоговая машина, в которой простейшие вычислительные блоки соединяются между собой в соответствии с математическими операциями решаемого уравнения, используется для математического моделирования.

АВМ с периодизацией или с повторением решения – машина, в которой этапы решения задач автоматически повторяются с помощью системы коммутации. Предел частоты повторений определяется частотными характеристиками решающих элементов. Вычислительные элементы АВМ однократного действия (операционные усилители, функциональные преобразователи и т. п.) пригодны для использования в АВМ с периодизацией. В таких АВМ используются интеграторы с малой постоянной времени. Устройство быстродействующих АВМ более сложное, чем у АВМ однократного действия, так как используются специальные схемы для разряда конденсаторов в конце цикла и схемы для автоматического ввода начальных значений в начале каждого вычислительного цикла. Самое большее преимущество АВМ такого типа — возможность наблюдать изменение результата в зависимости от параметров в реальном времени. Быстродействующие АВМ используются для приблизительного определения передаточной функции физической системы по семейству её переходных характеристик, для решения краевых задач, вычисления интеграла Фурье и корреляционного анализа.

В противоположность им существуют т.н. медленные машины – однократного действия, в которых используются интеграторы с относительно большими постоянными времени. Решение типовых задач на таких АВМ длится от нескольких секунд до нескольких минут. При этом результат изменения параметров может быть зафиксирован только после завершения всех вычислительных циклов.

В итеративных машинах процесс решения осуществляется итерационным способом в течение определенного числа итераций. Специфика такой АВМ позволяет управлять ходом вычислений в заданные моменты времени, например, возможно обрабатывать значения с выходов интеграторов и пересылать информацию из одного такта в другой в зависимости от условий.



Польский AKAT-1 и US Navy Mk III Torpedo Data Computer, аналоговый компьютер для управления торпедным огнём. Использовался на американских субмаринах во время ВМВ.



ABM Newmark, 1960 г., Кембридж и электрическая ABM MOHAI, построенная Boeing около 1953 г.

Достоинства и недостатки АВМ полностью определяются принципом их работы. Как уже говорилось, любая АВМ основана на реализации некоторого физического процесса, который соответствует исходной математической задаче. ЭВМ используют чисто математические методы решения путем выполнения последовательности арифметических и логических операций над данными, представленными в дискретном виде. Упрощенно говоря, АВМ реализует аппаратную модель исследуемой задачи с помощью физической коммутации некоторой схемы, а ЭВМ — математическую, на базе программного кода.

Соответственно с ростом сложности математической задачи объем аппаратуры АВМ растет, по крайней мере, линейно, а ЭВМ — остается неизменным. Но скорость получения решения на АВМ не зависит от сложности задачи (условно говоря, решение получается мгновенно), а на ЭВМ — весьма существенно. Однако при этом следует иметь в виду, что реальный цикл решения задачи определяется не только самими вычислениями: есть еще затраты на ввод программы (АВМ — коммутация схемы, ЭВМ — чтение файла с диска), подготовку и ввод данных (установка напряжений/ввод файла) и пр. Здесь АВМ явно отстает.

Результатом является то, что аналоговые вычислители уже давно потеряли актуальность в качестве универсальных машин. Это произошло естественным образом еще в 60-х годах по мере роста производительности ЭВМ и снижения их стоимости. Реально АВМ сейчас используются только при решении относительно простых задач, для которых жесткое аппаратное решение является вполне допустимым. Примером такого рода задач является и простейший спидометр и схема управления зениткой по данным радиолокационных наблюдений — один раз проектируется, зашивается в схему и работает, пока не выбросят. При этом вытеснение аналоговых устройств цифровыми является совершенно отчетливой тенденцией.

Точность вычислений. АВМ изначально имеют ограниченную точность, так как она определяется точностью как параметров физических компонент модели (например, сопротивлений и емкостей), так и приборов, измерительных и задающих исходные величины. Поэтому в АВМ используются прецизионные компоненты (до 0,001% погрешности), но погрешность конечных результатов все равно крайне редко опускается ниже 1-10%. Впрочем, этого бывает вполне достаточно для решения многих практических задач.

Отличительная черта методов вычислительной математики и ЭВМ заключается в том, что погрешность решения можно приблизить сколь угодно близко к нулю (разумеется, за счет увеличения объема и времени вычислений), причем это не зависит от архитектуры конкретной ЭВМ, в том числе и ее разрядности (разрядностью данных можно управлять программным образом), питания и прочего. В ЭВМ высокая точность вычислений достигается на аппаратуре с минимальными требованиями к качеству электронных компонент (имеется в виду допустимый разброс их параметров). Например, в цифровых схемах используются компоненты с погрешностью в 10-30%, что никак не влияет на правильность работы всего устройства. Это происходит потому, что них используется механизм бинарных переключателей с достаточно большим допуском сигналов. (Например, для 5-вольтовой ТТЛ-схем уровень сигналов лежит в диапазоне: логический 0 — 3,5-5 В, 1 — 0-1,5 В). Именно это лежит в основе создания цифровых интегральных микросхем, в которых каждый транзистор имеет значительный разброс параметров, а проводить отбраковку годных/негодных уже нет возможности.

Вот хороший рассказ⁴⁶ о том, почему большие аналоговые машины пришли в упадок: «Первое знакомство с АВМ на уровне теории и курса лабораторных работ я получил в МИФИ в середине 70-х годов, а на практике мне пришлось столкнуться в 80-м, когда перешел на новую работу и попал в лабораторию моделирования геофильтрационных процессов. Эти модели описываются достаточно сложными дифференциальными уравнениями, и для их решения в лаборатории использовался некий электрический интегратор (не помню его названия). Это был здоровый шкаф размером $4 \times 2,5 \times 1,5$ м, в котором находилось коммутационное поле и куча каких-то реле, источников питания, проводов и пр. В отдельных шкафах лежали груды сопротивлений и емкостей разных номиналов. Сотрудники лаборатории еще раньше начали также использовать расчеты на ЭВМ (арендовалось машинное время). Для судьбы же ЭИ 80-й год стал решающим — в институте вводилась в эксплуатацию первая собственная ЕС-1022, а в лаборатории единственная в институте СМ-1.

Тем не менее, в конце 1980 г. мне удалось стать свидетелем использования ЭИ. Почти неделю два-три сотрудника выполняли расчет электрических параметров модели. Потом в течение двух недель они проводили коммутацию и настройку интегратора. Само вычисление произошло мгновенно - в момент включения рубильника, но съемка и обработка полученных результатов заняла еще пару дней. Затем считался новый вариант — корректировка параметров, запись результатов (еще два-три дня) и т.д.

Одновременно я в тестовом режиме решал ту же задачу на СМ-1 (32 Кб оперативной памяти), для которой уже написал соответствующую программу. Решение одного варианта занимало 3-40 мин. (модель была очень чувствительна к исходным данным). Коррекция исходных данных для одного варианта и распечатка результатов требовали еще 10-15 минут. Сравнение результатов расчетов на ЭИ и СМ-1 выявили несколько ошибок в программе, но еще больше — ошибок при коммутации и замерах данных на ЭИ. Через пару недель таких параллельных расчетов, начался приказал выключить ЭИ и продолжать расчеты только на СМ-1. Уже через год работы на ЕС и СМ ЭВМ нам было даже как-то неловко вспоминать примитивность математической модели, которая была пределом для того ЭИ. Но списали и выкинули его только через 5 лет, когда переезжали в другое помещение — все это время на «протирку контактов» ЭИ ежемесячно выписывалось по пять литров спирта».

⁴⁶ Андрей Колесов, «Компьютерра» № 26, 1997

2.2. Физика ЭВМ - настоящее и будущее

2.2.1. Явление полупроводимости

Полупроводником, как известно, называют материал, который по своей удельной проводимости занимает промежуточное место между проводниками и диэлектриками. От проводников он отличается сильной зависимостью удельной проводимости от концентрации примесей, температуры и воздействия различных видов излучения, при определенных воздействиях превращаясь в диэлектрик. Классическим свойством полупроводника является увеличение электрической проводимости с ростом температуры.

Основным понятием из квантовой механики, описывающим полупроводники, является т.н. запрещенная зона – область значений энергии, которыми не может обладать электрон в идеальном (бездефектном) кристалле. В полупроводниках запрещенной зоной называют область энергий, отделяющую полностью заполненную электронами валентную зону (при $T=0$ К) от незаполненной зоны проводимости. В этом случае шириной запрещенной зоны называется разность энергий между дном (нижним уровнем) зоны проводимости и потолком (верхним уровнем) валентной зоны.

В результате вещества с очень широкой зоной не имеют электронов в зоне проводимости и не могут их там иметь – при любых условиях их энергии будет недостаточно для пересечения запрещенной зоны – кристалл скорее разрушится от воздействия. В проводниках все наоборот – зона проводимости заполнена электронами, с которыми легко взаимодействует электрическое поле, создавая ток.

Ширина запрещенной зоны в полупроводниках составляет 0,1–4 эВ. Кристаллы с шириной запрещенной зоны более 4 эВ обычно относят к диэлектрикам. Полупроводники же отличаются тем, что их зона невелика, но электронов обычно в зоне проводимости изначально нет, хотя при особых условиях – примеси, температура и т.п., они там появляются, и диэлектрик становится проводником. Естественно, возможен и обратный переход.

Полупроводники с шириной запрещенной зоны менее ~0,3 эВ называют узкозонными полупроводниками, а полупроводники с шириной запрещенной зоны более ~3 эВ – широкозонными полупроводниками. Полупроводники, в которых переход электрона из зоны проводимости в валентную зону не сопровождается потерей импульса (прямой переход), называются прямыми. Полупроводники, в которых переход электрона из зоны проводимости в валентную зону сопровождается потерей импульса (непрямой переход), называются непрямозонными. При этом в процессе поглощения энергии, кроме электрона и фотона, должна участвовать ещё и третья частица (например, фонон), которая заберёт часть импульса себе (на этом принципе работают, например, светодиоды или полупроводниковые лазеры).

Наличие прямых и непрямых переходов объясняется зависимостью энергии электрона от его импульса. При излучении или поглощении фотона при таких переходах общий импульс системы электрон-фотон сохраняется согласно закону сохранения импульса.

К числу полупроводников относятся многие химические элементы (германий, кремний, селен, теллур, мышьяк и др.), огромное количество сплавов и химических соединений (арсенид галлия и др.). Почти все неорганические вещества окружающего нас мира – полупроводники. Самым распространённым в природе полупроводником является кремний, составляющий почти 30 % земной коры.

В зависимости от того, отдаёт примесной атом электрон или захватывает его, примесные атомы называют донорными или акцепторными. От того, какой атом кристаллической решётки замещается, или в какую кристаллографическую плоскость встраивается, характер примеси может меняться. Проводимость полупроводников сильно зависит от температуры. Вблизи температуры абсолютного нуля полупроводники имеют свойства диэлектриков. При повышении температуры электроны получают достаточно энергии, чтобы оторваться от ядра: так образуется свободный отрицательный заряд – собственно электрон, и частица с избыточным положительным зарядом – атом без электрона. Во время разрыва связи между электроном и ядром появляется свободное место в электронной оболочке атома, куда переходит электрон с другого атома. На атом, откуда перешёл электрон, входит электрон из другого атома и т. д. Этот процесс обуславливается ковалентными связями атомов. Таким образом, происходит перемещение положительного заряда без перемещения самого атома. Этот условный положительный заряд называют дыркой.

Электронно-дырочный механизм проводимости проявляется у собственных (то есть без примесей) полупроводников. Он называется собственной электрической проводимостью полупроводников. Для создания полупроводниковых приборов часто используют кристаллы с примесной проводимостью. Такие кристаллы изготавливают внесением примесей с атомами трехвалентного или пятивалентного химического элемента.

Полупроводники n-типа

Термин «n-тип» происходит от слова «negative», обозначающего отрицательный заряд основных носителей. Этот вид полупроводников имеет примесную природу. В четырёхвалентный полупроводник (например, кремний) добавляют примесь пятивалентного полупроводника (например, мышьяка, получают классический кристалл арсенида кремния для изготовления диодов и транзисторов).

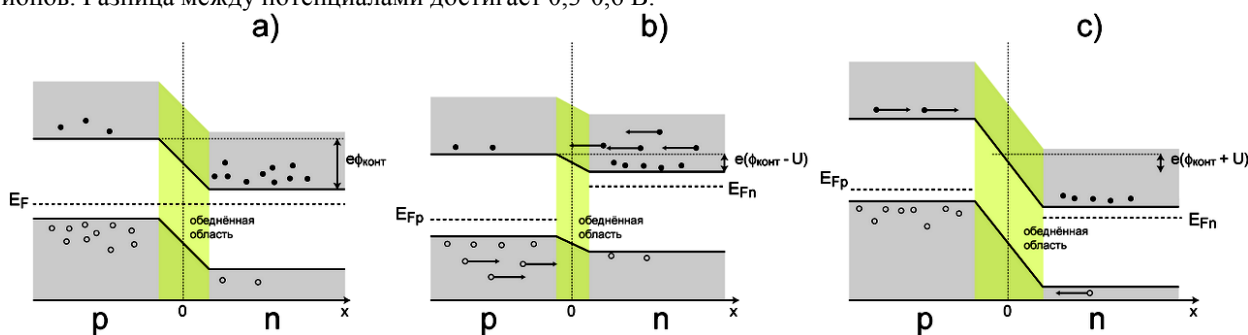
В процессе взаимодействия каждый атом примеси вступает в ковалентную связь с атомами кремния. Однако для пятого электрона атома мышьяка в насыщенных валентных связях нет места, и он переходит на

дальнюю электронную оболочку. Там для отрыва электрона от атома нужно меньшее количество энергии. Электрон отрывается и превращается в свободный. В данном случае перенос заряда осуществляется электроном, а не дыркой, то есть данный вид полупроводников проводит электрический ток подобно металлам. Примеси, которые добавляют в полупроводники, вследствие чего они превращаются в полупроводники n-типа, называются **донорными**.

Полупроводник p-типа

Термин «р-тип» происходит от слова «positive», обозначающего положительный заряд основных носителей. Этот вид полупроводников, кроме примесной основы, характеризуется дырочной природой проводимости. В четырёхвалентный полупроводник (например, в кремний) добавляют небольшое количество атомов трехвалентного элемента (например, индия). Каждый атом примеси устанавливает ковалентную связь с тремя соседними атомами кремния. Для установления связи с четвёртым атомом кремния у атома индия нет валентного электрона, поэтому он захватывает валентный электрон из ковалентной связи между соседними атомами кремния и становится отрицательно заряженным ионом, вследствие чего образуется дырка. Примеси, которые добавляют в этом случае, называются акцепторными.

Полупроводниковый диод состоит из двух типов полупроводников — дырочного и электронного. В процессе контакта между этими областями из области с полупроводником n-типа в область с полупроводником p-типа проходят электроны, которые затем рекомбинируют с дырками. Возникает электрическое поле между двумя областями, что устанавливает предел деления полупроводников — так называемый p-n переход. В результате в области с полупроводником p-типа возникает некомпенсированный заряд из отрицательных ионов, а в области с полупроводником n-типа возникает некомпенсированный заряд из положительных ионов. Разница между потенциалами достигает 0,3-0,6 В.

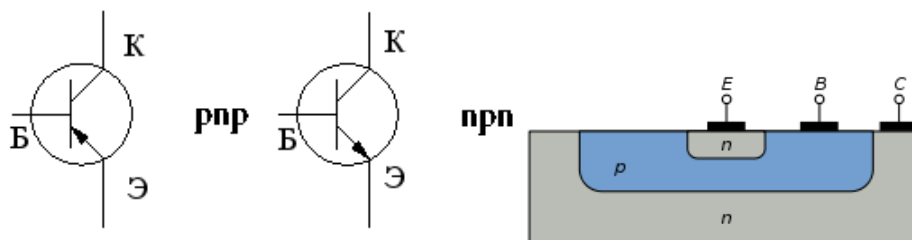


В процессе подачи напряжения плюсом на p-полупроводник и минусом на n-полупроводник внешнее электрическое поле будет направлено против внутреннего электрического поля p-n перехода и при достаточном напряжении электроны преодолеют p-n переход, и в цепи диода появится электрический ток (прямая проводимость). При подаче напряжения минусом на область с полупроводником p-типа и плюсом на область с полупроводником n-типа между двумя областями возникает область, которая не имеет свободных носителей электрического тока (обратная проводимость). Обратный ток полупроводникового диода не равен нулю, так как в обеих областях всегда есть неосновные носители заряда. Для этих носителей p-n переход будет открыт. Таким образом, p-n переход проявляет свойства односторонней проводимости, что обуславливается подачей напряжения с различной полярностью. Это свойство используют для выпрямления переменного тока.

2.2.2. Транзистор

Транзистор — полупроводниковое устройство, которое состоит из двух областей с полупроводниками p- или n-типа, между которыми находится область с полупроводником n- или p-типа. Таким образом, в транзисторе есть две области p-n перехода. Область кристалла между двумя переходами называют базой, а внешние области называют эмиттером и коллектором. Самой употребляемой схемой включения транзистора является схема включения с общим эмиттером, при которой через базу и эмиттер ток распространяется на коллектор.

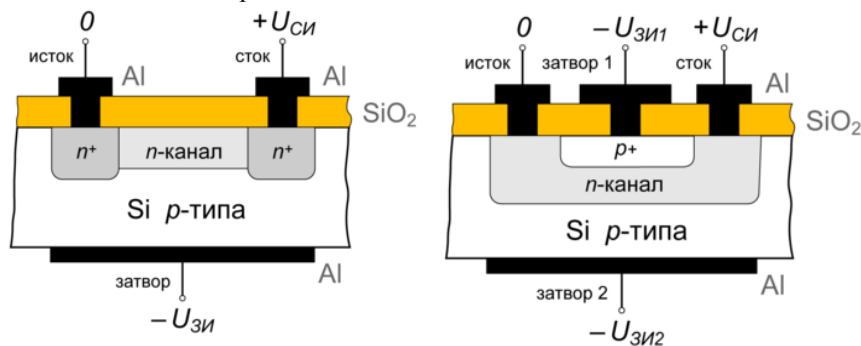
Биполярный транзистор используют для усиления электрического тока.



Биполярный транзистор состоит из трех различным образом легированных полупроводниковых зон: эмиттера E , базы B и коллектора C . В зависимости от типа проводимости этих зон различают NPN (E_n, B_p, C_n) и PNP транзисторы. К каждой из зон подведены проводящие контакты. База расположена между эмиттером и коллектором и изготовлена из слаболегированного полупроводника, обладающего большим сопротивлением.

В активном режиме работы транзистор включён так, что его эмиттерный переход смещён в прямом направлении (открыт), а коллекторный переход смещён в обратном направлении (закрыт). В **npn** транзисторе электроны, основные носители тока в эмиттере, проходят через открытый переход эмиттер-база (инжектируются) в область базы. Часть этих электронов рекомбинирует с основными носителями заряда в базе (дырками). Однако из-за того, что базу делают очень тонкой и сравнительно слабо легированной, большая часть электронов, инжектированных из эмиттера, диффундирует в область коллектора. Сильное электрическое поле обратно смещённого коллекторного перехода захватывает электроны и проносит их в коллектор. Ток коллектора, таким образом, практически равен току эмиттера, за исключением небольшой потери на рекомбинацию в базе, которая и образует ток базы $I_{\beta} = I_E + I_K$. Коэффициент α , связывающий ток эмиттера и ток коллектора ($I_K = \alpha I_{\beta}$), называется коэффициентом передачи тока эмиттера. Численное значение коэффициента $\alpha = 0.9 : 0.999$. Чем больше коэффициент, тем эффективней транзистор передаёт ток. Этот коэффициент мало зависит от напряжения коллектор-база и база-эмиттер. Поэтому в широком диапазоне рабочих напряжений ток коллектора пропорционален току базы, коэффициент пропорциональности равен $\beta = \alpha / (1 - \alpha) = 10 : 1000$. Таким образом, изменяя малый ток базы, можно управлять большим током коллектора.

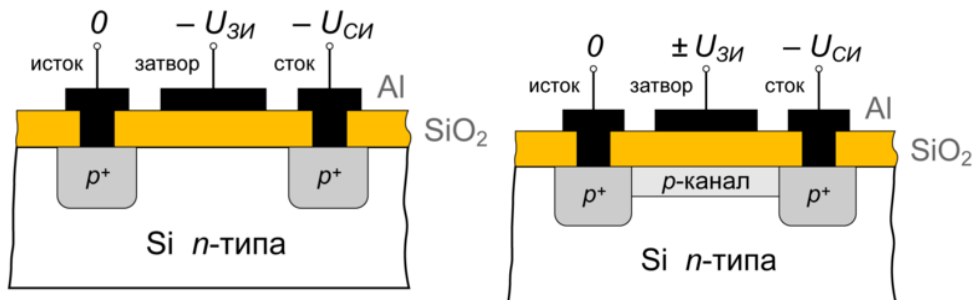
Полевой транзистор — полупроводниковый прибор, в котором ток изменяется в результате действия перпендикулярного току электрического поля, создаваемого входным сигналом. Протекание в полевом транзисторе тока обусловлено носителями заряда только одного знака.



Устройство полевого транзистора с управляющим p-n переходом

Полевой транзистор с управляющим p-n переходом — это транзистор, затвор которого изолирован от канала p-n переходом, смещённым в обратном направлении.

Такой транзистор имеет два невыпрямляющих контакта к области, по которой проходит управляемый ток основных носителей заряда, и один или два управляющих электронно-дырочных перехода, смещённых в обратном направлении. При изменении обратного напряжения на p-n переходе изменяется его толщина и, следовательно, толщина области, по которой проходит управляемый ток основных носителей заряда. Область, толщина и поперечное сечение которой управляется внешним напряжением на управляющем p-n переходе и по которой проходит управляемый ток основных носителей, называют каналом. Электрод, из которого в канал входят основные носители заряда, называют **истоком**. Электрод, через который из канала уходят основные носители заряда, называют **стоком**. Электрод, служащий для регулирования поперечного сечения канала, называют **затвором**.



Устройство полевого транзистора с изолированным затвором (МДП, МОП-транзисторы)

Электропроводность канала может быть как n-, так и p-типа; по ней различают полевые транзисторы с n-каналом и p-каналом. Все полярности напряжений смещения, подаваемых на электроды транзисторов с n- и с p-каналом, противоположны.

Управление током стока, то есть током от внешнего относительно мощного источника питания в цепи нагрузки, происходит при изменении обратного напряжения на p-n переходе затвора (или на двух p-n переходах одновременно). В связи с малостью обратных токов мощность, необходимая для управления током стока и потребляемая от источника сигнала в цепи затвора, оказывается ничтожно малой. Поэтому полевой транзистор может обеспечить усиление электромагнитных колебаний как по мощности, так и по току и напряжению.

Таким образом, полевой транзистор по принципу действия аналогичен вакуумному триоду. Исток в полевом транзисторе подобен катоду вакуумного триода, затвор – сетке, сток – аноду, но при этом полевой транзистор существенно отличается от вакуумного триода. Во-первых, для работы полевого транзистора не требуется подогрева катода. Во-вторых, любую из функций истока и стока может выполнять каждый из этих электродов. В-третьих, полевые транзисторы могут быть сделаны как с n-каналом, так и с p-каналом, что позволяет удачно сочетать в схемах эти два типа полевых транзисторов.

Полевой транзистор с изолированным затвором – это транзистор, затвор которого отделён в электрическом отношении от канала слоем диэлектрика.

В кристалле полупроводника с относительно высоким удельным сопротивлением, который называют подложкой, созданы две сильно легированные области с противоположным относительно подложки типом проводимости. На эти области нанесены металлические электроды – исток и сток. Расстояние между сильно легированными областями истока и стока может быть меньше микрона. Поверхность кристалла полупроводника между истоком и стоком покрыта тонким слоем (порядка 0,1 мкм) диэлектрика. Так как исходным полупроводником для полевых транзисторов обычно является кремний, то в качестве диэлектрика используется слой двуоксида кремния SiO_2 , выращенный на поверхности кристалла кремния путём высокотемпературного окисления. На слой диэлектрика нанесён металлический электрод – затвор. Получается структура, состоящая из металла, диэлектрика и полупроводника. Поэтому полевые транзисторы с изолированным затвором часто называют МДП-транзисторами.



МОП-транзисторы занимают на кристалле микросхемы в 6-9 раз меньшую площадь, чем транзисторы, используемые в ТТЛ, за счёт упрощения топологии, так, наиболее простой тип транзистора с индуцированным каналом требует всего одной операции легирования и одной – металлизации. В англоязычной литературе такой тип называется «обогащённый канал». Это позволило добиться высокой степени интеграции и создать микропроцессоры (процессоры, собранные в одной микросхеме). Однако схемы на транзисторах с индуцированным каналом требуют весьма высокого напряжения питания (27 вольт для типичных p-МОП структур, и 12 вольт для типичных n-МОП) и обладают невысоким быстродействием, задержка переключения для p-МОП схем составляла десятки, в лучшем случае – единицы микросекунд, а для n-МОП – сотни наносекунд. Увеличить быстродействие с одновременным снижением питающего напряжения удалось применением транзисторов со встроенным каналом, работающим в режиме обеднения. Такие транзисторы требуют на одну операцию легирования больше, но позволили n-МОП схемам работать от одного источника напряжением 5 вольт.

Дальнейшее повышение быстродействия было связано с отказом от металлических затворов и переходом на затворы из поликристаллического кремния. Чтобы ещё увеличить быстродействие, для изоляции затвора от канала были использованы диэлектрики с меньшим коэффициентом диэлектрической проницаемости, чем у оксида кремния, поэтому полевые транзисторы современных цифровых СБИС уже неправомерно называть МОП-транзисторами. С целью радикального снижения энергопотребления в одном логическом элементе могут применяться транзисторы обоих типов проводимости, как с каналом n-типа, так и с каналом p-типа. Такая схема называется КМОП – «комплементарная». В отличие от схем на одном типе проводимости, КМОП схемы практически не потребляют тока в статическом режиме, так как в цепочке последовательно соединённых транзисторов хотя бы один всегда закрыт, и лишь во время переходного процесса все транзисторы оказываются ненадолго открытыми. КМОП-структуры требуют большего числа технологических операций при изготовлении, что первоначально ограничивало достижимую степень интеграции (довольно долго по этому параметру лидировали n-МОП структуры).

2.2.3. ЭВМ будущего – от мемристоров до оптических машин

Компьютеры прошли огромный путь, от прототипа Беббиджа, до Core i7. Многие аспекты устройства компьютеров сильно эволюционировали, многие, напротив, ничуть не изменились. Многие уникальные архитектуры не выдержали испытания временем, как уже стало ясно читателю этой книги, но общие физические принципы остаются неизменными. Уже не менее 30 лет утверждают, что технологический предел будет вот-вот достигнут, и развитие традиционных компьютеров остановится, но пока этот миг так и не наступил.

Тем не менее, конец классической архитектуры все равно неизбежен – скорость света конечна, как и минимальный размер элементов, так что рано или поздно наши машины будут выглядеть и работать абсолютно не так, как сейчас. Давайте кратко познакомимся с наиболее перспективными направлениями инженерной мысли ближайшего будущего.

Как мы уже видели, существует несколько существенно различных моделей вычислений, так что ЭВМ будущего, возможно, будут устроены совершенно непредвиденным образом. Однако компьютер как управляющее устройство в любом случае должен содержать исполнительный блок (центральный процессор), и устройство этого блока жёстко зафиксировано его функциями: он должен уметь выдавать последовательные инструкции другим устройствам и совершать выбор дальнейшей программы действий в зависимости от сложившихся условий.

Эти два требования автоматически предполагают модель устройства, называемую «relaxed sequential execution» по-английски или «алгоритмической моделью» в русскоязычной традиции. Эта модель, в отличие от многих других, не имеет квантового аналога, так как квантовые вычисления принципиально обратимы, а совершение выбора принципиально необратимо.

Для полноты приведём список всех известных на данный момент масштабируемых аппаратных моделей вычислений:

- алгоритмическая модель, то есть, условно-пошаговое выполнение инструкций («условно» означает, что допускается параллельное выполнение независимых инструкций);
- потоковые векторные вычисления и их естественное обобщение — квантовые конвейеры;
- клеточные/сетевые автоматы и их естественное обобщение — квантовые автоматы.

На данный момент на практике за исключением модели последовательного выполнения используются лишь потоковые векторные вычисления: на бытовых компьютерах в специализированных сопроцессорах для трёхмерной графики (GPU), а также в их аналогах повышенной мощности (GPGPU) на вычислительных станциях для численных симуляций и обработки больших массивов данных. Центральный процессор делегирует некоторые специфические задачи потоковому процессору, однако управляющие функции в полной мере выполняет только он сам. С развитием квантовых вычислений это положение вряд ли изменится: квантовые конвейеры и автоматы будут лишь дополнениями центрального процессора, но не его заменой.

На элементарном уровне процессоры состоят из двух типов элементов: переключателей и соединяющих их дорожек. Задача дорожек в том, чтобы переносить сигнал, а задача переключателей в том, чтобы этот сигнал трансформировать. Существует много разных реализаций переключателей и дорожек. В окружающей нас цифровой электронике в качестве переключателей используются транзисторы, а дорожки делаются из проводников.

Передача сигнала

В двоичной цифровой электронике для кодирования сигнала используется напряжение. Дорожка несёт сигнал «ноль», если на неё подан потенциал $U_0 \pm \varepsilon$, а если на неё подан потенциал $U_1 \pm \varepsilon$, то она несёт сигнал «единица». Базовые уровни U_0 и U_1 выбираются так, чтобы $U_0 + \varepsilon < U_1 - \varepsilon$, где ε — неизбежная по техническим причинам погрешность. Переход из одного состояния в другое — это остановка или, наоборот, разгон огромного количества электронов в проводнике. Оба эти процесса занимают довольно много времени и неизбежно связаны с большими энергетическими потерями.

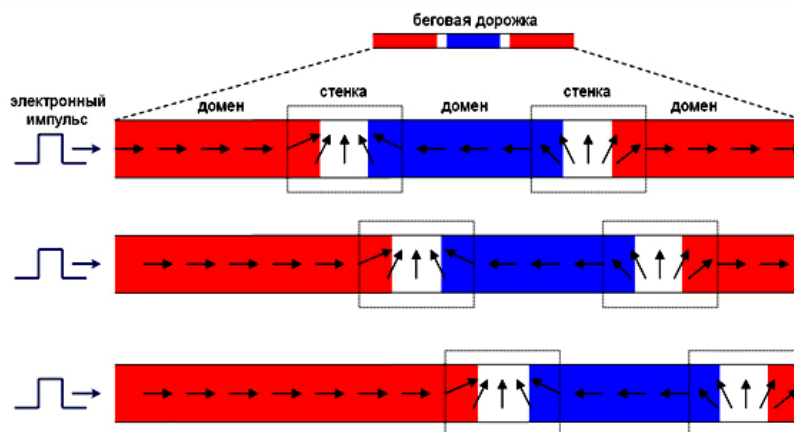
Существуют альтернативы цифровой электронике, лишённые этого недостатка: по дорожкам можно пускать постоянный поток электронов (т.н. спинтроника) или фотонов (т.н. фотонная логика или оптические компьютеры), а сигнал кодировать при помощи поляризации этого потока. Как электроны, так и фотоны, движущиеся по одномерному волноводу, могут пребывать ровно в двух ортогональных состояниях поляризации, одно из которых объявляется нулём, а другое единицей.

Спинтроника является разделом квантовой электроники, занимающийся изучением спинового токопереноса (спин-поляризованного транспорта) в твердотельных веществах, в частности в гетероструктурах ферромагнетик-парамагнетик или ферромагнетик-сверхпроводник.

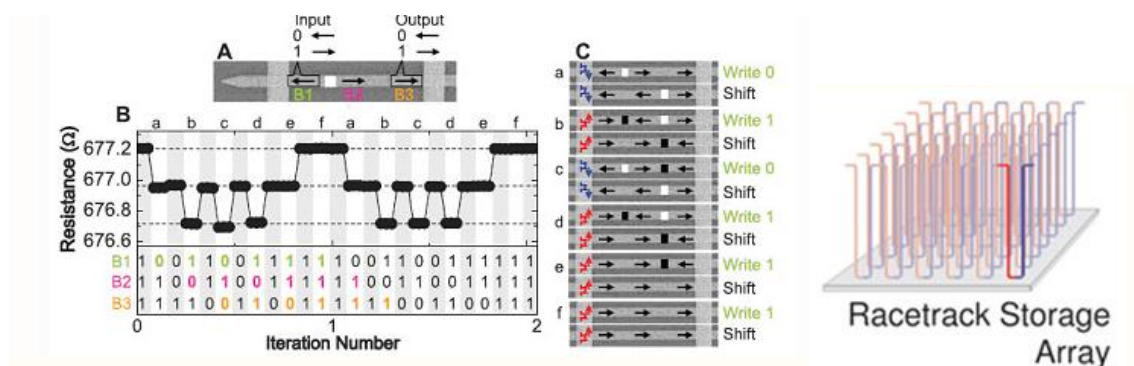
В таких гетероструктурах источником спин-поляризованных электронов (спин-инжектором) является проводящий ферромагнетик (проводник или полупроводник), обладающий в намагниченном состоянии спонтанной, спиновой упорядоченностью носителей заряда.

В ферромагнитных полупроводниках достигаются уровни спиновой поляризации значительно более высокие (до 100 %), чем в металлах (до 10 %). Во внешнем магнитном поле возможно зеемановское расщепление зоны проводимости в полупроводнике с формированием двух зеемановских энергетических подуровней. При инжекции спин-поляризованных электронов в такой полупроводник возможны управляемые переходы как на верхний, так и на нижний уровень, что даёт, в частности, возможность создания инверсии населённости и, соответственно, генерации когерентного электромагнитного излучения с управлением частоты магнитным полем. Другие эффекты возникают в джозефсоновских переходах с изолирующим ферромагнетиком: в этом случае возможно управление туннелированием с помощью внешнего магнитного поля. Теоретически спинтроника позволяет создать следующие ценные для нас компоненты:

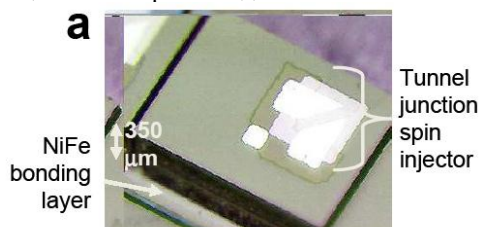
Память STT-RAM (Spin Torque Transfer RAM, magnetic racetrack memory, MRM), трековая память. Она использует спиновый ток для перемещения наноразмерных магнитных объектов — доменных стенок — в пределах магнитных нанопроволок. Под действием такого тока доменные стенки бегут друг за другом по этой проволоке, словно бегуны по спринтерской дорожке (треку):



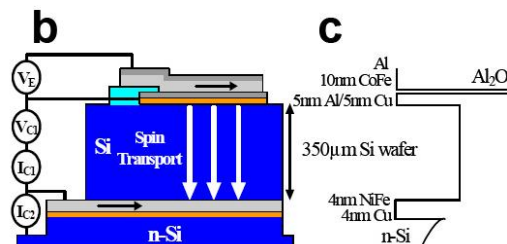
Исследовательская группа под руководством Стюарта Паркина анонсировала в 2011 г. первые результаты работ по созданию STT-RAM. С. Паркин и его коллеги научились создавать доменные стенки и гонять их по проволоке в любом направлении со скоростью свыше 100 м/с. Для проволоки длиной в несколько микрон это отвечает времени доступа в несколько десятков наносекунд — довольно быстро даже по современным меркам. Более того, авторам удалось создать и продемонстрировать в работе самый настоящий трехбитный сдвиговый регистр — возможно, на его основе через 6-7 лет будет построена полноценная память.



Далее идут спиновые транзисторы, представляющие собой слоистую структуру «ферромагнетик — кремний — ферромагнетик — кремний с примесями». После прохождения первого ферромагнитного слоя электрический ток приобретает спин-поляризацию, которая частично сохраняется при движении через слой кремния, что позволяет управлять значением спинового тока на выходе путем изменения ориентации магнитных полей двух слоев ферромагнетика. Лучшее значение по состоянию на 2007 год — сохранение спино-вой поляризации у 37 % электронов при температуре -73°C и толщине слоя кремния до 350 нм.



a — изображение кремниевого спинового транзистора, **b** — схема работы и **c** — зонная диаграмма его компонентов. При постоянном напряжении на эмиттере V_e измерялся «первый коллекторный ток» I_{c1} на NiFe-контакте и «второй коллекторный ток» I_{c2} на индиевом контакте, осажденном на кремниевой подложке n-типа



- Логические схемы, обладающие по сравнению с современными CMOS-схемами потенциально более высоким быстродействием (время задержки сигнала менее 1 нс), более низким тепловыделением (тепловыделение вентиля 10^{-17} Дж) и не подверженные воздействию ионизирующих излучений.

Фотоника, в силу большей простоты, пока опережает спинтронику. Работы по созданию оптического процессора начались еще в далеких 80-х годах. Оптический процессор должен был использовать специальные элементы, в которых свет управлял бы светом. Логические операции представлены как взаимодействие вещества со светом. В 1990 г. фирма Bell (группа Алана Хуана) создала макет оптического устройства и продемонстрировала выполнение логических и арифметических операций с очень высоким быстродействием. Процессор второго поколения носил название DOC-II (*Digital Optical Computer*) и был способен проверять до 80 тыс. страниц текста в секунду при выполнении команды поиска слова.

В основу процессора были положены двумерные матрицы бистабильных полупроводниковых элементов со множествами квантовых ям. Эти элементы обладали электрооптическими свойствами (в англоязычной литературе SEED - self-electro-optic-effect devices). Освещение элементов производилось полупроводниковым лазером через голографическую решетку Даммена. Мощность излучения лазера составила 10 мВт, длина волны – 850 нм. Свет проходил через один диод, в цепи возникал ток, что, в свою очередь, приводило к падению напряжения на структуре решетки и к повышению пропускания света через вторую структуру. Так возникала обратная связь, и совокупность элементов образовывала логические ячейки.

Первый оптический компьютер занимал один квадратный метр и состоял из четырех каскадов. На выходе каждого каскада определялось пространственное распределение излучения по состоянию входящей в состав каскада жидкокристаллической маски, которая управлялась обычным компьютером. Во втором поколении оптических компьютеров использовалась векторно-матричная логика. Поток данных в компьютере DOC-II излучали 64 модулируемых лазерных диода, длина волны каждого составляла 837 нм. Свет от каждого диода отображался на одну строчку матричного пространственного модулятора, общий размер которого составлял 64128 элементов. Отдельный элемент матрицы - это не что иное, как брэгговская оптическая ячейка (на основе GaP). Свет, который выходит из модулятора, попадает на целый ряд фотодиодов (128 штук). В секунду компьютер может сделать 0,8192 переключения, при этом на одно переключение затрачивается 7,15 фДж, или около 3000 фотонов.

Разумеется, у оптических компьютеров хватает недостатков. Основной из них – низкий уровень интеграции, что дает плату на порядок или два больше и тяжелее всех современных компонентов. В настоящее время ведутся работы по созданию схем большей интеграции – основной из них проект High Performance Optoelectronic Communication – НРОС, фирмы IBM. Его прототип был создан в 2008 г. и обеспечил пакетную передачу данных со скоростью более 1 Тбит/сек, правда, расстояние передачи и габариты, не говоря уже о сложности производства, все еще далеки от идеальных. В НРОС использовалась входная матрица с вертикально расположенными лазерными диодами. Диоды соединялись волноводами и обычной оптикой, оснащенной матрицами переключения на основе дифракционных оптических элементов. Выходная система состоит из матрицы фотодиодов, совмещенной с входной матрицей. В модуле используются технологии CMOS, Bi-CMOS, GaAs, оптические межсоединения организованы с помощью свободного распространения световых пучков. В итоге получается квазичетырехмерная структура. НРОС переключается со скоростью 10^{15} оп/сек и потребляет всего 1 фДж на переключение.

Но это все мелочи, по сравнению с Lenslet EnLight 256 - единственным существующим коммерческим оптическим процессором! Строго говоря, это не процессор общего назначения, а всего лишь DSP, хоть и превосходящий обычные сигнальные процессоры в три-четыре раза по производительности, но концепция внушает уважение. Он был создан израильской фирмой Lenslet в 2008 г., на сегодняшнее время его судьба неизвестна. Если говорить точнее, EnLight256 – это гибридный оптический процессор, содержащий преобразователи. Создать полностью оптический компьютер пока слишком дорого, но простая замена ядра с сохранением всех остальных электронных компонент позволяет получить огромный прирост производительности.

Ядро этого процессора - оптическое, а входная и выходная информация представляется в электронном виде. Ядро состоит из 256-ти VCSEL-лазеров, пространственного модулятора света, набора линз и приемников. Производительность процессора составляет 8 триллионов операций в секунду: за один такт (8 нс) процессор умножает 256-байтный вектор на матрицу 256x256.

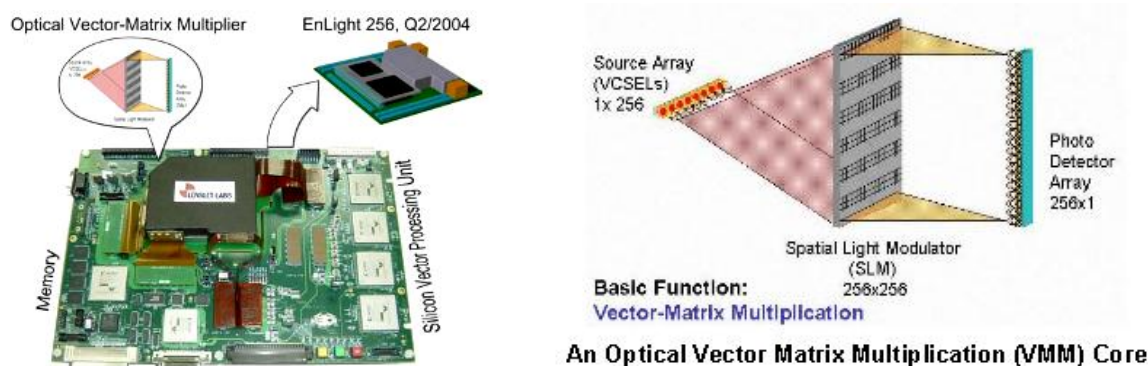
Оптическая матрица VMM (Vector-Matrix Multiplication), ядро процессора, конвертирует электрическую информацию в свет, затем производит необходимые преобразования этой информации, направляя свет через программируемую внутреннюю оптику. Свет на выходе ощущается множеством датчиков и преобразуется снова в электрический сигнал.

VMM состоит из трех основных элементов:

- N не-когерентных лазеров, которые представляются как вектор, состоящий из N элементов, каждый элемент – это 8 бит;
- Пространственного модулятора Multiple Quantum Well (MQW), состоящего из $N \times N$ пиксельных модуляторов, размещенных на одном чипе;

- Ряд N детекторов света, которые интегрированы в массив аналогово-светового преобразования (Analog to Digital Converters, ADC). Детекторы установлены так, чтобы получать лучи от матрицы модулятора. Вывод столбца детектора – это вектор-результат.

Каждый элемент входного вектора проектируется на столбец матрицы. Каждый ряд матрицы проектируется на один детектор в векторе результата (вывода). Программирование оптического цифрового сигнального процессора (Optical Digital Signal Processing Engine, ODSPE) заключается в изменении значений, которые сохранены в пространственном модуляторе (Spatial Light Modulator, SLM). Загрузка приложения (или данные внутри приложения) аналогична замене матрицы в пространственном модуляторе. Кстати, пространственный модулятор Ablaze может поставляться как отдельный продукт, так что ничто не мешает создать свой оптический процессор.



Так выглядит система в сборе и модуль VMM из рекламной брошюры производителя

Теоретически компьютер на базе EnLight256 способен обрабатывать 15 видеоканалов стандарта HDTV в режиме реального времени и позволяет создать новое направление в голографическом 3D TV.

Что лучше в долгосрочной перспективе – фотонная логика или спинтроника, пока непонятно:

- в обоих случаях поток частиц течёт с неизбежными тепловыми потерями, однако эти потери могут быть сколь угодно малы;
- в обоих случаях скорость потока ниже скорости света, однако, может быть сколь угодно близка к ней;
- в обоих случаях требуются сложные технические приёмы, чтобы наводки, тепловой шум и фоновое излучение (пролетающие мимо случайные частицы) не искажали сигнал.

Иными словами, известные физические ограничения обеих технологий одинаковы. По-видимому, наилучшие результаты будут давать комбинированные технологии: транспортировка фотонов выигрывает на длинных дистанциях и проигрывает на коротких; при этом уже разработаны рабочие прототипы переходников (в обе стороны) между поляризацией фотонного и электронного потока. Идеальным материалом для спинтронных дорожек является графен: он обеспечивает превосходную скорость при очень низких тепловых потерях. Кроме того, уже существуют рабочие прототипы следующих элементов:

- дорожек, надёжно сохраняющих поляризацию тока на коротких и средних дистанциях;
- ячеек памяти, способных сохранить поляризацию и затем формировать поток сохранённой поляризации.

Определённым прогрессом фотоники, достигнутым в 2013 г. стало создание совместными усилиями Гарварда и специалистов Российского квантового центра⁴⁷ устройства, в котором отдельный атом взаимодействует с фотонами из оптического волокна, причем с очень высокой эффективностью.

Устройство, которое они создали, можно назвать интерфейсом, то есть системой взаимодействия между фотонами, движущимися по оптическому волокну, и отдельным атомом. Атом в таком интерфейсе может, например, переключать пропускание фотонов через волокно, то есть работать как оптический транзистор, или же его можно использовать как некую квантовую ячейку, как элемент квантовой памяти. Получился универсальный инструмент, который подходит для работы как с классической информацией, так и с квантовой.

Сделать такую вещь весьма непросто. Во-первых, нужно уметь работать с отдельными атомами, двигать их, удерживать на определенном расстоянии от оптического волокна. И во-вторых, атомы, вообще говоря, очень плохо взаимодействуют с фотонами. Чтобы заставить их «разговаривать» друг с другом, пришлось создать специальный резонатор, расположенный на конце волокна. Именно он обеспечивает то, что в интерфейсе атом и фотон с вероятностью, практически в 100 процентов, будут взаимодействовать.

Интерфейс представляет собой оптическое волокно, которое на конце вытянуто в тоненькую, буквально в полмикрона, жилу. На нее приклеена примерно такого же размера структура прямоугольного сече-

⁴⁷ Совместность, традиционно, заключалась в том, что несколько наших специалистов уехали работать в Гарвард.

ния, это резонатор или, как мы его называем, фотонный кристалл. В нем «просверлены» дырки, он имеет форму, напоминающую хоккейную клюшку, просто по техническим причинам. И вот над этими дырками на расстоянии в 200 нанометров в вакууме висит атом.

Такие интерфейсы могут быть полезны как в обычной фотонике, так и в квантовой криптографии, о которой подробнее ниже.

Переключатели

На данный момент успешно созданы и протестированы транзисторы, состоящие из одной молекулы. Их скорость работы и компактность практически достигают теоретического предела, однако пока не существует технологий, которые позволяли бы их массово производить и использовать. Вся современная цифровая электроника изготавливается на основе металл-оксидных полевых транзисторов (MOSFET). Уже несколько десятилетий производители микросхем лишь уменьшают линейные размеры и энергопотребление таких транзисторов, однако уже были проведены промышленные испытания некоторых типов многостокковых и беспереходных транзисторов (BDT, JNT). Такие транзисторы смогут обеспечить повышение производительности в 4–6 раз. Этот переход не потребует сильных изменений технологического процесса и ожидается в ближайшие 5–10 лет.

Альтернативой транзистору может служить мемристор (*memristor*, от *memory* — «память», и *resistor* — «электрическое сопротивление») — пассивный элемент в микроэлектронике, способный изменять свое сопротивление в зависимости от протекавшего через него заряда (интеграла тока за время работы). Может быть описан как двухполюсник с нелинейной вольт-амперной характеристикой, обладающий гистерезисом.

Теория мемристора была создана в 1971 г. проф. Цай Шаотаном (Леонам Чуа в англ. транскрипции). Мемристор устанавливает отношения между интегралами по времени силы тока, протекающего через элемент, и напряжения на нем. Долгое время мемристор считался теоретическим объектом, который нельзя построить.

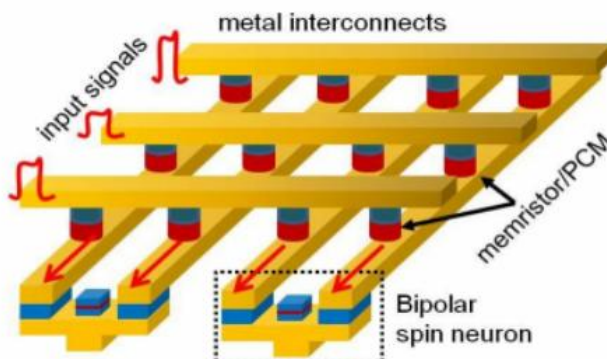
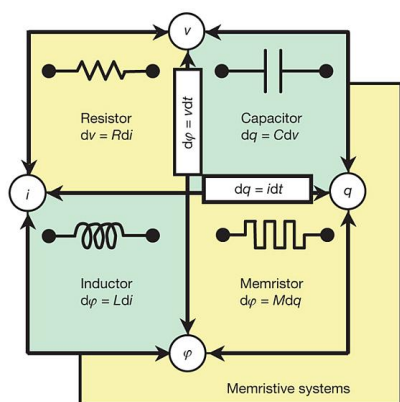
Существует два класса физических величин — измеряемые в точке (или в сечении как в математической плоскости, не важно) и между двумя точками (поверхностями). В электронике в точке (сечении) измеряются ток (I) и заряд (Q), а между точками — напряжение (U) и магнитный поток (Φ). Пассивные элементы электронных цепей как бы «материализуют» отношения на их границах (выводах), соотнося измеряемые в точке (на выводе) величины с измеряемыми между точками (выводами).

Науке и инженерии известны (до статьи Цай Шаотана) следующие элементы:

- резистор, сопротивление, - соотносит измеряемое между двумя точками (выводами) напряжение (в инженерной практике — падение напряжения) с измеряемым в точке (протекающим через вывод) током;
- конденсатор, емкость, - соотносит измеряемое между двумя точками (выводами) напряжение (в инженерной практике — падение напряжения) с измеряемым в точке зарядом;
- индуктивность - соотносит измеряемый между двумя точками (выводами) магнитный поток с измеряемым в точке током.

Цай предположил, что симметрия требует наличия недостающего элемента, соотносящего измеряемый между двумя точками (выводами) магнитный поток с измеряемым в точке зарядом. Для этого элемента Цай придумал название мемристор или мемистор. Мемистор можно было и в те времена реализовать с помощью разных схем, использующих активные элементы цепей (в первую очередь, операционные усилители).

Лабораторный образец мемристора был создан в 2008 г. коллективом ученых во главе с Р. С. Уильямсом в лаборатории фирмы Hewlett-Packard. В отличие от теоретической модели, устройство не накапливает заряд, подобно конденсатору, и не поддерживает магнитный поток, как катушка индуктивности.



Работа устройства обеспечивается за счет химических превращений в тонкой (5 нм) двухслойной пленке двуокиси титана. Один из слоев пленки слегка обеднен кислородом, и кислородные вакансии мигрируют между слоями под действием приложенного к устройству электрического напряжения. Данную реализацию мемристора следует отнести к классу наноионных устройств. Наблюдающееся в мемристоре явление гистерезиса позволяет использовать его в качестве ячейки памяти. В принципе, мемристоры могут заменить транзисторы во многих случаях, но такая возможность пока рассматривается только гипотетически.

Теоретически они могут быть более емкими и быстрыми, чем современная флеш-память. Также их блоки могут заменить RAM. Умение мемристоров «запоминать» заряд позволит отказаться от загрузки системы компьютера: в памяти компьютера, отключенного от питания, будет храниться его последнее состояние. Его можно будет включить и начать работу с того места, на котором остановился. Это же свойство позволит отказаться от некоторых компонентов современного ПК, что позволит сделать компьютеры компактнее и дешевле.

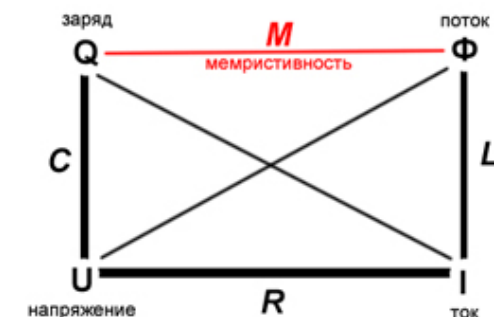
Технология на данный момент готова к производству компаниями Hynix и HP. Планировалось, что накопители на базе мемристоров выйдут в 2013 г., но выпуск был отменен, так как компании не хотят подрывать очень выгодный рынок флэш-памяти (SSD), которая по сравнению с мемристорами дорога и относительно ненадежна. Будет ли мемристорная память выпущена, пока не известно, скорее всего, процесс растянется еще лет на 10, пока не будет достигнут абсолютный предел флэш-технологии, и компании не выжмут из нее все возможные прибыли.

В 2012 г. специалисты из лаборатории Intel Circuit Research в Хиллсборо опубликовали работу с описанием нейроморфного чипа, дизайн которого основан на двух технологиях: поперечные вентили (lateral spin valves) и мемристоры. Поперечные вентили – крошечные магниты, которые реагируют на спин проходящих электронов. Мемристоры – это, попросту говоря, резисторы с памятью. По мнению разработчиков такой архитектуры, она позволяет сократить энергопотребление микросхем в 15-300 раз по сравнению с КМОП-микросхемами того же размера. Кроме того, нейроморфные чипы по своему дизайну больше подходят для задач того же типа, с которыми лучше всего справляется мозг: обработка аналоговой информации, когнитивные вычисления, ассоциативная память и др.

Нейристоры можно рассматривать как усовершенствованный вариант мемристоров. Они представляют интерес не только для исследовательских работ, но и для разработки нейрокомпьютеров, поскольку соответствуют математической модели Ходжкина-Хаксли. Они наиболее точно воспроизводят особенности передачи нервного импульса посредством потенциала действия.

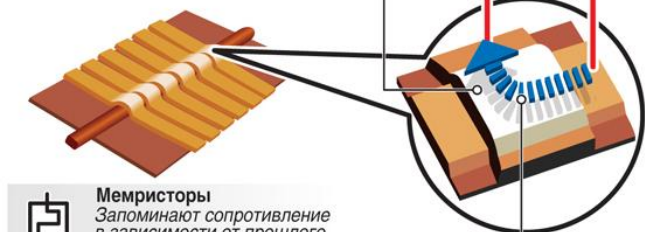
Исследователи рассчитывают получить с помощью нейристоров возможность моделировать более вариативное соотношение состояний – так называемый нейронный код. Физической основой такой способности является свойство гистерезиса, присущее как нервным клеткам, так и мемристорам. У тех и других отклик на внешнее воздействие зависит не только от текущего состояния, но и от поведения всей системы в прошлом. В случае мемристора его сопротивление зависит от предыдущей величины приложенного к нему напряжения. Подобно нервной системе, большая группа нейристоров сможет демонстрировать и некую толику вариативности. Это будет достигаться за счёт естественного разброса параметров компонентов и динамики условий их работы.

Кроме того, учёным удалось создать трёхмерный массив таких элементов, способный выполнять логические операции и работающий аналогично синапсам – сигнальным линиям между нейронными клетками



ПРИНЦИП ДЕЙСТВИЯ

Часть микросхемы, на которой размещены мемристоры



Мемристоры
Запоминают сопротивление в зависимости от прошлого воздействия
Размер: 2 нанометра

Транзисторы
Функционируют лишь когда через них проходит ток
Размер: 30 - 40 нанометров

Пленка из диоксида титана
Мемристор
Электрический ток способен изменить структуру диоксида титана, меняя сопротивление элемента в миллион раз. Эффект сохраняется до тех пор, пока не будет приложено новое внешнее воздействие

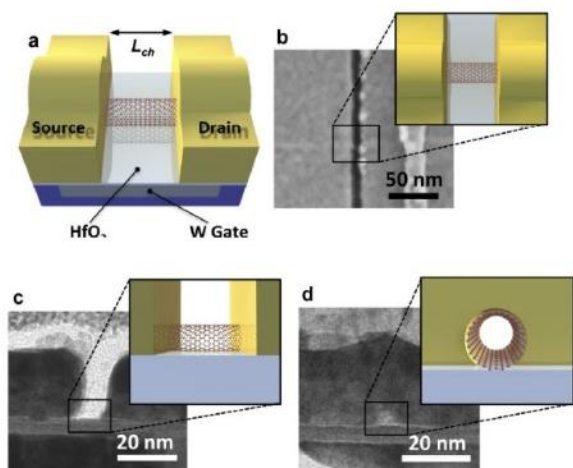
Отрывок из научно-популярного введения в проблему

в мозгу человека. Скорость передачи сигнала по синапсу зависит от времени активации нейронов: чем меньше временной промежуток между активацией, тем быстрее передается сигнал по синапсу. Точно так же работает и массив мемристоров: при подаче тока с промежутками в 20 мс сопротивление мемристора вдвое меньше, чем при 40-мс промежутках.

Уже изученные свойства мемристоров позволяют говорить о том, что на их основе можно создавать компьютеры принципиально новой архитектуры, по производительности значительно превышающие полупроводниковые. В компьютере на базе мемристоров каждый блок получает, перерабатывает и передает в другие блоки свои массивы данных.

В мемристорном компьютере параллельно и независимо друг от друга работают множество модулей, а возможность запоминать и оперировать неограниченным множеством значений от 0 до 1 означает, что исполняемые программы не ограничены двоичным кодом. Более того, станут в принципе ненужными отдельные аппаратные компоненты компьютера – процессоры, видеоипы, память и жёсткие диски; машина будет архитектурно однородным устройством, где одновременно будут храниться все данные и проводиться все операции с ними. Для апгрейда достаточно будет установить дополнительные мемристорные модули, а для ремонта – заменить вышедшие из строя.

Мемристорный компьютер не надо будет загружать: сразу после включения он будет готов продолжить работу, причём с того самого места, на котором она была прервана. По сравнению с современной техникой, энергопотребление мемристорных машин будет ничтожным.



Что из этого воплотится в реальности – не известно, но существует один класс вычислителей будущего поистине легендарный, настолько, что не упомянуть его просто невозможно. Это квантовые компьютеры.

2.2.4. Квантовые вычисления

Идея использования квантовых вычислений впервые была высказана советским математиком Ю.И. Маниным в 1980 году в его знаменитой монографии «Вычислимое и невычислимое». Правда, интерес к его труду возник лишь два года спустя, в 1982 году, после опубликования статьи на ту же тему американского физика-теоретика нобелевского лауреата Ричарда Фейнмана. Он заметил, что определенные квантово-механические операции нельзя в точности переносить на классический компьютер. Это наблюдение привело его к мысли, что подобные вычисления могут быть более эффективными, если их осуществлять при помощи квантовых операций.

Рассмотрим, к примеру, квантово-механическую задачу об изменении состояния квантовой системы, состоящей из n спинов, за определенный промежуток времени. Общее состояние системы из n спинов описывается вектором в 2^n -мерном комплексном пространстве, а изменение ее состояния – унитарной матрицей размером $2^n \times 2^n$. Если рассматриваемый промежуток времени очень мал, то матрица устроена очень просто, и каждый из ее элементов легко вычислить, зная взаимодействие между спинами. Если необходимо узнать изменение состояния системы за большой промежуток времени, то нужно перемножить такие матрицы, причем для этого требуется экспоненциально большое количество операций. Так мы сталкиваемся с PNP-полной задачей, нерешаемой за полиномиальное время на классических компьютерах. Но если классические компьютеры не способны решать квантовые задачи, то, возможно, для этого целесообразно использовать саму квантовую систему? И если это действительно возможно, то подходят ли квантовые системы для решения других вычислительных задач? Подобные вопросы как раз и рассматривались Фейнманом и Маниным.

Уже в 1985 году Дэвид Дойч предложил конкретную математическую модель квантовой машины, однако вплоть до середины 90-х годов направление квантовых вычислений развивалось довольно вяло. Практическая реализация квантовых компьютеров оказалась весьма сложной. К тому же в научном сообществе с пессимизмом относились к тому, что квантовые операции способны ускорить решение вычислительных задач. Так продолжалось вплоть до 1994 года, когда американский математик Питер Шор предложил для квантового компьютера алгоритм разложения n -значного числа на простые множители за время, полиномиально зависящее от n (квантовый алгоритм факторизации). Квантовый алгоритм факторизации Шора стал одним из основных факторов, приведших к интенсивному развитию квантовых методов вычислений и появлению алгоритмов, позволяющих решать некоторые NP-проблемы.

Естественно, возникает вопрос: почему, собственно, предложенный Шором квантовый алгоритм факторизации привел к таким последствиям? Дело в том, что задача разложения числа на простые множители

Естественно, не стоит забывать главную мораль этой книги – гладко было на бумаге. Огромное количество технологий за последние 50 лет готовились совершить переворот в ИТ. Действительно прорывных изобретений наберется не более 5-10, и среди них нет ничего, что изменило бы hardware радикально и навсегда. Посмотрим, повторит ли мемристор судьбу прочих забытых изобретений или действительно станет провозвестником новой компьютерной революции, каким стал транзистор.

Если рассматривать не столь отдаленное будущее, то еще в 2011 IBM продемонстрировала первый транзистор, длина затвора которого меньше 10 нм. Таким образом, компания обошла самые совершенные на сегодняшний день технологии. В транзисторе используются углеродные нанотрубки.

имеет прямое отношение к криптографии, в частности к популярным системам шифрования RSA. Благодаря возможности разложения числа на простые множители за полиномиальное время квантовый компьютер теоретически позволяет расшифровывать сообщения, закодированные при помощи многих популярных криптографических алгоритмов, таких как RSA. До сих пор этот алгоритм считался сравнительно надежным, так как эффективный способ разложения чисел на простые множители для классического компьютера в настоящее время неизвестен. Шор придумал квантовый алгоритм, позволяющий разложить на простые множители n -значное число за $n^3 \cdot (\log(n))^{const}$ шагов. Естественно, практическая реализация такого алгоритма может иметь скорее негативные, чем позитивные последствия, поскольку позволяет подбирать ключи к невзламываемым пока шифрам, подделывать электронные подписи и т.п.

Наш мир представляется неискущённому взгляду как набор вещей, каждая из которых состоит из более мелких вещей, и т.д. В комнате есть столы, стулья, шкафы, компьютеры и прочая мебель, каждый из этих предметов состоит из атомов, каждый атом состоит из ядра и электронов, дальше можно не продолжать. Если хочешь узнать всё о данном предмете, раздели его (мысленно) на возможно более мелкие составные части и изучи каждую часть по отдельности. Знание обо всём предмете представляет собой сумму знаний о его частях.

Такую концепцию устройства мира можно назвать «механической», так как максимальную свою мощь эта концепция проявила в механике. Механика расправляется с природой по принципу «разделяй и властвуй»: выделим, например, в Солнечной системе составные части – планеты, и для каждой планеты найдём её характеристики – массу, скорость, координаты. Объединив эту информацию, получим «состояние Солнечной системы» в данный момент времени. Теперь можно по точным механическим законам вычислять состояние Солнечной системы когда угодно – в прошлом, в будущем, и предсказывать что угодно – затмения, парады планет.

В квантовой механике, однако, нужна другая концепция. Даже в простейшем случае системы, состоящей из двух частей, знание о системе как целом не может быть получено складыванием знаний о её частях по отдельности. Так, в случае системы двух электронов мы имеем важное знание о системе в целом: её полный спин равен нулю. Если же рассматривать любой из двух электронов по отдельности, то о его спиновом состоянии можно сказать только, что он неполяризован (то есть его спин с равной вероятностью равен $\pm 1/2$). Но, зная о двух электронах, что они неполяризованы, не получишь, сложив эти знания, что полный спин равен нулю! (Если каждое из двух слагаемых с равной вероятностью либо $1/2$, либо $-1/2$, то сумма не обязана равняться нулю.) Таким образом, полное знание о системе включает в себя, помимо знания обо всех её частях, ещё нечто. Это нечто и называют «нелокальными корреляциями» между частями системы.

И вот оказывается, что квантовомеханическая концепция устройства мира, включающая в себя нелокальные корреляции, способна породить идею исключительной важности для вычислительных технологий. Идея состоит в том, чтобы использовать это «нечто», – нелокальные корреляции между квантовыми элементами памяти для хранения промежуточной информации.

В нелокальные корреляции можно записать экспоненциально большой объём информации, то есть при количестве элементарных составляющих квантового компьютера (кубитов), равном 1000, что, по большому счету немного – система из 1000 сцепленных частиц, мы имеем ОЗУ размером порядка 2^{1000} бит. Такие объёмы памяти невозможны, с точки зрения классического подхода, потому что это число больше, чем число атомов во всей Вселенной. Далее, с точки зрения механической концепции, даже если мы располагаем такой большой памятью, использовать её эффективно всё равно не получится: каждое элементарное действие компьютера затрагивает минимум одну, максимум 2-3 ячейки памяти, так что для заполнения 2^{1000} бит памяти необходимо совершить не меньше 2^{1000} элементарных операций. Но число 2^{1000} превосходит количество всех процессов (включая, например, соударения атомов) с момента возникновения Вселенной.

Здесь снова приходит на помощь квантовомеханическое мировоззрение: так как нелокальные корреляции принадлежат не отдельным составным частям устройства (кубитам), а всему устройству в целом, то каждая манипуляция с одним-двумя кубитами перегружает сразу всю ОЗУ нашего квантового компьютера. Подробное исследование показало, что достаточно уметь выполнять операции над каждым кубитом по отдельности и над каждой парой кубитов, чтобы выполнять полномасштабные квантовые вычисления, решающие экспоненциально сложные задачи за полиномиальное время. Ещё, конечно, для работы квантового компьютера нужно уметь записывать и считывать информацию с отдельных кубитов.

Математически квантовые вычисления описываются с использованием символов бра и кет (*bra-ket* ← *bracket* скобка) – алгебраического формализма описания квантовых состояний, предложенного Дираком. Вспомним алгебру из первой части – её различные структуры – кольца, поля и т.д. Помимо таких относительно простых конструкций, в математике существуют и более сложные – классический пример – векторное (или линейное) пространство L . Суть его состоит в том, что сначала мы вводим вспомогательную конструкцию – например поле действительных чисел со всеми аксиомами, а затем рассматриваем объекты (вектора), состоящие из элементов нашего поля. Все это, в принципе, очевидно, просто для использования необходимо переформулировать более строго.

Линейным наше векторное пространство называется, так как обладает дополнительной аксиомой линейности – мы можем не только складывать и умножать сами вектора, но и умножать их на элементы поля, т.е. числа, при этом $\forall \vec{l}_1, \vec{l}_2 \in L$ и $a \in \mathbb{R}$ $a \cdot (\vec{l}_1 + \vec{l}_2) = a \cdot \vec{l}_1 + a \cdot \vec{l}_2$.

Теперь рассмотрим множество однозначных непрерывных комплексных функций от действительных переменных $\psi_a(x)$. Функции будем различать символом a , который назовем индексом функции. Символ x будет обозначать совокупность действительных переменных, его мы назовем индексом представления. Для простоты можно понимать под x и обычную координату.

Множество функций ψ_a одного и того же аргумента обладает свойствами, аналогичными свойствам обычных векторов, их можно складывать, вычитать, умножать на комплексные числа, со всеми соответствующими аксиомами. Таким образом, функции образуют линейное векторное пространство, являясь своеобразными «векторами».

Рассмотрим функцию с индексом a . Обычный вектор можно без проблем записать через его координаты (т. н. разложение по базису): $\vec{a} = a_x \cdot \vec{i} + a_y \cdot \vec{j} + a_z \cdot \vec{k}$, иначе

$$\vec{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$$

Введем следующее обозначение. Нашу функцию-вектор ψ_a обозначим $|a\rangle$, соответственно, т.к. значений функции от всех возможных x бесконечно много, вектор представится как

$$|a\rangle = \begin{pmatrix} \vdots \\ \psi_a(x) \\ \vdots \end{pmatrix}$$

Такой вектор принято называть кет-вектор a . Возникает вопрос, а что будет, если столбец вектора транспонировать и превратить в строку, а затем, поскольку у нас комплексные числа, поменять их на сопряженные? Тогда мы получим так называемый эрмитово сопряженный ему вектор – бра-вектор $\langle a|$:

$$\langle a| = (\dots \psi_a^*(x) \dots)$$

Такие векторы образуют т.н. сопряженное (в определенном грубом смысле - перпендикулярное) пространство L^* . Эрмитово сопряжение обозначается знаком $+$ сверху, т.е. $\langle a| = |a\rangle^+$. Так как кет- и бра-векторы принадлежат разным пространствам, их нельзя смешивать и просто так проводить над ними операции. Зато любым двум таким векторам можно сопоставить их скалярное произведение: $\langle a|b\rangle$.

По аналогии с обычными векторами:

$$\langle a|b\rangle = \int_{-\infty}^{+\infty} \psi_a(x) \psi_b^*(x) dx$$

Также для таких векторов можно ввести норму, $\|a\| = \sqrt{\langle a|a\rangle}$. Множество кет-векторов с конечной нормой образует важный объект функционального анализа – гильбертово пространство, обозначаемое L^2 . Далее работа с таким бесконечномерным функциональным пространством происходит как с обычным векторным – выбирается базис и т.п. Если в гильбертовом пространстве задано правило сопоставления одного вектора другому (аналог функции, сопоставляющей число другому числу), то говорят что в пространстве действует оператор $\hat{L}$:

$$|a\rangle \xrightarrow{\hat{L}} |La\rangle$$

или в другой записи: $\hat{L} |a\rangle = |La\rangle$. Наконец, с операторами можно производить разные действия и получать тем самым алгебру операторов. Для бра-векторов все так же.

Идея работы классического компьютера заключается в том, что он, в любой момент времени, может находиться лишь в одном из фиксированных базисных состояний - $|0\rangle, |1\rangle, \dots, |n-1\rangle$. Квантовый же компьютер находится во всех этих состояниях одновременно, при этом в каждом состоянии $|j\rangle$ - со своей комплексной амплитудой λ_j .

Оно называется квантовой суперпозицией⁴⁸ данных классических состояний.

⁴⁸ Если функции ψ_1 и ψ_2 являются допустимыми волновыми функциями, описывающими состояние квантовой системы, то их линейная суперпозиция $\psi = c_1\psi_1 + c_2\psi_2$ также описывает какое-то состояние данной системы. Если измерение какой-либо физической величины $\hat{f}$ в состоянии ψ_1 приводит к результату f_1 , а в состоянии ψ_2 приводит к результату f_2 , то измерение в ψ приведет к результату f_1 или f_2 с вероятностями $|c_1|^2$ и $|c_2|^2$ соответственно. Любая наблюдаемая величина (например, положение, импульс или энергия частицы) является собственным значением эрмитова линейного оператора, соответствующим конкретному собственному состоянию этого оператора, то есть определенной волновой функции, действие оператора на которую сводится к умножению на число — собственное значение. Линейная комбинация двух волновых функций — собственных состояний оператора также будет описывать реально существующее физическое состояние системы. Однако для такой системы наблюдаемая величина уже не будет иметь конкретного значения, и в результате измерения будет получено одно из двух значений с вероятностями, определяемыми квадратами коэффициентов (амплитуд), с которыми базисные функции входят в линейную комбинацию. (Разумеется, волновая функция системы может быть линейной комбинацией и более чем двух базисных состояний, вплоть до бесконечного их количества).

Записывается такая суперпозиция, как:

$$|\Psi\rangle = \sum_{j=0}^{N-1} \lambda_j |j\rangle$$

Квантовое состояние $|\Psi\rangle$ может изменяться во времени двумя принципиально различными путями: унитарная квантовая операция (квантовый вентиль, *quantum gate*), в дальнейшем просто операция или просто измерение (наблюдение).

Если классические состояния $|j\rangle$ есть пространственные положения группы электронов в квантовых точках⁴⁹, управляемых внешним полем V , то унитарная операция есть решение уравнения Шредингера для этого потенциала. Измерение же есть случайная величина, принимающая значения $|j\rangle$ с вероятностями $|\lambda_j|^2$ соответственно. Квантовое вычисление есть контролируемая классическим управляющим компьютером последовательность унитарных операций⁵⁰ простого вида (над одним, двумя или тремя кубитами). В конце вычисления состояние квантового процессора измеряется, что и дает искомый результат вычисления.

Данные в процессе вычислений представляют собой квантовую информацию, которая по окончании процесса преобразуется в классическую путём измерения конечного состояния квантового регистра. Выигрыш в квантовых алгоритмах достигается за счёт того, что при применении одной квантовой операции большое число коэффициентов суперпозиции квантовых состояний, которые в виртуальной форме содержат классическую информацию, преобразуется одновременно.

Идея квантовых вычислений состоит в том, что квантовая система из L двухуровневых квантовых элементов (квантовых битов, кубитов) имеет 2^L линейно независимых состояний, а значит, вследствие принципа квантовой суперпозиции, пространство состояний такого квантового регистра является 2^L -мерным гильбертовым пространством. Операция в квантовых вычислениях соответствует повороту вектора состояния регистра в этом пространстве. Таким образом, квантовое вычислительное устройство размером L кубит фактически задействует одновременно 2^L классических состояний.

Физическими системами, реализующими кубиты, могут быть любые объекты, имеющие два квантовых состояния: поляризационные состояния фотонов, электронные состояния изолированных атомов или ионов, спиновые состояния ядер атомов, и т. д.

Если у нас имеется система из нескольких кубитов, и мы измеряем не все, а лишь первый, то результат такого измерения невозможно записать как вектор в гильбертовом пространстве состояний. Такое состояние, в котором участвует наше незнание о том, какой же результат получится на втором кубите, называют смешанным состоянием. В нашем случае такое смешанное состояние называют проекцией исходного состояния $|\Psi\rangle$ на второй кубит и записывают в виде матрицы плотности вида $\rho_2 = p_0 \cdot \rho_{\Psi_0} + p_1 \cdot \rho_{\Psi_1}$.

Упрощённая схема вычисления на квантовом компьютере выглядит так: берется система кубитов, на которой записывается начальное состояние. Затем состояние системы или её подсистем изменяется посредством унитарных преобразований, выполняющих те или иные логические операции. В конце измеряется значение, и это результат работы компьютера. Роль проводов классического компьютера играют кубиты, а роль логических блоков классического компьютера играют унитарные преобразования. Такая концепция квантового процессора и квантовых логических вентилях была предложена в 1989 г. Дэвидом Дойчем. Также Дойч в 1995 г. нашёл универсальный логический блок, с помощью которого можно выполнять любые квантовые вычисления.

Оказывается, что для построения любого вычисления достаточно двух базовых операций. Квантовая система дает результат, только с некоторой вероятностью являющийся правильным. Но за счет небольшого увеличения операций в алгоритме можно приблизить вероятность получения правильного результата к единице. С помощью базовых квантовых операций можно симулировать работу обычных логических элементов, из которых сделаны обычные компьютеры. Поэтому любую задачу, которая решена сейчас, квантовый компьютер решит, и почти за такое же время. Следовательно, новая схема вычислений будет не слабее нынешней.

Чем же квантовый компьютер лучше классического? Большая часть современных ЭВМ работают по такой же схеме: n бит памяти хранят состояние, и каждый такт времени изменяются процессором. В квантовом случае система из n кубитов находится в состоянии, являющимся суперпозицией всех базовых состояний, поэтому изменение системы касается всех 2^n базовых состояний одновременно. Теоретически новая схема может работать в экспоненциальное число раз быстрее классической. Практически, например, квантовый алгоритм Гровера поиска в базе данных показывает квадратичный прирост мощности против классических алгоритмов.

⁴⁹ Квантовая точка — фрагмент проводника или полупроводника, носители заряда которого ограничены в пространстве — один из главных кандидатов для представления кубитов в квантовых вычислениях. Размер квантовой точки должен быть настолько мал, чтобы квантовые эффекты были существенными. Это достигается, если кинетическая энергия электрона заметно больше всех других энергетических масштабов: в первую очередь больше температуры, выраженной в энергетических единицах. В этом одна из сложностей создания квантового компьютера — эта система работает лишь при очень низких температурах, тут-то и приносятся не выделяющие тепла обратимые операции.

⁵⁰ Унитарный оператор — ограниченный линейный оператор $U: H \rightarrow H$ на гильбертовом пространстве H , который удовлетворяет соотношению $U^*U = UU^* = I$, где звездочка — знак эрмитова сопряжения, а I — единичный оператор.

Множества задач, допускающих решение на квантовом компьютере и на классическом, совпадают, квантовый компьютер, таким образом, не увеличивает число алгоритмически разрешимых задач. Весь смысл применения квантового компьютера в том, что некоторые задачи он способен решить существенно быстрее, чем любой из классических.

Любая задача, решаемая квантовым алгоритмом, может быть решена и классическим компьютером путем прямого вычисления унитарных матриц экспоненциальной размерности, получения явного вида квантовых состояний. В частности, проблемы, неразрешимые на классических компьютерах (например, проблема остановки), остаются неразрешимыми и на квантовых. Но такое прямое моделирование требует экспоненциального времени, и потому возникает возможность, используя квантовый параллелизм, ускорять на квантовом компьютере некоторые классические алгоритмы в миллионы раз.

Один шаг квантового вычисления совершает гораздо большую работу, чем один шаг классического. Однако было бы ошибкой приравнивать квантовое вычисление к распараллеленному классическому. Например, квантовый компьютер не может решить задачу перебора быстрее, чем за $\sqrt{T_{cl}}$, где T_{cl} - время работы классического детерминированного алгоритма перебора, в то время, как недетерминированный решит ее за $\log(T_{cl})$. Но недетерминированный классический алгоритм требует экспоненциального ресурса памяти, то есть не является физически осуществимым, тогда как квантовый алгоритм не противоречит известным законам природы.

Языки квантового программирования позволяют выражать квантовые алгоритмы с использованием высокоуровневых конструкций. Их цель не столько создание инструмента для программистов, сколько предоставление средств для исследователей, для облегчения понимания работы квантовых вычислений. Среди существующих квантовых языков можно отметить QPL, QCL и Haskell-подобный функциональный QML.

Группа американских учёных, получив финансирование от исследовательского центра Национальной разведки США (IARPA) разработала высокоуровневый язык программирования Quipper. Он создан на основе Haskell и лучше подходит для реализации квантовых алгоритмов, чем QCL (основан на C)

На сегодняшний день известно как минимум 45 алгоритмов для квантовых компьютеров. Все они описаны в научных статьях, но ни один не был реализован в программном коде. С появлением Quipper появилась такая возможность. В дальнейшем программисты смогут просто использовать готовые библиотеки для квантовых компьютеров, как они это делают сейчас на высокоуровневых языках для классической архитектуры.

Разработчики Quipper призывают закодировать все известные алгоритмы на новом ЯП, сами они для проверки реализовали семь нетривиальных квантовых алгоритмов из литературы:

- Бинарное спаянное дерево (Binary Welded Tree, BWT) – поиск обозначенного узла в графе.
- Булева формула (Boolean Formula, BF) – любая формула AND-OR размера N может быть вычислена за время $\sqrt{N} + o(1)$ на квантовом компьютере. Реализована версия, которая вычисляет выигрышную стратегию в настольной игре гекс.
- Порядок класса (Class Number, CL) – квантовый алгоритм для вычисления уравнения Пелля и главного идеала за полиномиальное время.
- Оценка основного состояния квантово-механической системы (Ground State Estimation, GSE) – вычисление энергетических уровней для конкретной молекулы.
- Квантовые линейные системы (Quantum Linear Systems, QLS), – решение линейной системы уравнений.
- Кратчайший уникальный вектор (Unique Shortest Vector, USV) – поиск кратчайшего вектора среди имеющихся вариантов.
- Поиск треугольника (Triangle Finding, TF) – поиск треугольников в насыщенном графе.

Простейший алгоритм телепортации реализует точный перенос состояния одного кубита (или системы) на другой. В простейшей схеме используются 3 кубита: телепортируемый кубит и запутанная пара, один кубит которой находится на другой стороне. Отметим, что в результате работы алгоритма первоначальное состояние источника разрушится — это пример действия общего принципа невозможности клонирования — невозможно создать точную копию квантового состояния, не разрушив оригинал. Не получится скопировать произвольное состояние, и телепортация — замена этой операции. Телепортация позволяет передавать квантовое состояние системы с помощью обычных классических каналов связи. Таким образом, можно, в частности, получить связанное состояние системы, состоящей из подсистем, удаленных на большое расстояние.

Основные проблемы, связанные с созданием и применением квантовых компьютеров:

- необходимо обеспечить высокую точность измерений;
- внешние воздействия могут разрушить квантовую систему или внести в неё искажения.

Обе они более-менее эффективно решаются в течение последних 20 лет. В настоящее время построены только ограниченные варианты квантовой машины. Вопрос о том, до какой степени возможно масштабирование такого устройства, является предметом новой интенсивно развивающейся области — многочастичной квантовой механики. Центральным здесь является вопрос о природе декогерентности (точнее, о коллапсе волновой функции), который пока остается открытым.

В конце 2000-х г. во многих научных лабораториях были созданы однокубитные квантовые процессоры (по существу, управляемые двухуровневые системы, о которых можно было предполагать возможность масштабирования на много кубитов). Очень скоро был реализован жидкостной ЯМР-квантовый компьютер (до 7 кубит, IBM, И. Чанг).

В 2005 г. группой Ю. Пашкина при помощи японских специалистов был построен двухкубитный квантовый процессор на сверхпроводящих элементах. Примерно в это время до десятка кубит было сделано на ионах в ловушках Пауля (Д. Винланд, П. Золлер, Р. Блатт). В ноябре 2009 г. физикам из Национального института стандартов и технологий в США впервые удалось собрать программируемый квантовый компьютер, состоящий из двух кубит. В феврале 2012 г. компания IBM сообщила о достижении значительного прогресса в физической реализации квантовых вычислений с использованием сверхпроводящих кубитов, которые, по мнению компании, позволят начать работы по созданию квантового компьютера.

Наиболее известна и скандальна в этом отношении канадская фирма D-Wave Systems. Еще в феврале 2007 г. она заявила о создании образца квантового компьютера, состоящего из 16 кубит (устройство получило название Orion). Информация об этом устройстве не отвечала требованиям достоверного научного сообщения, поэтому новость не получила научного признания и была воспринята как шарлатанство. Более того, дальнейшие планы компании — создать уже в ближайшем будущем 1024-кубитный компьютер — вызвали скепсис у членов экспертного сообщества.

В ноябре 2007 г. та же компания продемонстрировала работу образца 28-кубитного компьютера (устройство получило название Leda) онлайн на конференции, посвященной суперкомпьютерам. Данная демонстрация также вызвала скепсис. 11 мая 2011 г. представлен компьютер D-Wave One, созданный якобы на базе 128-кубитного процессора и с 20 мая он доступен для покупки по цене \$11.000.000. Компьютер монозадачен и выполняет только дискретную оптимизацию. Первый вариант расположен в вычислительном центре Южно-Калифорнийского университета, его рабочая температура составляет 20 мК, компьютер тщательно экранирован от внешних электрических и магнитных полей.

В том же году Lockheed Martin подписала многолетний контракт с D-Wave Systems, касающийся выполнения сложных вычислительных задач на квантовых процессорах. Контракт также включает в себя техническое обслуживание, сопутствующие услуги и покупку квантового компьютера D-Wave One. В 2013 г. к числу купивших компьютер присоединились Google и NASA.

В то же время квантовые компьютеры D-Wave Systems подвергаются критике со стороны многих исследователей. Так, профессор Массачусетского Технологического Института Скотт Ааронсон считает, что D-Wave пока не смогла доказать, что ее компьютер решает какие-либо задачи быстрее, чем обычный компьютер, и что используемые 128 кубитов удастся ввести в состояние квантовой запутанности. Если же кубиты не находятся в запутанном состоянии, то это не квантовый компьютер.

Однако 23 августа 2012 года было объявлено об успешном решении задачи нахождения трехмерной формы белка по известной последовательности аминокислот в его составе с использованием 115 кубитов квантового компьютера D-Wave One из 128 имеющихся методом квантового отжига, и в декабре 2012 года представлен новый процессор Vesuvius, который объединяет 512 кубитов.

В мае 2013 г. профессор Колледжа Эмхерст (Новая Шотландия) Катерина МакГью (Catherine McGeoch) объявила о своих результатах сравнения компьютера D-Wave One (процессор Vesuvius) с четырехпроцессорным компьютером на основе 2,4 ГГц чипа Intel с 16 Гб оперативной памяти. В первом тесте одну из задач класса QUBO, хорошо подходящую для структуры процессора, компьютер D-Wave One выполнил за 0,5 секунды, в то время как компьютеру с процессором Intel потребовалось 30 мин. Во втором тесте требовалась специальная программа для «перевода» задачи на язык компьютера D-Wave, и скорость вычислений двух компьютеров была примерно равной. В третьем тесте, в котором также требовалась программа «перевода», компьютер D-Wave One за 30 минут нашёл решение 28 из 33 заданных задач, в то время как компьютер на процессоре Intel нашёл решение только для 9 задач.

В настоящее время сообщество сошлось на следующем. D-Wave монозадачен и не универсален. Программирование его в общепринятом смысле невозможно, работает он не с традиционными кубитами, а использует кватовый отжиг.

В математике и приложениях квантовым отжигом называется довольно общий метод нахождения глобального минимума некоторой заданной функции среди некоторого набора решений-кандидатов. Он преимущественно используется для решения задач, где поиск происходит по дискретному множеству с множеством локальных минимумов.

При квантовом отжиге текущее решение-кандидат случайным образом заменяется его соседом, если в том состоянии «энергия» (оптимизируемый функционал) меньше. Процесс регулируется параметром «напряжённость поля туннелирования», отвечающим за размер «просматриваемой» области. Изначально поле туннелирования достаточно сильное, поэтому поиск происходит по всему пространству. Потом напряжённость уменьшается, система оседает в нескольких состояниях с наименьшими энергиями. Если повезет, она найдет глобальный минимум и там останется. В пределе мы получаем классическую систему в одном из основных состояний

Теоретически было доказано, что для решения любой произвольной задачи существует алгоритм квантового отжига, но проблема кроется в нахождении этого алгоритма, что составляет нетривиальную зада-

чу, для решения которой опять нужен квантовый компьютер. Для некоторых простых случаев алгоритм удастся найти на обычном компьютере.

У квантового отжига есть такое ограничение. Если наращивать количество кубитов (модулей), то энергетический спектр системы будет уплотняться экспоненциально. А это значит, что время релаксации системы в основное состояние будет также экспоненциально возрастать. Т.е. от наращивания количества кубитов выигрыша в скорости решения задачи не будет.

Таким образом, D-Wave оказался монозадачной приставкой к традиционному компьютеру, ускоряющей решение узкого класса задач оптимизации вероятностными методами, т.е. заявленное достижение и реальность немного не совпали, хотя квантовым оптимизирующим сопроцессором D-Wave назвать можно.

Однако есть и позитивные мнения. Физик-экспериментатор и программист квантовых вычислений д-р Сьюзен Джилдерт (Suzanne Gildert) из Бирмингемского университета говорит, что D-Wave для построения своего квантового компьютера использует «естественные квантовые вычисления», которые очень сильно отличаются от традиционно предлагаемых схем и решают многие проблемы.

«Некоторые схемы квантовых вычислений пытаются реализовать идеи, почерпнутые из области традиционных компьютерных вычислений – такие как логические операции – и сделать их «квантовую» версию, что крайне трудно. Реализация «квантовых» версий компьютерных операций – процесс очень деликатный. Это подобно попытке удерживать в равновесии карандаш, поставленный на брусок дерева, и затем передвигать этот брусок. Это почти невозможно. Вы должны постоянно напрягаться, чтобы удержать карандаш (другими словами, кубит) в вертикальном положении. Падение карандаша – это нарушение когерентности в нашем контексте, – говорит Джилдерт. – В нашем подходе NQC, который более масштабируемый и надежный, мы, образно говоря, позволяем карандашу лежать на плоской поверхности бруска. Таким образом, мы избегаем проблем, связанных с потерей когерентности битами, поскольку состояние, в которое мы устанавливаем систему, является более естественным (вот почему мы называем эти вычисления NQC)».

Джилдерт добавляет, что основной областью применения квантового компьютера будет машинное самообучение, что вплотную связано с искусственным интеллектом. Для этого необходимо разрабатывать ПО, которое может обучаться на основании опыта, в противоположность современному статическому ПО. «Это радикально отличается от того, как мы используем вычисления сегодня, – говорит Джилдерт. – Смысл в том, что обучающееся ПО не является повсеместным. Когда мы более пристально рассмотрим машинное обучение, то обнаружим, что имеется несколько очень трудных и фундаментальных математических проблем, известных как оптимизационные проблемы. D-Wave строит машину, которая является попыткой решить эти трудные проблемы, открывая дверь полностью новому способу программирования и созданию полезной части кода».

«Маловероятно, чтобы квантовые вычисления заменили десктопы в скором времени, – говорит Джилдерт. – Если оценивать годы, то это зависит от инвестиций, фондов и людей, работающих над этой проблемой. Логично предположить, что эти машины будут служить облачными сопроцессорами в существующих ЦОД. Квантовые компьютеры очень хороши для решения специфического класса задач в области ИИ и машинного обучения, так что мы концентрируемся на создании инструментов, которые помогут использовать потенциал квантовых вычислений людям, работающим в этих областях».

Ей возражает Джим Талли (Jim Tully), вице-президент и главный исследователь в области электроники и полупроводников из Gartner Research, говорит, что у D-Wave на самом деле не истинно квантовые вычисления. «Подкласс квантовых вычислений, продемонстрированный D-Wave и называемый квантовый отжиг, включает суперпозицию, но не включает зацепление. Поэтому не является истинно квантовыми вычислениями. Квантовый отжиг потенциально полезен лишь для оптимизационных целей, особенно для очень быстрого нахождения минимума в наборе данных».

Одному из заказчиков этой машины, она, несмотря на все споры, нравится. Кейт Мордофф (Keith Mordoff), директор Communications Information Systems & Global Solutions из Lockheed Martin, говорит: «Да, у нас есть полностью функционирующий квантовый компьютер, который отличается от классических схем. D-Wave использует адиабатический подход, или квантовый отжиг, который задает сложную систему, чье основное состояние (состояние с наименьшей энергией) представляет решение поставленной проблемы. Сначала конструируется простая система и переводится в свое основное состояние (что сделать относительно несложно), затем она адиабатически изменяется до тех пор, пока не становится сложной. По мере того как система развивается, она остается в основном состоянии, затем просто измеряется состояние финальной системы. И оно будет ответом на поставленную проблему. Изменение от простой системы к сложной происходит за счет включения фонового магнитного поля».

Почему же некоторые ученые считают, что квантовый компьютер в традиционном смысле слова невозможен вообще?

В начале февраля 2013 г. физик MIT Скот Ааронсон (Scott Aaronson) в своем блоге предложил приз в \$100 тыс. тому, кто докажет, принципиальную невозможность создания масштабируемого квантового компьютера. Побудительной силой для заявления Ааронсона стала дискуссия между Жилем Калайи (Gil Kalai, Институт математики Иерусалимского Университета) и Арамом Херроу (Arav Hamow, Департамент математики Йельского Университета) в области квантовых методов обработки информации. Она была инспирирова

их знаменитой презентацией «Why Quantum Computers Cannot Work and How» опубликованной также в феврале 2013 г.⁵¹ В статье он подвергает квантовые вычисления строгой критике со следующих позиций.

Отказоустойчивые квантовые системы не могут быть построены, т.е. в процессе квантовой эволюции возникает неустраняемый шум – декогерентизация, приводящий к ошибкам вычислений, с которыми не в состоянии справиться принцип Ландауэра. Квантовая механика не работает с состояниями, она работает с вероятностями, оперирует величинами, которые описывают вероятность нахождения системы в некотором состоянии. Состояния постоянно меняются, т.е. система эволюционирует. И может сложиться такая ситуация, что вычисления еще не были закончены, а кубиты сменяют свои состояния сами по себе либо из-за воздействия сторонних помех, например, теплового шума.

Поясним, в чём разница между влиянием помех на работу классического и квантового компьютеров. Квантовый компьютер сочетает в себе черты цифровых и аналоговых ВМ. При цифровой обработке информации случайная помеха может привести к сбою, только если она превысит некоторый (легко увеличиваемый кодами коррекции ошибок до произвольного значения) порог, а вероятность такой большой помехи экспоненциально мала. Так что для ЭВМ всевозможные шумы и неидеальность аппаратуры не представляют принципиальной трудности. Путём усовершенствования аппаратуры можно добиться сколь угодно низкой вероятности ошибки, и есть методы исправления ошибок. При аналоговой обработке информации, наоборот, ошибки всё время накапливаются, и если прошла помеха, то её уже не устранить. Разницу можно почувствовать, если несколько раз перезаписать аудиозапись в цифровом (в компьютере, на CD) или в аналоговом (с кассеты на кассету) виде. Квантовый компьютер – это, по сути, аналоговое устройство. Но так как в нём есть и черты ЭВМ (представление информации в виде нулей и единиц), то можно найти способы исправления некоторых классов ошибок. Однако, традиционные алгоритмы тут не годятся, измерение-считывание разрушает состояние кубита.

Однако, проблема защиты от помех – это не единственная настоящая фундаментальная трудность на пути создания большого квантового компьютера. Физические корни проблем весьма глубоки и уходят в различие между микроскопическими и макроскопическими объектами. В цепочке «А состоит из В, В состоит из С, ...» есть огромная разница между двумя её концами, они говорят на разных языках – классическом и квантовом.

Можно привести множество критериев, различающих микро- и макрообъекты. Например, микрообъекты лишены индивидуальности (все электроны тождественны), тогда как каждый макрообъект вполне индивидуален. Микрообъект имеет характерный энергетический спектр (набор уровней), тогда как у макрообъекта нельзя выделить отдельные энергетические уровни – они все «слились» в сплошной континуум. Но для нас наиболее важно следующее. Макрообъект всё время флуктуирует, шумит. Например, между концами макроскопического проводника всегда имеется случайное напряжение, средний квадрат которого зависит от сопротивления проводника и от температуры (проводник при нулевой температуре не шумел бы, но абсолютный нуль недостижим).

У микрообъектов флуктуаций нет: атом не шумит. Флуктуации макроскопических тел приводят к тому, что ЭПР-корреляции между разными телами усредняются, то есть пропадают; так что именно благодаря непрерывным шумам мы воспринимаем окружающие нас (микроскопические) предметы как хорошо отделённые друг от друга и индивидуальные. Квантовый компьютер, таким образом, должен вести себя как микрообъект, иначе все корреляции пропадут, и он не сможет работать. Но набор из 1000 кубитов по своим масштабам ближе к макрообъектам. Так что, пытаясь заставить набор 1000 кубитов работать как квантовый компьютер, мы совершаем некоторое насилие: заставляем его говорить на неродном языке. Удастся ли это – неизвестно. Если удастся, то это будет большой прорыв в понимании квантовой механики, ведь все проблемы с её осознанием можно свести к тому, что квантовый объект и измерительный прибор требуют двух разных языков описания. Пока что представляется, что макро- и микромир не пересекаются: каждый физический объект ведёт себя либо макроскопически, либо микроскопически. В этой связи интересно, что физики пытаются представить себе переход от макро- к микроповедению как фазовый переход:

«Очень шумная квантовая система ведёт себя классически, и может эффективно моделироваться классической машиной Тьюринга, но если величина некогерентности достаточно низка, может возникнуть скрепленное квантовое состояние и никакое эффективное классическое моделирование системы уже невозможно... Для теоретика, знакомого с критическими явлениями, естественно задаться вопросом относительно универсальных характеристик этого фазового перехода... Решение этой задачи могло бы привести к новой физической реализации квантовых вычислений» – пишет Дж. Прескилл в книге «Квантовая информация и квантовые вычисления».

Надо отметить, впрочем, что есть и такие макрообъекты, которые ведут себя отчасти аналогично микрообъектам. Это так называемые бозе-конденсаты: сверхпроводники, сверхтекучие жидкости; сюда же примыкает и лазерное излучение. Так, поведение сверхпроводника (в теории Гинзбурга-Ландау) описывается макроскопической фазой, которая и по происхождению, и по свойствам близка к фазе волновой функции

⁵¹ Помимо этого известна полемическая статья «How Quantum Computers Fail: Quantum Codes, Correlations in Physical Systems, and Noise Accumulation» их же авторства.

микрочастицы. Отличие в том, что макроскопическая фаза сверхпроводника описывает коллективное движение микрочастиц; макроскопическую фазу можно измерять не разрушая. Может быть, использование бозе-конденсата станет перспективным направлением в создании квантового компьютера, и тогда дилемма микробъект-макробъект станет менее острой.

Еще одним интересным применением квантовой механики является т.н. квантовая криптография. В отличие от традиционной криптографии, которая использует математические методы для обеспечения секретности информации, квантовая криптография рассматривает случаи, когда информация переносится с помощью объектов квантовой механики. Процесс отправки и приёма информации всегда выполняется физическими средствами, например, при помощи электронов в электрическом токе или фотонов в линиях волоконно-оптической связи. А подслушивание может рассматриваться как измерение определённых параметров физических объектов — в нашем случае, переносчиков информации.

Используя квантовые явления, можно спроектировать и создать такую систему связи, которая всегда может обнаружить подслушивание. Попытка измерения взаимосвязанных параметров в квантовой системе вносит в неё нарушения, разрушая исходные сигналы, а значит, по уровню шума в канале легитимные пользователи могут распознать степень активности перехватчика. Для объяснения алгоритма воспользуемся широко распространенной схемой персонажей (Алиса и Боб обмениваются секретной информацией). Алиса посылает Бобу последовательность фотонных импульсов. Каждый из импульсов случайным образом поляризован в одном из четырех направлений: $\uparrow \leftrightarrow \nearrow \nwarrow$

Например, Алиса посылает: $\leftrightarrow \nearrow \nwarrow \uparrow$.

Боб настраивает свой детектор произвольным образом на измерение серии либо диагонально, либо ортогонально поляризованных импульсов (мерить одновременно и те и другие нельзя): $XX + XX + +$.

В тех случаях, где Боб угадал поляризацию, он получит правильный результат (такую же поляризацию, какую посылала Алиса). В остальных случаях результат будет случайным. Боб и Алиса по открытому каналу сообщают друг другу использованные типы поляризаций (диагональная или ортогональная) и оставляют только правильно измеренные. В нашем примере Боб угадал поляризацию 2-го, 5-го, 6-го и 7-го импульсов. Таким образом, остаются: $\nearrow \nwarrow \leftrightarrow \uparrow$.

По заранее оговоренным условиям эти результаты превращаются в последовательность битов (например, 0° и 45° принимаются за единицу, 90° и -45° — за ноль, в приведённом выше примере получится 1010). Перехват сообщения-ключа Боб и Алиса могут обнаружить посредством контроля ошибок, сверив случайно выбранные из сообщения биты. Несовпадения указывают на перехват сообщения, тогда ключ изменяется, то есть передается повторно. Если расхождений нет, то биты, использованные для сравнения, отбрасываются, ключ принимается. С вероятностью $1 - 2^{-k}$ (где k — число сравненных битов) канал не прослушивался. Впрочем, если недоброжелатель может не только прослушивать основной канал, но и фальсифицировать работу открытого канала, то вся схема рушится.

Описанный алгоритм носит название протокола квантового распределения ключа BB84. В нём информация кодируется в ортогональные квантовые состояния. Помимо использования ортогональных состояний для кодирования информации, можно использовать и неортогональные состояния (например, протокол B92).

В 2011 году в Токио прошла демонстрация проекта «Tokyo QKD Network», в ходе которого разрабатывается квантовое шифрование телекоммуникационных сетей. Была проведена пробная телеконференция на расстоянии в 45 км. Связь в системе идёт по обычным оптоволоконным линиям. Предполагается стабильная длительная эксплуатация, а также применение для мобильной связи.

Наконец, упомянем еще об одном квантовом эффекте — квантовой телепортации. Это передача квантового состояния на расстояние при помощи разъединённой в пространстве сцепленной пары и классического канала связи, при которой состояние разрушается в точке отправления при проведении измерения, после чего воссоздается в точке приёма.

В 1993 году шесть физиков — Беннет, Броссар и другие — написали в *Physical Review Letters* статью, в которой и придумали замечательную терминологию для квантовой телепортации. В их работе сам протокол передачи квантовой информации был описан чисто теоретически.

Квантовая телепортация не передаёт энергию или вещество на расстояние. Обязательным этапом при квантовой телепортации является передача информации между точками отправления и приёма по классическому, неквантовому каналу, которая может осуществляться не быстрее, чем со скоростью света, тем самым не нарушая принципов современной физики.

Квантовая телепортация осуществляется за счёт разделения информации на «квантовую часть» и «классическую часть» и независимой передаче этих двух компонентов. Для передачи «квантовой части» используются характерные для квантово-запутанных частиц корреляции Эйнштейна — Подольского — Розена, а для передачи классической информации годится любой обычный канал связи.

Для простоты будем иметь в виду квантовую систему с двумя возможными состояниями ψ_1 и ψ_2 (например, проекцию спина электрона или фотона на заданную ось). Такие системы часто называют кубитами. Однако описанный способ пригоден для передачи состояния любой системы, имеющей конечное число состояний.

Пусть у отправителя есть частица A , находящаяся в произвольном квантовом состоянии $\psi_A = \alpha\psi_1 + \beta\psi_2$, и он хочет передать это квантовое состояние получателю, то есть сделать так, чтобы у получателя оказалась в распоряжении частица B в том же самом состоянии. Иными словами, необходимо передать отношение двух комплексных чисел α и β с бесконечной точностью. Заметим, что главная цель здесь — это передать информацию не как можно быстрее, а как можно аккуратнее. Для достижения этой цели выполняются следующие шаги.

Отправитель и получатель договариваются заранее о создании пары квантово-запутанных частиц C и B , причём C попадёт отправителю, а B — получателю. Поскольку эти частицы запутаны, то каждая из них не обладает своей волновой функцией (вектором состояния), но вся пара целиком (а точнее, интересующие нас степени свободы) описываются единым четырёхмерным вектором состояния ψ_{BC} .

Квантовая система частиц A и C имеет четыре состояния, однако мы не можем описать её состояние вектором — чистым (полностью определённым) состоянием обладает лишь система из трёх частиц A, B, C . Когда отправитель совершает измерение, имеющее четыре возможных исхода над системой из двух частиц A и C , он получает одно из 4 собственных значений измеряемой величины. Поскольку при этом измерении система из трёх частиц A, B, C коллапсирует в некое новое состояние, причём состояния частиц A и C становятся известны полностью, то сцепленность разрушается, и частица B оказывается в некотором определённом квантовом состоянии.

Именно в этот момент происходит как бы передача «квантовой части» информации. Однако восстановить передаваемую информацию пока невозможно: получатель знает, что состояние частицы B как-то связано с состоянием частицы A , но не знает, как именно!

Для выяснения этого необходимо, чтобы отправитель сообщил получателю по обычному классическому каналу результат своего измерения (затратив при этом два бита, соответствующие зацепленному состоянию AC , измеренному отправителем). По законам квантовой механики, имея результат измерения, проведённого над парой частиц A и C и плюс к тому запутанную с C частицу B , получатель сможет совершить необходимое преобразование над состоянием частицы B и восстановить исходное состояние частицы A .

Полная передача информации осуществится только после того, как получатель будет обладать данными, полученными по обоим каналам. До того, как получен результат по классическому каналу, получатель ничего не может сказать о переданном состоянии. Фантастическое понятие телепортации происходит из специфической интерпретации эксперимента: «исходное состояние частицы A после всего произошедшего разрушается. То есть, состояние было не скопировано, а перенесено из одного места в другое».

В 1997 году была осуществлена первая квантовая телепортация фотонов (на самом деле экспериментов было два — группы Заиллингера и Де Мартини; Заиллингера просто больше цитируют). В работе они телепортировали поляризацию фотонов — направление этой поляризации суть квантовая величина, то есть такая величина, которая принимает различные значения с разной вероятностью. Как оказалось, измерить эту величину нельзя, а вот телепортировать можно.

Тут надо вот что учесть: в экспериментах Заиллингера и Де Мартини телепортация была вероятностной, то есть работала с некоторой вероятностью успеха. Им удалось достичь вероятности не меньше $2/3$.

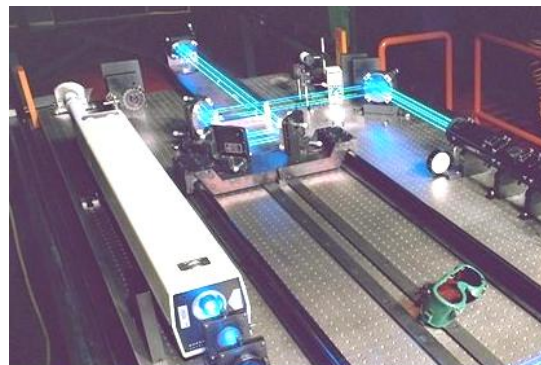
В 1998 году группе Юджина Ползика в Калифорнийском технологическом институте удалось осуществить так называемую детерминистскую телепортацию. У них телепортировались фаза и амплитуда светового импульса. Они, как говорят физики, так же как скорость и местоположение электрона, являются «некоммутирующими переменными», поэтому подчиняются уже упоминавшемуся принципу Гейзенберга, то есть не допускают одновременное измерение.

10 мая 2010 года в эксперименте, поставленном физиками из Научно-технического университета Китая и Университета Цинхуа, проводилась передача квантового состояния фотона на 16 км.

В июне 2013 года той же группе физиков из Калифорнии удалось провести эксперимент по детерминистской телепортации коллективного спина 10^{12} атомов цезия на полметра. Эта работа попала на обложку *Nature Physics*.

В реальности фотоны, путешествуя по квантовому каналу, теряются (например, если этот канал — обычное оптоволокно). Главное отличие между вероятностной и детерминистской телепортациями как раз и заключается в отношении к этим потерям. Вероятностной все равно, сколько там потерялось — если из миллиона фотонов хотя бы один дошел, то уже хорошо. В этом смысле, конечно, она больше подходит для пересылки фотонов на большие расстояния (в настоящее время рекорд составляет 143 километра).

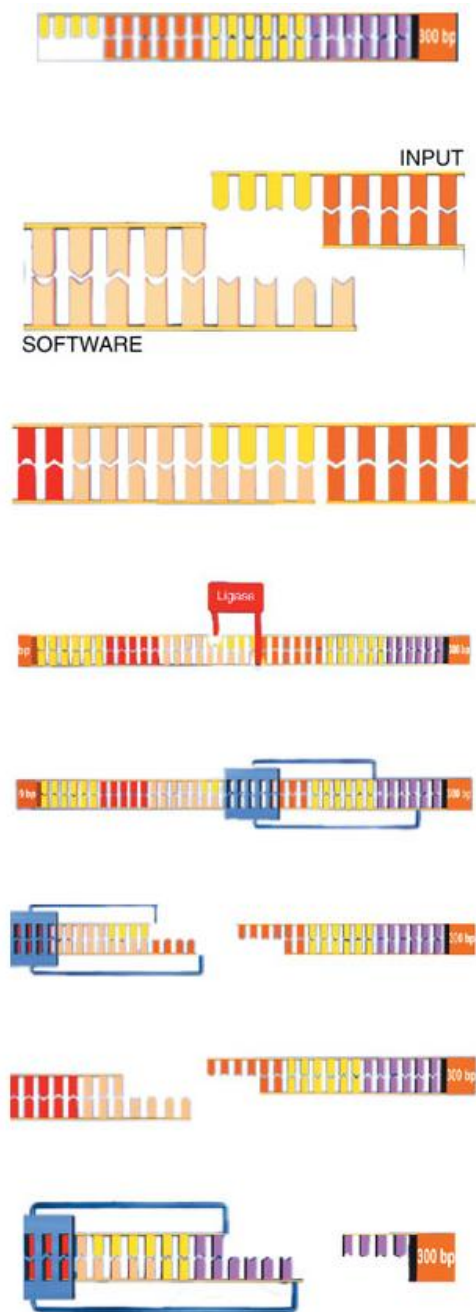
Детерминистская же телепортация к потерям относится хуже — вообще говоря, чем выше потери, тем хуже качество телепортации, то есть на принимающем конце провода получается не совсем исходное квантовое состояние — но зато она работает каждый раз, когда нажимаешь на кнопку.



Оптический стол с лазером, зеркалами и другими устройствами для телепортационных экспериментов в NASA

2.2.5. Молекулярные машины

Последней технологией, которую мы рассмотрим являются т.н. молекулярные и ДНК-компьютеры. Успехи здесь самые скромные, а перспективы самые туманные.



Молекулярные компьютеры — вычислительные системы, использующие вычислительные возможности молекул (преимущественно, органических). Молекулярными компьютерами используется идея вычислительных возможностей расположения атомов в пространстве.

Под молекулярным компьютером обычно понимают такие системы, которые используют отдельные молекулы в качестве элементов вычислительного тракта. В частности, молекулярный компьютер может представлять логические электрические цепи, составленные из отдельных молекул; транзисторы, управляемые одной молекулой, и т. п.

Также молекулярным компьютером могут назвать ДНК-компьютер, вычисления в котором соответствуют различным реакциям между фрагментами ДНК. От классических компьютеров такие отличаются тем, что химические реакции происходят сразу между множеством молекул независимо друг от друга.

В 1997 году Леонард Адлеман, профессор университета Южной Калифорнии, продемонстрировал, что с помощью пробирки с ДНК можно весьма эффективно решать классическую комбинаторную «задачу о коммивояжере» (кратчайший маршрут обхода вершин графа). Классические компьютерные архитектуры требуют множества вычислений с опробованием каждого варианта.

Метод ДНК позволяет сразу сгенерировать все возможные варианты решений с помощью известных биохимических реакций. Затем возможно быстро отфильтровать именно ту молекулу-нить, в которой закодирован нужный ответ.

Проблемы, возникающие при этом:

- требуется чрезвычайно трудоёмкая серия реакций, проводимых под тщательным наблюдением;
- существует проблема масштабирования задачи.

Было подсчитано, что при масштабировании методики Адлемана для решения задачи обхода надо не 7 пунктов, а около 200, масса количества ДНК, необходимого для представления всех возможных решений превысит массу нашей планеты. В результате машина Адлемана так и осталась концептом. Еще один прототип - конечный биоавтомат Бененсона-Шапиро - технология многоцелевого ДНК-компьютера, разрабатываемая израильским профессором Эхудом Шапиро (Ehud Shapiro) и Яковом Бененсоном из Вейцмановского института. Его основой являются уже известные свойства биомолекул, таких как ДНК и ферменты. Функционирование ДНК-компьютера сходно с функционированием конечного автомата.

Разработанный Эхудом Шапиро и его коллегами биокомпьютер требует для работы лишь составления правильной молекулярной смеси. Затем примерно за час эта смесь самостоятельно порождает молекулу ДНК, в которой закодирован ответ на постав-

ленную перед вычислителем несложную задачу. В этом биокомпьютере ввод и вывод информации, а также роль «программного обеспечения» берут на себя молекулы ДНК. В качестве же «аппаратного обеспечения» выступают два белка-энзима естественного происхождения, которые манипулируют нитями ДНК. При совместном замешивании молекулы программного и аппаратного обеспечения гармонично воздействуют на молекулы ввода, в результате чего образуются выходные молекулы с ответом. В целом же система функционирует как простой конечный автомат.

Задачи, которые способен решать этот автомат, зависят от подаваемых на вход молекул и от молекул программного обеспечения. Пользуясь формальным языком, сейчас автомат способен обрабатывать «регулярные выражения», т. е. находить ответы на несложные вопросы относительно содержания списков, содержащих два типа символов, таких как «0» и «1», или «a» и «b». Например, четно ли число единиц в последовательности? Или есть ли в



последовательности по крайней мере один символ «b»? В общей сложности биомолекулярный нанокomпьютер израильских ученых сейчас можно запрограммировать на отыскание ответов для 756 подобных вопросов.

Понятно, что реализованный группой Шапиро конечный автомат - это пока что достаточно узкоспециализированное устройство, существенно уступающее по своим возможностям современным компьютерам общего назначения. Однако, полагают ученые, в течение ближайшего десятилетия биомолекулярные устройства такого типа вполне можно будет научить выполнению нетривиальных лабораторных приложений, а еще через несколько десятилетий станут возможны и медицинские приложения. «Живая клетка содержит совершенно немыслимые молекулярные машины, которые манипулируют кодирующими информацию молекулами типа ДНК и РНК. Делается это с помощью способов, которые на фундаментальном уровне чрезвычайно похожи на компьютерные вычисления, - говорит Шапиро. - Пока что мы просто не знаем, каким образом можно эффективно модифицировать эти машины или по аналогии создавать собственные. Весь трюк заключается в том, чтобы отыскать эти естественно существующие машины и принципы их работы, чтобы начать их комбинировать и приспособлять для нужных нам вычислений».

Израильские ученые вовсе не ставят перед собой задачу создать устройство, конкурирующее в эффективности с традиционными компьютерами. Заставить ДНК

работать в качестве полноценного микропроцессора - эта задача пока еще очень далека от разрешения, и маловероятно, что в обозримом будущем мы отправимся в ближайший компьютерный магазин покупать ПК на основе ДНК.

Ещё в 1959 г. неугомонный Ричард Фейнман, поучаствовавший в теоретическом обосновании почти всего - от квантовой механики до информатики, указал на то, что молекулы, обладающие определёнными свойствами, смогут работать как переключатели и заменить собой транзисторы.

В молекулярных компьютерах используется не ДНК, а произвольные молекулярные ансамбли - т.н. «интеллектуальные молекулы». Такие ансамбли могут существовать в двух термодинамически устойчивых состояниях, каждое из которых имеет свои физические и химические свойства. Переводить молекулу из одного состояния в другое (переключать) можно с помощью света, тепла, химических агентов, электрического и магнитного поля и т.д. Фактически такие переключаемые бистабильные молекулы - это наноразмерная двухбитовая система,

воспроизводящая на молекулярном уровне функцию классического транзистора.

Особенно интересны такие превращения бистабильных молекул, после которых сильно меняется электронная конфигурация. Например, после изомеризации в молекуле образуется единая сопряжённая электронная система, следовательно, появляется способность проводить электрический ток. Могут меняться и другие свойства: спектры поглощения сдвигаться в видимую область, возникать нелинейные оптические свойства и, что особенно ценно, флуоресценция.

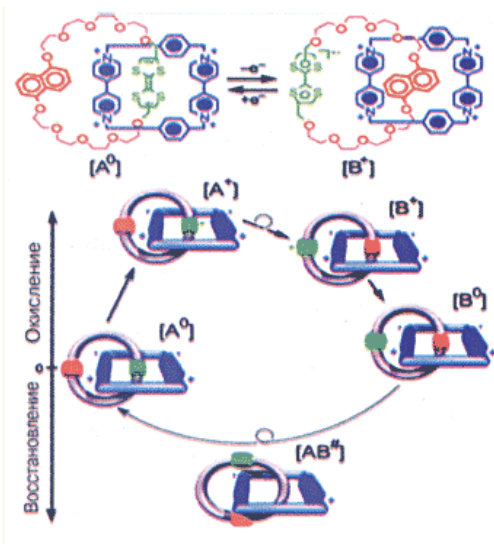
Теоретически использование молекул позволит понизить техпроцесс до 1-5 нм и улучшить время отклика до 10^{-15} с (эквивалентно частоте процессора 10^6 ГГц).

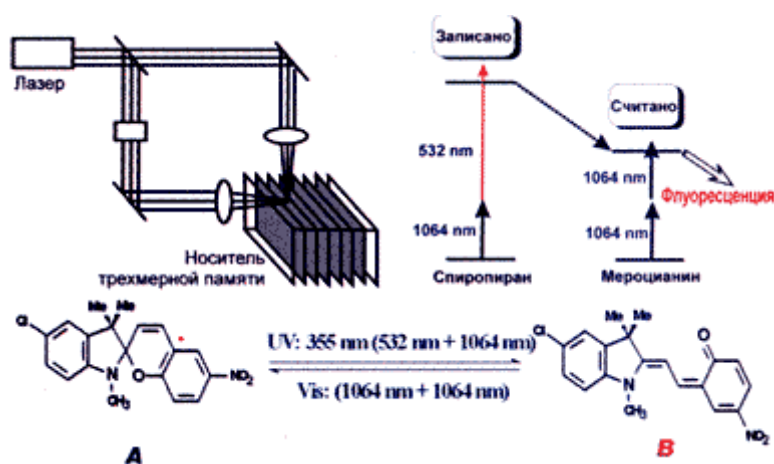
Пока, однако, все эти прекрасные вещи натываются на одну проблему - полное отсутствие материальной базы. Несколько лет назад Д.Ф. Стоддарт и Д. Хис в сотрудничестве с фирмой HP описали переключатель на топологических изомерах супрамолекул катенана, но дальше описания дело пока не пошло.

Также предлагалась трехмерная молекулярная память на бистабильных молекулярных структурах и их превращениях. Конечно, для различных типов памяти потребуются различные характеристики этих превращений, а чтобы обеспечить долгое хранение записанной информации, будут нужны системы с большим временем жизни изомеров.

Впервые принципиальную возможность такой схемы показал П. Рентцпис (Калифорнийский университет) в конце 80-х годов XX века. Он использовал для этого, в частности, фотохромную спиропирановую систему. Поглотив два фотона, молекула А перегруппируется в окрашенную мероцианиновую форму В. Считывание записанной таким образом информации происходит при регистрации флуоресценции молекулы В, также возбуждаемой двухквантовым переходом. Флуоресценция - не единственный, но в силу особенно высокой чувствительности наиболее привлекательный метод считывания записанной информации.

К числу лучших фотохромных систем принадлежит фульгиды индольного ряда. Впервые их получили в СССР, но в настоящее время активно изучают и в США и Европе. В 2000 г. американская компания ConstellationSD, начинавшая свою деятельность в России, объявила о создании первого трёхмерного флуоресцентного диска FMD-ROM, материалом для которого служат 2-индолилфульгиды и 3-индолилфульгиды, вмещающего на десяти слоях до 140 гигабайт. Однако, с появлением Blu Ray, данная технология стала неактуальной, т.к. BD-диск вмещает в максимальном варианте не меньше информации, а технически проще в десятки раз и стоит на порядок дешевле.





Интенсивные исследования по созданию органической трёхмерной памяти вели в Японии под руководством М. Ирие на молекулах диарилэтена, но результат был примерно такой же.

М. Рид (Йельский университет) и Д. Тур (компания HP) сделали сэндвич примерно из 1000 молекул ароматического дитиола и поместили его между золотыми электродами. При определённом напряжении, поданном на электроды, этот сэндвич удерживает электроны (то есть хранит данное состояние в памяти) в течение примерно 10 минут. Что касается проводников, то планировали использовать нанотрубки, однако останавливает сложность и чудовищная цена их получения.

В результате исследования в области молекулярных ЭВМ сопряжены со следующими основными проблемами:

- Чрезвычайная дороговизна изготовления нужных веществ (например, фуллеренов и нанотрубок), делающая попытки получить их в количестве большем, чем для одной опытной машины, почти нереальными.
- Изготовление, сводится практически к ручному манипулированию молекулами или цепочкам сверхсложных химических реакций/физических процессов.
- Большая сложность в изготовлении стабильных экзотических структур – мономолекулярных плёнок, атомарных транзисторов и прочего.
- Низкая стабильность прототипов – малейшее загрязнение примесями, повышение температуры на доли градуса и прочие помехи уничтожают плоды месячных трудов. Таким образом, прототипы не выходят из стерильных лабораторий и вообще могут существовать лишь в очень ограниченном наборе условий.

Из этого можно сделать вывод, что квантовый компьютер, который к тому же сулит большие перспективы, мы, пожалуй, увидим раньше, чем чисто молекулярный. Как известно – дьявол в деталях, и если создание КВМ – одна большая теоретическая проблема, раскусив которую мы получим все и сразу, то создание МВМ – тысячи мелких и очень дорогих проблем, решать которые в общих чертах несложно, но попытки практического воплощения увязают в трудностях, как в болоте.

2.3. Основы схемотехники

2.3.1. Типы цифровых схем и их разработка

Все используемые микросхемы можно разделить следующим образом:

- *Полупроводниковая микросхема* – все элементы и межэлементные соединения выполнены на одном полупроводниковом кристалле (например, кремния, германия, арсенида галлия, оксида гафния).
- *Плёночная интегральная микросхема* – все элементы и межэлементные соединения выполнены в виде плёнок:
 - толстоплёночная интегральная схема;
 - тонкоплёночная интегральная схема.
- *Гибридная микросхема* (микросборка) – кроме полупроводникового кристалла содержит несколько бескорпусных диодов, транзисторов или других электронных компонентов, помещённых в один корпус.
- *Смешанная микросхема* – кроме полупроводникового кристалла содержит плёночные пассивные элементы, размещённые на поверхности кристалла.

В подавляющем большинстве случаев для разработчика цифровых схем достаточно трех моделей, трех уровней представления о работе цифровых устройств:

1. Логическая модель.
2. Модель с временными задержками.
3. Модель с учетом электрических эффектов (или электрическая модель).

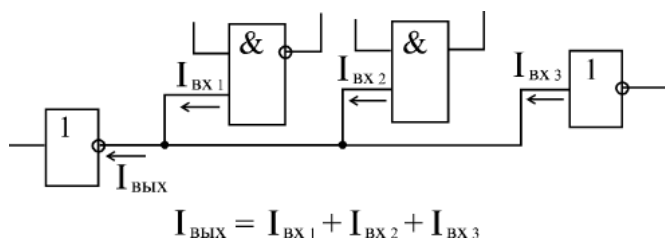
Опыт показывает, что первой простейшей модели достаточно примерно в 20% всех случаев. Она применима для всех цифровых схем, работающих с низкой скоростью, в которых быстродействие не принципиально. Привлечение второй модели, учитывающей задержки срабатывания логических элементов, позволяет охватить около 80% всех возможных схем. Ее применение необходимо для всех быстродействующих устройств и для случая одновременного изменения нескольких входных сигналов. Добавление третьей модели, учитывающей входные и выходные токи, входные и выходные сопротивления и емкости элементов, дает возможность проектирования практически 100% цифровых схем. В первую очередь, третью модель надо применять при объединении нескольких входов и выходов, при передаче сигналов на большие расстояния и при нетрадиционном включении логических элементов (с переводом их в аналоговый или в линейный режимы).

Для иллюстрации работы перечисленных моделей рассмотрим работу самого простейшего логического элемента – инвертора. Инвертор изменяет (инвертирует) логический уровень входного сигнала на противоположный уровень выходного сигнала или, как говорят, изменяет полярность логического сигнала.

На рисунке показано, как будет выглядеть выходной сигнал инвертора при использовании трех его моделей (трех уровней его представления). Эти графики называются временными диаграммами. Из рисунка видно, что в первой логической модели считается, что элемент срабатывает мгновенно, любое изменение уровня входного сигнала сразу же, без задержки приводит к изменению уровня выходного сигнала.



Во второй модели выходной сигнал изменяется с некоторой задержкой относительно входного.



Наконец, в третьей модели выходной сигнал не только задерживается по сравнению с входным, но и его изменение происходит не мгновенно – процесс смены уровней сигнала (или, как говорят, фронт сигнала) имеет конечную длительность. Кроме того, третья модель учитывает изменение уровней логических сигналов.

На практике разработчик, как правило, в начале проектирования пользуется исключительно первой моделью, а затем для некоторых узлов применяет вторую или (реже) еще и третью модель. При этом первая модель не требует вообще никаких цифровых расчетов, для нее достаточно только знание таблиц истинности или алгоритмов функционирования микросхем. Вторая модель предполагает расчет (по сути, суммирование) временных задержек элементов на пути прохождения сигналов. В результате этого расчета может выясниться, что требуется внесение изменений в схему.

Расчеты по третьей модели могут быть различными, в том числе и довольно сложными, но в большинстве случаев они все-таки сводятся лишь к суммированию входных и выходных токов логических элементов. В результате этих расчетов может выясниться, что требуется применение микросхем с более мощными выходами или включение дополнительных элементов.

Проектирование цифровых устройств принципиально отличается от проектирования аналоговых устройств, при котором сложные расчеты неизбежны. Разработчик цифровых устройств имеет дело только с логикой, с логическими сигналами и с алгоритмами работы цифровых микросхем. А что происходит внутри этих микросхем, для него практически не имеет значения.

Справочные данные на цифровые микросхемы обычно содержат большой набор параметров, каждый из которых можно отнести к одному из трех перечисленных уровней представления, к одной из трех моделей.

Например, таблица истинности микросхемы (для простых микросхем) или описание алгоритма ее работы (для более сложных микросхем) относится к первому логическому уровню. Поэтому знать их наизусть каждому разработчику необходимо в любом случае.

Величины задержек логических сигналов между входами и выходами относятся ко второму уровню представления. Типичные величины задержек составляют от единиц наносекунд до десятков наносекунд. Величины задержек для разных микросхем могут быть различными, поэтому в справочниках всегда указывается максимальное значение.

Необходимо также помнить, что задержка при переходе выходного сигнала из единицы в нуль t_{PHL} , как правило, отличается от задержки при переходе выходного сигнала из нуля в единицу t_{PLH} .

Например, для одной и той же микросхемы $t_{PLH} < 1$ нс, а $t_{PHL} < 8$ нс. Здесь английская буква P означает Propagation (распространение), L означает Low (низкий уровень сигнала, нуль), а H — High (высокий уровень сигнала, единица). Количество величин задержек, определяемых справочником для микросхемы, может изменяться от двух до нескольких десятков. Уровни входных и выходных токов, а также уровни входных и выходных напряжений относятся к третьему уровню представления.

Входной ток микросхемы при приходе на вход логического нуля I_{IL} , как правило, отличается от входного тока при приходе на вход логической единицы I_{IH} . Например, $I_{IL} = -0.1$ мА, а $I_{IH} = 20$ мкА (считается, что положительный ток втекает во вход микросхемы, а отрицательный — вытекает из него).

Точно так же выходной ток микросхемы при выдаче логического нуля I_{OL} может отличаться (и обычно отличается) от выходного тока при выдаче логической единицы I_{OH} . Например, для одной и той же микросхемы $I_{OH} < 0.4$ мА, а $I_{OL} < 8$ мА (считается, что положительный ток втекает в выход микросхемы, а отрицательный — вытекает из него). Надо также учитывать, что разные входы и выходы одной и той же микросхемы могут иметь различные входные и выходные токи.

Для выходных напряжений логического нуля U_{OL} и единицы U_{OH} в справочниках обычно задаются предельно допустимые значения при данной величине выходного тока. В этом случае, чем больше выходной ток, тем меньше напряжение логической единицы и тем больше напряжение логического нуля. Например, $U_{OH} > 2.5$ В (при $I_{OH} < 0.4$ мА), а $U_{OL} < 0.5$ В (при $I_{OL} < 8$ мА).

Задаются в справочниках также и допустимые уровни входных напряжений, которые микросхема еще воспринимает как правильные логические уровни нуля и единицы. Например, $U_{IH} > 2$ В, $U_{IL} < 0.8$ В. Как правило, входные напряжения логических сигналов не должны выходить за пределы напряжения питания. В обозначениях напряжений и токов буква I означает Input (вход), буква O означает Output (выход), L — Low (нуль), а H — High (единица).

К третьему уровню представления относятся также величины внутренней емкости входов микросхемы (обычно от единиц до десятков пикофард) и допустимая величина емкости, к которой может подключаться выход микросхемы, то есть емкость нагрузки C_L (порядка 100 пФ). На этом же уровне представления задаются максимально допустимые величины длительности положительного фронта t_{LH} и отрицательного фронта t_{HL} входного сигнала, например, $t_{HL} < 1$ мкс, $t_{LH} < 1$ мкс. То есть при большей длительности перехода входного сигнала из единицы в нуль и из нуля в единицу микросхема может работать неустойчиво, неправильно, нестандартно.

К третьему уровню представления можно отнести также такие параметры, как допустимое напряжение питания микросхемы U_{CC} и максимальный ток, потребляемый микросхемой I_{CC} . При этом потребляемый ток I_{CC} зависит от уровней выходных токов микросхемы. Эти параметры надо учитывать при выборе источника питания для проектируемого устройства, а также в процессе изготовления печатных плат — при выборе ширины токоведущих дорожек.

Наконец, к третьему же уровню относится ряд параметров, которые часто упоминаются в литературе, но не всегда приводятся в справочных таблицах:

- *Порог срабатывания* — уровень входного напряжения, выше которого сигнал воспринимается как единица, а ниже — как нуль. Для наиболее распространенных ТТЛ микросхем он примерно равен 1.3~1.4 В.
- *Помехозащищенность* — характеризует величину входного сигнала помехи, накладывающегося на входной сигнал, который еще не может изменить состояние выходных сигналов. Помехозащищенность определяется разницей между напряжением U_{IH} и порогом срабатывания (это помехозащищенность единичного уровня), а также разницей между порогом срабатывания и U_{IL} (это помехозащищенность нулевого уровня).
- *Коэффициент разветвления* — число входов, которое может быть подключено к данному выходу без нарушения работы. Определяется отношением выходного тока к входному. Стандартная величина коэффициента разветвления при использовании микросхем одного типа (одной серии) равна 10.
- *Нагрузочная способность* — параметр выхода, характеризующий величину выходного тока, которую может выдать в нагрузку данный выход без нарушения работы. Чаще всего нагрузочная способность прямо связана с коэффициентом разветвления.

Таким образом, большинство справочных параметров микросхемы относятся к третьему уровню представления (к модели с учетом электрических эффектов), поэтому в большинстве случаев (до 80%) знать их точные значения наизусть не обязательно. Достаточно помнить примерные типовые значения параметров для данной серии микросхем.

Рассмотрим сначала входы микросхем.

На первом уровне представления (логическая модель) и на втором уровне представления (модель с временными задержками) о входах микросхем вообще ничего знать не нужно. Вход рассматривается как бесконечно большое сопротивление, никак не влияющее на подключенные к нему выходы. Правда, количество входов, подключенных к одному выходу, влияет на задержку распространения сигнала, но, как правило, незначительно, поэтому это влияние учитывается редко.

Даже на третьем уровне представления (электрическая модель) в большинстве случаев не нужно знать о внутреннем строении микросхемы, о схемотехнике входов. Достаточно считать, что при подаче на вход сигнала логического нуля из этого входа вытекает ток, не превышающий I_{IL} , а при подаче сигнала логической единицы в этот вход втекает ток, не превышающий I_{IH} .

Для правильной логики работы микросхемы достаточно, чтобы уровень напряжения входного сигнала логического нуля был меньше U_{IL} , а уровень напряжения входного сигнала логической единицы был больше U_{IH} .

Особым случаем является ситуация, когда какой-нибудь вход не подключен ни к одному из выходов — ни к общему проводу, ни к шине питания (так называемый висящий вход). Иногда возможности микросхемы используются не полностью и на некоторые входы не подаются сигналы. Однако при этом микросхема может не работать или работать нестабильно, так как ее правильное включение подразумевает наличие на всех входах логических уровней, пусть даже и неизменных. Поэтому рекомендуется подключать неиспользуемые входы к напряжению питания микросхемы U_{CC} или к общему проводу (к земле), в зависимости от того, какой логический уровень необходим на этом входе. Но для некоторых серий микросхем, выполненных по технологии ТТЛ (например, К155 или КР531), неиспользуемые входы надо подключать к напряжению питания не напрямую, а только через резистор величиной около 1 кОм (достаточно одного резистора на 20 входов).

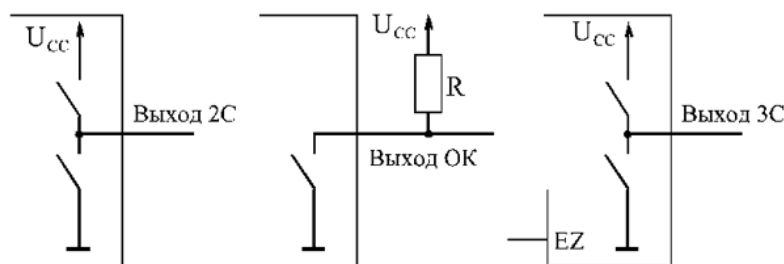
На неподключенных входах микросхем ТТЛ формируется напряжение около 1.5~1.6 В, которое иногда называют висячим потенциалом. Обычно этот уровень воспринимается микросхемой как сигнал логической единицы, но рассчитывать на это не стоит. Потенциал, образующийся на неподключенных входах микросхем КМОП, может восприниматься микросхемой и как логический нуль, и как логическая единица. В любом случае все входы надо куда-то подключать. Неподключенными допускается оставлять только те входы (ТТЛ, а не КМОП), состояние которых в данном включении микросхемы не имеет значения.

Выходы микросхем принципиально отличаются от входов тем, что учет их особенностей необходим даже на первом и втором уровнях представления.

Существуют три разновидности выходных каскадов, существенно различающиеся как по своим характеристикам, так и по областям применения:

- стандартный выход или выход с двумя состояниями (обозначается 2С, 2S или, реже, ТТЛ, TTL);
- выход с открытым коллектором (обозначается ОК, OC);
- выход с тремя состояниями или (что то же самое) с возможностью отключения (обозначается 3С, 3S).

Стандартный выход 2С имеет всего два состояния: логический нуль и логическая единица, причем оба они активны, то есть выходные токи в обоих этих состояниях (I_{OL} и I_{OH}) могут достигать заметных величин. На первом и втором уровнях представления такой выход можно считать состоящим из двух выключателей, которые замыкаются по очереди, причем замкнутому верхнему выключателю соответствует логическая единица на выходе, а замкнутому нижнему — логический нуль.



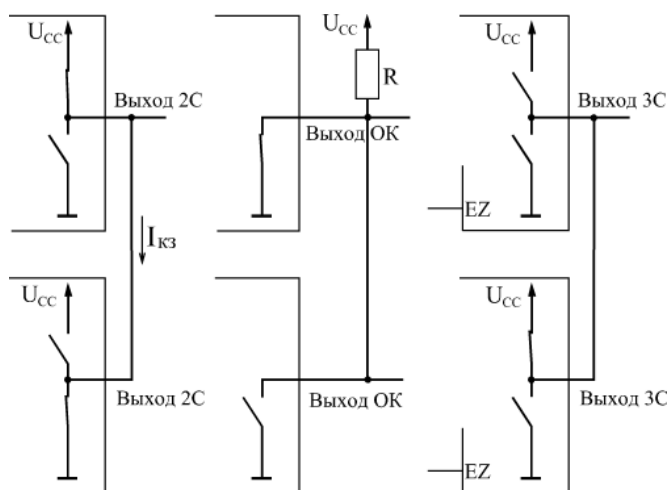
Три типа выходов цифровых микросхем

Выход с открытым коллектором ОК тоже имеет два возможных состояния, но только одно из них (состояние логического нуля) активно, то есть обеспечивает большой втекающий ток I_{OL} . Второе состояние сводится, по сути, к тому, что выход полностью отключается от присоединенных к нему входов. Это состоя-

ние может использоваться в качестве логической единицы, но для этого между выходом ОК и напряжением питания необходимо подключить нагрузочный резистор R (так называемый pull-up) величиной порядка сотен Ом. На первом и втором уровнях представления такой выход можно считать состоящим из одного выключателя, замкнутому состоянию которого соответствует сигнал логического нуля, а разомкнутому — отключенное, пассивное состояние. Правда, от величины резистора R зависит время переключения выхода из нуля в единицу, что влияет на задержку $t_{\text{ИЗ}}$, но при обычно используемых номиналах резисторов это не слишком важно.

Наконец, выход с тремя состояниями 3С очень похож на стандартный выход, но к двум состояниям добавляется еще и третье — пассивное, в котором выход можно считать отключенным от последующей схемы. На первом и втором уровнях представления такой выход можно считать состоящим из двух переключателей, которые могут замыкаться по очереди, давая логический нуль и логическую единицу, но могут и размыкаться одновременно. Это третье состояние называется также высокоимпедансным или Z-состоянием. Для перевода выхода в третье Z-состояние используется специальный управляющий вход, обозначаемый OE (Output Enable — разрешение выхода) или EZ (Enable Z-state).

Почему же помимо стандартного выхода (2С) были предложены еще два типа выходов (ОК и 3С)? Дело в том, что выходы, имеющие помимо активных еще и пассивное состояние, очень удобны для объединения их между собой. Например, если на один и тот же вход надо по очереди подавать сигналы с двух выходов, то выходы 2С для этого не подходят, а вот выходы ОК и 3С — подходят.



Объединение выходов цифровых микросхем

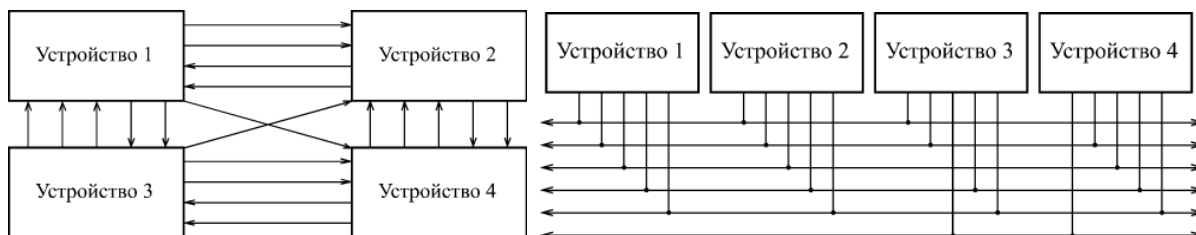
При объединении двух или более выходов 2С вполне возможна ситуация, при которой один выход стремится выдать сигнал логической единицы, а другой — сигнал логического нуля. Легко заметить, что в этом случае через верхний замкнутый ключ выхода, выдающего единицу, и через нижний замкнутый ключ выхода, выдающего нуль, пойдет недопустимо большой ток короткого замыкания. Это аварийная ситуация, при которой уровень получаемого выходного логического сигнала точно не определен — он может восприниматься последующим входом и как нуль, и как единица. Конфликтующие выходы могут даже выйти из строя, нарушив работу микросхем и схемы в целом.

Зато в случае объединения двух выходов ОК такого конфликта в принципе произойти не может. Даже если ключ одного выхода замкнут,

а другого — разомкнут, аварийной ситуации не произойдет, так как недопустимо большого тока не будет, а на объединенном выходе будет сигнал логического нуля. А при объединении двух выходов 3С аварийная ситуация хоть и возможна (если оба выхода одновременно находятся в активном состоянии), но ее легко можно предотвратить, если организовать схему так, что в активном состоянии всегда будет находиться только один из объединенных выходов 3С.

Объединение выходов цифровых микросхем совершенно необходимо также при шинной (магистральной) организации связей между цифровыми устройствами. Шинная организация связей применяется, например, в компьютерах и в других микропроцессорных системах. Суть ее сводится к следующему.

При классической организации связей все сигналы между устройствами передаются по своим отдельным линиям (проводам). Каждое устройство передает свои сигналы всем другим независимо от других устройств. В этом случае обычно получается очень много линий связи, к тому же правила обмена сигналами по этим линиям (или протоколы обмена) чрезвычайно разнообразны.

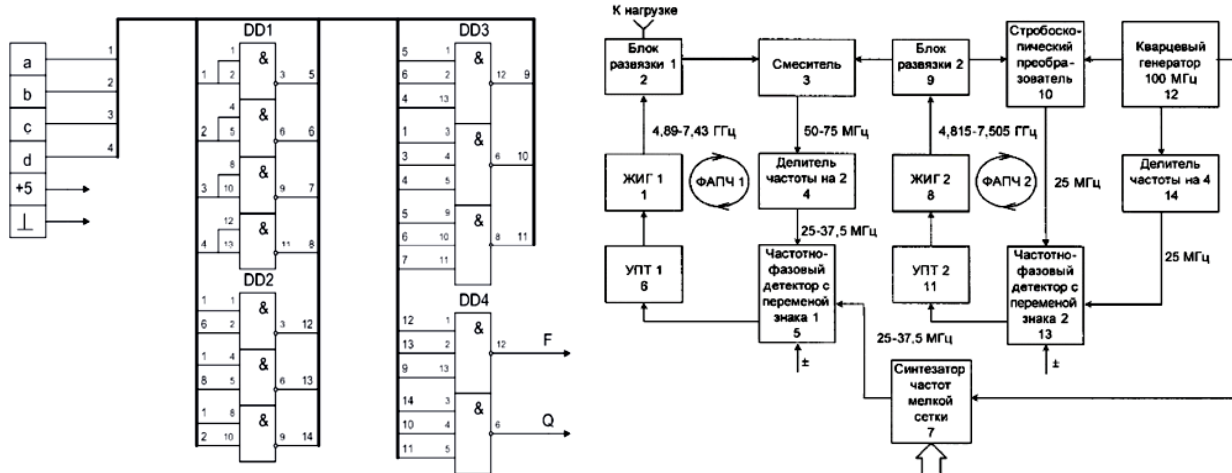


Классическая и шинная организация связей

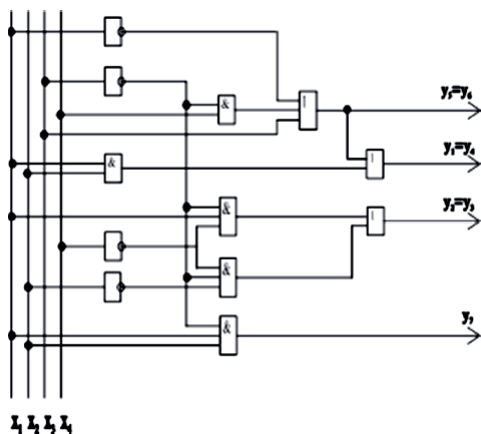
На третьем уровне представления (электрическая модель) необходимо уже учитывать, что выходные ключи представляют собой не простые тумблеры (как на первых двух уровнях представления), а транзисторные ключи со своими специфическими параметрами. Однако в большинстве случаев достаточно знать, какой ток может выдать данный выход при логическом нуле и при логической единице. Величины этих токов не должны превышать суммы токов всех входов, подключенных к данному выходу. Количеством входов, которое можно подключить к одному выходу, определяется коэффициент разветвления или нагрузочная способность микросхемы. Существуют микросхемы с обычной нагрузочной способностью и с повышенной (больше обычной в два раза и более). Выходы 3С, как правило, имеют повышенную нагрузочную способность (то есть обеспечивают большие выходные токи). Выходы 2С и ОК могут быть как с обычной, так и с повышенной нагрузочной способностью.

Только в сложных случаях, например, при переводе логического элемента в линейный режим за счет обратных связей, нужен учет других параметров входных и выходных каскадов. Но в этих редких случаях гораздо проще и надежнее не считать ничего самому, а воспользоваться стандартными схемами включения микросхем или подобрать режимы работы и номиналы внешних элементов (резисторов, конденсаторов) непосредственно на макете проектируемого устройства. В отличие от расчетов, такой подход даст полную гарантию работоспособности выбранного решения.

Принципиальная схема — наиболее подробная. Она обязательно показывает все использованные в устройстве элементы и все связи между ними. Если схема строится на основе микросхем, то должны быть показаны номера выводов всех входов и выходов этих микросхем. Принципиальная схема должна позволять полностью воспроизвести устройство. Обозначения принципиальной схемы наиболее жестко стандартизованы, отклонения от стандартов не рекомендуются.



Функциональная схема представляет собой гибрид структурной и принципиальной. Некоторые наиболее простые блоки, узлы, части устройства отображаются на ней, как на структурной схеме, а остальные –



как на принципиальной схеме. Функциональная схема дает возможность понять всю логику работы устройства, все его отличия от других подобных устройств, но не позволяет без дополнительной самостоятельной работы воспроизвести это устройство. Что касается обозначений, используемых на функциональных схемах, то в части, показанной как структура, они не стандартизованы, а в части, показанной как принципиальная схема, — стандартизованы.

В технической документации обязательно приводятся структурная или функциональная схема, а также обязательно принципиальная схема. В научных статьях и книгах чаще всего ограничиваются структурной или функциональной схемой, приводя принципиальные схемы только некоторых узлов.

2.3.2. Типы логики

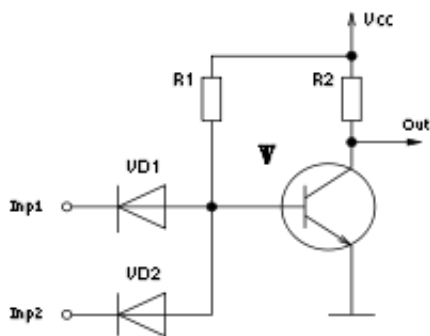
Логические элементы подразделяются по типу использованных в них электронных элементов. Наибольшее применение в настоящее время находят следующие логические элементы:

- Микросхемы на биполярных транзисторах:
 - РТЛ — резисторно-транзисторная логика (устаревшая, заменена на ТТЛ);
 - ДТЛ — диодно-транзисторная логика (устаревшая, заменена на ТТЛ);
 - ТТЛ — транзисторно-транзисторная логика — микросхемы сделаны из биполярных транзисторов с многоэмиттерными транзисторами на входе;
 - ТТЛШ — транзисторно-транзисторная логика с диодами Шоттки — усовершенствованная ТТЛ, в которой используются биполярные транзисторы с эффектом Шоттки;
 - ЭСЛ — эмиттерно-связанная логика — на биполярных транзисторах, режим работы которых подобран так, чтобы они не входили в режим насыщения, — что существенно повышает быстродействие;
 - ИИЛ — интегрально-инжекционная логика.
- Микросхемы на униполярных (полевых) транзисторах (самые экономичные по потреблению тока):
 - МОП-логика (металл-оксид-полупроводник, MOS) — микросхемы формируются из полевых транзисторов n-МОП или p-МОП типа;
 - КМОП-логика (комплементарная МОП-логика, CMOS) — каждый логический элемент микросхемы состоит из пары полевых транзисторов (n-МОП и p-МОП).
 - Смешанная технология BiCMOS.

КМОП и ТТЛШ технологии являются наиболее распространёнными логиками микросхем. Микросхемы, изготовленные по ЭСЛ-технологии, являются самыми быстрыми, но и наиболее энергопотребляющими и применялись при производстве вычислительной техники в тех случаях, когда важнейшим параметром была скорость вычисления. В СССР самые производительные ЭВМ типа ЕС106х изготавливались на ЭСЛ-микросхемах. Сейчас эта технология используется редко.

Микросхемы на биполярных транзисторах:

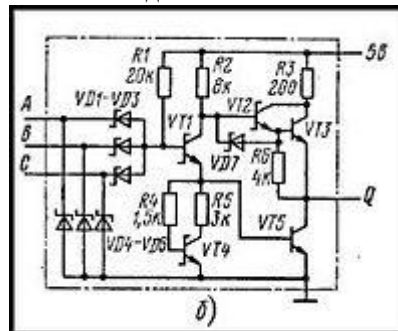
РТЛ (резисторно-транзисторная логика, RTL)	Самый простой вид логического вентиля. На рисунке — элемент 2ИЛИ-НЕ. Разновидностью является резисторно-емкостная транзисторная логика (РЕТЛ) аналогична РТЛ, но на входах в ней используются конденсаторы, пропускающие только импульсы. Достоинства: конструктивная простота, низкая стоимость. Недостатки: высокая рассеиваемая мощность (как на включенном ключе, так и на резисторах), нечёткий уровень сигналов (уровень единицы от : 0.9В до напряжения питания), крайне низкое быстродействие, низкая помехоустойчивость, сложность разработки и малая интеграция, низкая нагрузочная способность выходов (обычно не более трёх входов других элементов). После перехода на интегральные микросхемы РТЛ-логика практически исчезла и применяется только в специальных целях.
---	--



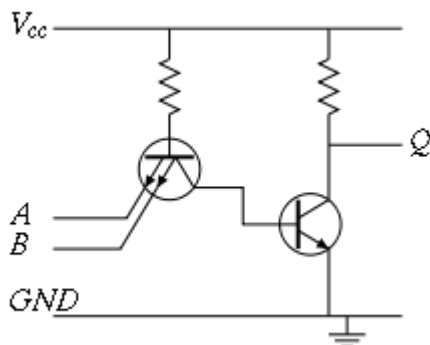
ДТЛ (диодно-транзисторная логика, DTL)

На рисунке – упрощённая схема двухвходового ДТЛ-элемента 2И-НЕ, технология построения цифровых схем на основе биполярных транзисторов, диодов и резисторов. Своё название технология получила благодаря реализации логических функций (например, 2И) с помощью диодных цепей, а усиления и инверсии сигнала — с помощью транзистора.

Основное преимущество ДТЛ перед более ранней технологией РТЛ — возможность создания большого числа входов.



Логический элемент 3И-НЕ в серии микросхем 74LS(K555). Логические элементы на основе ДТЛ являлись основой для многих ЭВМ второго поколения, например БЭСМ-6, IBM 1401.

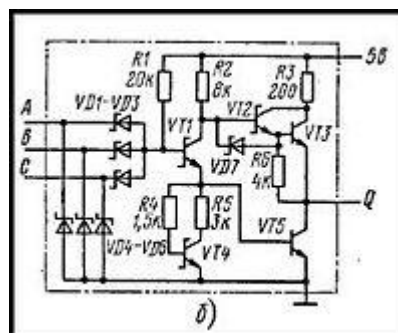


ТТЛ (транзисторно-транзисторная логика, TTL)

Название *транзисторно-транзисторный* возникло из-за того, что транзисторы используются как для выполнения логических функций (например, И, ИЛИ), так и для усиления выходного сигнала.

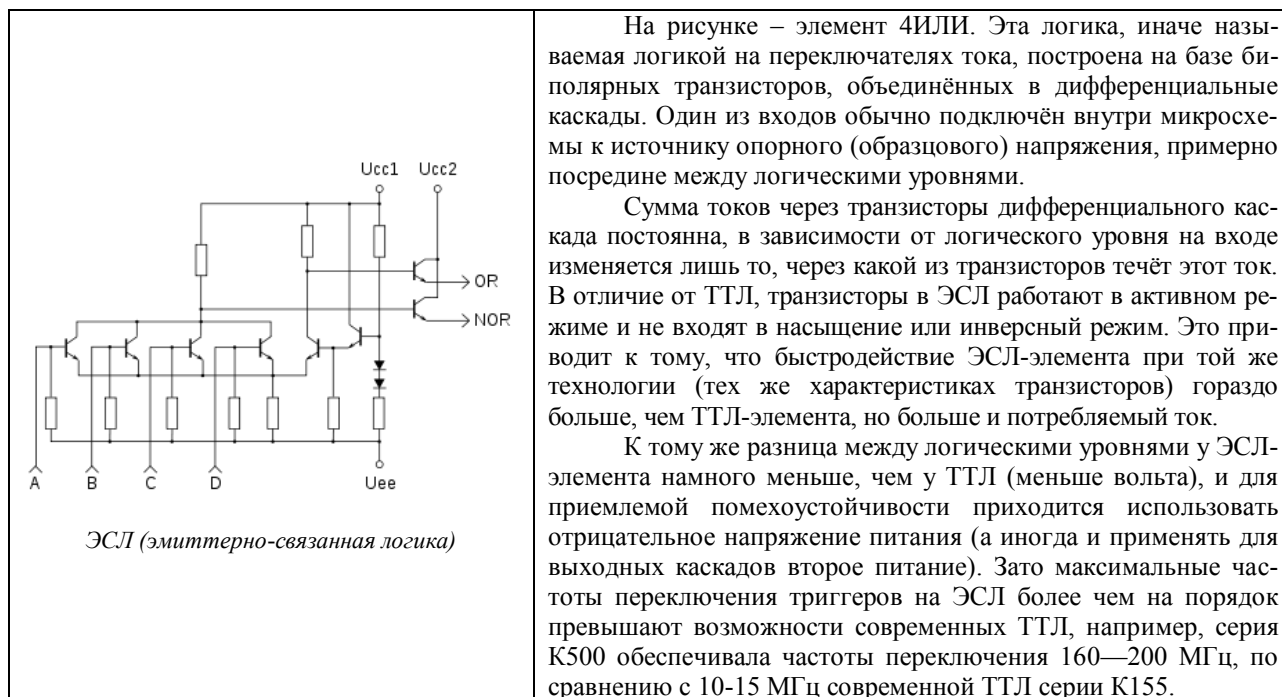
Простейший базовый элемент ТТЛ выполняет логическую операцию И-НЕ, в принципе повторяет структуру ДТЛ микросхем и в то же время за счёт использования многоэмиттерного транзистора объединяет свойства диода и транзисторного усилителя, что позволяет увеличить быстродействие, снизить потребляемую мощность и усовершенствовать технологию изготовления микросхемы.

ТТЛ стала популярной среди разработчиков электронных систем после того, как в 1965 году фирма Texas Instruments представила серию интегральных микросхем 7400, которая стала промышленным стандартом, но ТТЛ-микросхемы производятся и другими компаниями. Поскольку биполярные ИМС серии TI74 наиболее распространены, их повторяет продукция других фирм (AMD, Fairchild Semiconductor, Harris, Intel, Motorola и т.д.).

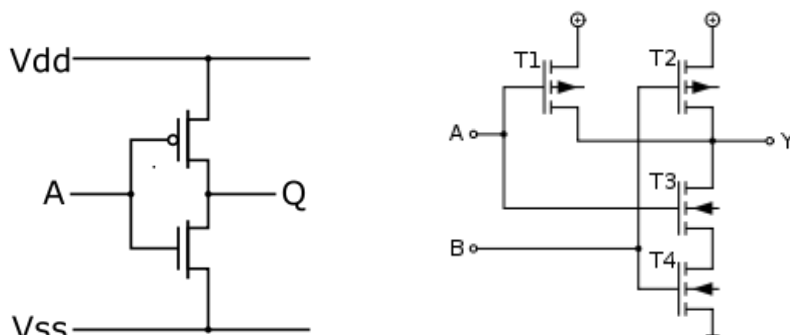


Логический элемент 3И-НЕ в серии микросхем 74LS(K555).

Разновидность ТТЛ – ТТЛШ использует во входных цепях диоды Шоттки, в которых барьер Шоттки не позволяет транзистору войти в режим насыщения. В результате задержки переключения малы, а быстродействие высокое.



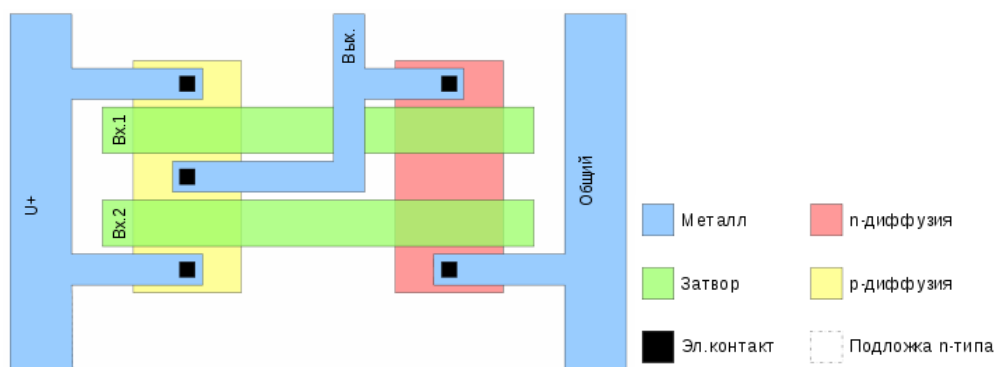
Наиболее современными в настоящее время являются **микросхемы на полевых транзисторах.**



Статический КМОП инвертор и 2И-НЕ

В технологии КМОП используются полевые транзисторы с изолированным затвором с каналами разной проводимости. Отличительной особенностью схем КМОП по сравнению с биполярными технологиями (ТТЛ, ЭСЛ и др.) является очень малое энергопотребление в статическом режиме (в большинстве случаев можно считать, что энергия потребляется только во время переключения состояний). Отличительной особенностью структуры КМОП по сравнению с другими МОП-структурами (N-МОП, P-МОП) является наличие как n-, так и p-канальных полевых транзисторов; как следствие, КМОП-схемы обладают более высоким быстродействием и меньшим энергопотреблением, однако при этом характеризуются более сложным технологическим процессом изготовления и меньшей плотностью упаковки.

Подавляющее большинство современных логических микросхем, в том числе, процессоров, используют схемотехнику КМОП.



2И-НЕ вентиль на кристалле в процессоре

Долгое время КМОП рассматривалась как энергосберегающая, но медленная альтернатива ТТЛ, поэтому микросхемы КМОП нашли применение в электронных часах, калькуляторах и других устройствах с батарейным питанием, где энергопотребление было критичным.

К 1990 году с повышением степени интеграции микросхем встала проблема рассеивания энергии на элементах. В результате технология КМОП оказалась в выигрышном положении. Со временем были достигнуты скорость переключения и плотность монтажа, недостижимые в технологиях, основанных на биполярных транзисторах.

Ранние КМОП-схемы были очень уязвимы к электростатическим разрядам. Сейчас эта проблема в основном решена, но при монтаже КМОП-микросхем рекомендуется принимать меры по снятию электрических зарядов.

III. ИНФОРМАТИКА

3.1. Типы данных и их представление в машине

3.1.1. Типизация

Как правило, типы данных в описании архитектуры не всегда строго соответствуют подобным типам в математике. Например, тип «целое число» большинства языков программирования не соответствует принятому в математике типу «целое число», так как в математике указанный тип не имеет ограничений ни сверху, ни снизу, а в языках программирования эти ограничения есть. В языках программирования имеется множество целых типов, отличающихся допустимым диапазоном значений (определяемым объемом занимаемой памяти). Стоит отметить, что в большинстве реализаций языков и систем выход за границу целого типа (переполнение) не приводит к исключительной ситуации.

По сути, система типов данных имеет два значения – одно, с точки зрения теории программирования, это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений. Второе определение важно для архитектуры – это метод хранения и обработки данных в процессоре.

В обоих случаях тип данных определяет:

- допустимое множество значений,
- допустимый на нем набор операций,
- способ реализации хранения значений и выполнения операций.

Теоретически не могут существовать языки и архитектуры, в которых отсутствуют типы (включая полиморфные). Это следует из того, что все ЭВМ основаны на машине Тьюринга или на лямбда-исчислении. И в том, и в другом случае необходимо оперировать как минимум одним типом данных — хранящимся на ленте (машина Тьюринга) или передаваемым и возвращаемым из функции (лямбда-исчисление). Естественно, вопрос – наличие всего лишь одного стандартного неизменного типа – это типизация или нет, чисто философский.

Бестиповая типизация

С точки зрения компьютера (и соответствующих ЯП), типов все-таки нет — все сущности считаются просто последовательностями бит различной длины. Бестиповость, типизация обычно присуща низкоуровневым (язык ассемблера, Forth) и эзотерическим (Brainfuck, HQ9, Piet) языкам. Однако и у нее, наряду с недостатками, есть некоторые преимущества.

Преимущества:

- Позволяет писать на предельно низком уровне, причем компилятор / интерпретатор не будет мешать какими-либо проверками типов. Вы вольны производить любые операции над любыми видами данных.
- Получаемый код обычно более эффективен.
- Прозрачность инструкций. При знании языка обычно нет сомнений, что из себя представляет тот или иной код.

Недостатки:

- Сложность. Часто возникает необходимость в представлении комплексных значений, таких как списки, строки или структуры;
- Отсутствие проверок. Любые бессмысленные действия, например вычитание указателя на массив из символа, будут считаться совершенно нормальными, что чревато трудноуловимыми ошибками;
- Низкий уровень абстракции. Работа с любым сложным типом данных ничем не отличается от работы с числами, что конечно будет создавать много трудностей.

Сильная бестиповая типизация

Да, такое существует. Например, в языке ассемблера нельзя ассемблировать программу, если вы попытаетесь загрузить в регистр `cx` (16 бит) данные из регистра `rax` (64 бита):

```
mov cx, eax;
```

В результате получается, что процессор, конечно, не знает типов в традиционном смысле, но различает вид данных по тому, в каком регистре они находятся. Даже в ассемблере, например, различаются типы «целое число» и «вещественное число». Это объясняется тем, что для чисел рассматриваемых типов отводятся различные объёмы памяти, используются различные регистры микропроцессора, а для операций с ними применяются различные команды и различные ядра процессора.

Концепция типа данных появилась в языках программирования высокого уровня как естественное отражение того факта, что обрабатываемые программой данные могут иметь различные множества допустимых значений, храниться в памяти компьютера различным образом, занимать различные объёмы памяти и обрабатываться с помощью различных команд процессора.

Каждый язык программирования поддерживает один или несколько встроенных типов данных (базовых типов), кроме того, развитые языки программирования предоставляют программисту возможность описывать собственные типы данных, комбинируя или расширяя существующие.

Итак, с какими типами данных и каким образом работает процессор?

Прежде всего, это числовой тип, который хранит, конечно же, числа. К нему могут применяться все обычные арифметические операции. Числа бывают разные и простейший их тип – целые. Обычно их классифицируют по наличию знака.

Беззнаковые целые представляют только неотрицательные числа, при этом все разряды кода используются для представления значения числа, и максимальное число соответствует единичным значениям кода во всех разрядах: $111\dots111$. m -байтовая переменная целого типа без знака, очевидно, принимает значения от 0 до $2^{8m}-1$. Беззнаковые целые используются для адресации памяти или представления символов, работа с ними предельно проста. В некоторых языках, например Java, беззнаковые целые типы (за исключением символического) отсутствуют.

Целочисленные со знаком могут принимать как положительные, так и отрицательные значения. Существует несколько различных способов представления целых значений в двоичном коде в виде величины со знаком:

Обратный код: Двоичное представление числа 5 есть 101. Обратный 8-разрядный двоичный код числа +5 записывается как 00000101. Обратный 8-разрядный двоичный код числа –5 есть 11111010.

Прямой код: При записи числа в прямом коде старший разряд является знаковым разрядом. Если его значение равно 0, то число положительное, если 1 — то отрицательное. В остальных разрядах (которые называются цифровыми разрядами) записывается двоичное представление модуля числа.

Дополнительный код: При записи числа в дополнительном коде старший разряд является знаковым. Если его значение равно 0, то в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом. Если число, записанное в прямом коде, отрицательное, то все разряды числа инвертируются, а к результату прибавляется 1. К получившемуся числу дописывается старший (знаковый) разряд, равный 1.

Также целочисленные подразделяют по длине на короткие (`short`, обычно 1/2 стандартного размера), длинные (`long`, обычно 2 стандартных размера) и целые стандартной длины (`int`), совпадающие с размером машинного слова на целевой платформе. В некоторых языках может также использоваться тип данных «двойное длинное» (`long long` или `quad`), представляющий собой учетверенное слово.

Кроме целых, процессор активно использует вещественную арифметику и простейший вид вещественного числа – с фиксированной запятой. Пример арифметики с фиксированной запятой – перевод рублей в копейки. В таком случае, чтобы запомнить сумму 12 рублей 34 копейки, мы записываем в ячейку памяти число 1234.

Такую арифметику описывают, указав f бит для хранения дробной части и $i = n - f$ целой части, обозначают обычно просто i, f . Например: арифметика 8,24 отводит на целую часть 8 битов и 24 — на дробную. Соответственно, она способна хранить числа от -128 до $128 - z$ с ценой (весом) младшего разряда $z = 2^{-24}$.

Применение таких чисел оправдывает себя во многих случаях, например:

- Для ускорения вычислений в местах, где не требуется высокая точность. В большинстве современных процессоров ФЗ аппаратно не реализована, но даже программная ФЗ очень быстра — поэтому она применяется в игровых движках, растеризаторах и т. д.

- Чтобы обеспечить минимальную поддержку дробных чисел на целочисленном процессоре — микроконтроллера, мобильного телефона, приставок, вплоть до PlayStation и т. д. Если не решаются некорректные задачи и СЛАУ высокого порядка, фиксированной запятой зачастую достаточно — важно только подобрать подходящий вес младшего разряда для каждой из величин.

- Для записи чисел, которые по своей природе имеют постоянную абсолютную погрешность: координаты в программах вёрстки, денежные суммы. Например, файлы метрики шрифтов TeX используют 32-битный знаковый тип с фиксированной запятой (12,20).

• Кроме того, фиксированная запятая ведёт себя абсолютно предсказуемо – при подсчёте денег это позволяет наладить разные виды округления, а в играх – наиболее простой способ реализовать запись повторов.

Недостаток фиксированной запятой – узкий диапазон чисел с угрозой переполнения на одном конце диапазона и потерей точности вычислений на другом. Эта проблема и привела к изобретению плавающей запятой.

Чис- вающей стоит из	Точность	Одинарная	Двойная	Расширенная (четверная)	ло с пла- вающей со- трех частей:
	Размер (байты)	4	8	10	
	Число десятичных знаков	7	15	19	
	Наименьшее значение, денормализованное	$1.4 \cdot 10^{-45}$	$5 \cdot 10^{-324}$	$1.9 \cdot 10^{-4951}$	
	Наименьшее значение, нормализованное	$1.4 \cdot 10^{-38}$	$2.3 \cdot 10^{-308}$	$3.4 \cdot 10^{-4932}$	
	Наибольшее значение	$3.4 \cdot 10^{38}$	$1.7 \cdot 10^{308}$	$1.1 \cdot 10^{4932}$	
	Поля	S-E-F	S-E-F	S-E-I-F	
	Размеры полей	1-8-32	1-11-52	1-15-1-63	

- мантиссы (выражающей значение числа без учёта порядка);
- знака мантиссы (указывающего на отрицательность или положительность числа);
- порядка (выражающего степень основания числа, на которое умножается мантисса).

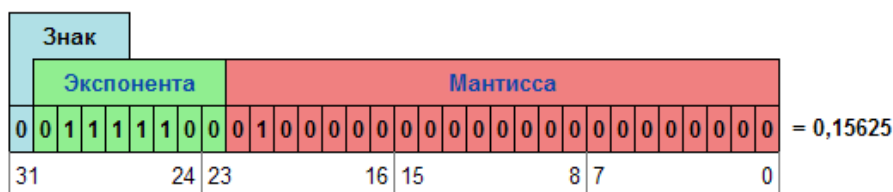
Вот диапазон представления таких чисел:

Здесь S – знак, E – показатель степени, I – целая часть, F – дробная часть.

Структура числа половинной точности:



Структура числа одинарной точности:



Пример расширенного числа:

$$0x7FFE\ FFFF\ FFFF\ FFFF\ FFFF\ FFFF\ FFFF\ FFFF \\ \approx 1.189731495357 \cdot 10^{4932}$$

Иногда, например, в криптографии или математических исследованиях, возникает потребность в работе с числами, превышающими любые диапазоны длины стандартных вещественных переменных. В результате, даже вещественные регистры математического сопроцессора не в состоянии поместить такое число целиком. Этой проблемой занимается т.н. длинная арифметика — операции над числами, разрядность которых превышает длину машинного слова данной платформы. Эти операции реализуются не аппаратно, а программно, используя базовые аппаратные средства работы с числами меньших порядков. Частный случай — арифметика произвольной точности — относится к арифметике, в которой длина чисел ограничена только объёмом доступной памяти. Примером языка с встроенной поддержкой таких типов является UBASIC, применяемый в криптографии.

Такая арифметика применяется в нескольких случаях:

- Компьютеры низкой разрядности, микроконтроллеры. Например, микроконтроллеры серии AVR имеют 8-битный регистр и 10-битный АЦП – так что при обработке информации без длинной арифметики не обойтись.
- Криптография. Большинство систем шифрования данных, а также систем цифровой подписи данных используют целочисленную арифметику по модулю m , где m – очень большое положительное натуральное число.

рально чисто, не обязательно простое. Например, алгоритмы шифрования RSA, Rabin или метод Эль Гамала требуют наличия эффективных методов для операций перемножения и возведения в степень чисел, имеющих порядки, превышающие 10^{309} .

- Математическое и финансовое ПО, требующее, чтобы результат вычисления на компьютере совпал до последнего разряда с результатом вычисления на бумаге. В частности, калькулятор Windows (начиная с Windows 95) поддерживает длинную арифметику. Калькулятор Windows Vista и далее проводит четыре арифметических действия с намного большей точностью, чем аппаратно позволяет процессор x86. Для обычных научных и инженерных расчётов длинная арифметика применяется редко: ошибки во входных данных обычно намного больше, чем ошибки округления.

- Спортивное и олимпиадное программирование. С распространением стандартных библиотек для работы с длинными числами популярность темы постепенно сходит на нет.

Аппаратно для реализации арифметики произвольной точности от процессора требуется лишь поддержка косвенной адресации; в арифметике же фиксированной точности можно обойтись даже без неё. Тем не менее, определённые функции ускоряют длинную арифметику, одновременно упрощая её программирование:

- Флаг переноса. Операции «сложить/вычесть с переносом», «циклический сдвиг через бит переноса».
- Автоинкрементные и автодекрементные операции доступа к памяти.
- Умножение $w \cdot w \rightarrow 2w$, деление $2w/w \rightarrow w$.

В большинстве языков высокого уровня существует арифметика длиной в два слова. Более длинную арифметику обычно приходится писать своими силами, по мере возможности оптимизируя на ассемблере — в языках высокого уровня таких абстракций, как «регистровая пара» и «бит переноса» обычно нет. В Turbo Pascal существовал шестибайтовый эмулированный дробный тип — Real (в Delphi переименованный в Real48). Вычисления с ним также проводились с помощью длинной арифметики.

Для такой арифметики существует огромное количество специальных алгоритмов, ускоряющих стандартные операции, например, для умножения, вместо стандартного способа используется умножение Карацубы, метод Toom 3-Way и т.д.

Существовали платформы, поддерживающие по умолчанию и более сложные типы, например такие векторы, как Cray. Векторные процессоры были распространены в сфере научных вычислений, где они являлись основой большинства суперкомпьютеров с 1980-х до 1990-х, но резкое увеличение производительности традиционных архитектур привело к вытеснению векторных машин.

В большинстве современных микропроцессоров имеются векторные расширения (SSE). Кроме того, современные видеокарты и физические ускорители можно рассматривать как векторные сопроцессоры.

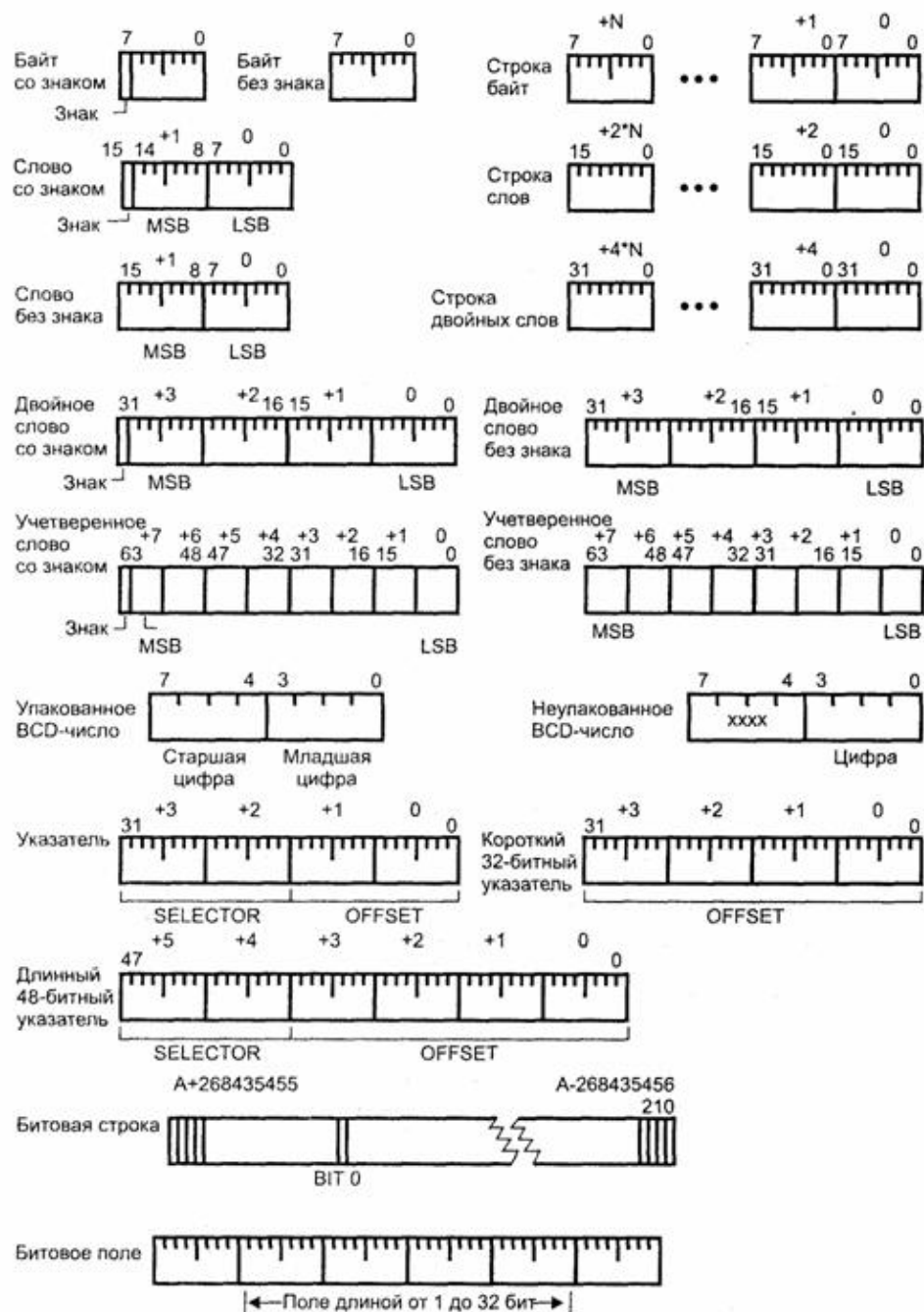
Количество и вид типов данных, аппаратно поддерживаемых современными платформами, очень велико, хотя различия между ними несущественны для всех, кто не занимается низкоуровневым программированием. Например, x86 аппаратно поддерживает следующие виды данных:

- Битовое поле (Bit Field) — группа до 32 смежных бит, располагающихся не более чем в 4 байтах;
- Битовая строка (Bit String) — набор смежных бит длиной до 4 Гбит;
- Байт (Byte) — 8 бит;
- Числа без знака: байт/слово/двойное/четверное слово (Unsigned Byte / Word / Double Word / Quad Word), 8/16/32/64 бит;
- Целые числа со знаком: байт/слово/двойное/четверное слово (Integer Byte / Word / Double Word / Quad Word). Единичное значение самого старшего бита (знак) является признаком отрицательного числа, которое хранится в дополнительном коде;
- Двоично-десятичные числа (BCD — Binary Coded Decimal):
 - 8-разрядные упакованные (Packed BCD), содержащие два десятичных разряда в одном байте;
 - 8-разрядные неупакованные (Unpacked BCD), содержащие один десятичный разряд в байте (значение бит 7:4 при сложении и вычитании несущественно, при умножении и делении они должны быть нулевыми).
- Строки байт, слов и двойных слов (Bit String, Byte String, Word String, Double Word String) длиной до 4 Гбайт.
- Указатели:
 - длинный указатель (48 бит) — 16-битный селектор (или сегмент) и 32-битное смещение;
 - короткий указатель — 32-битное смещение;
 - просто указатель (32 бит, единственный тип указателя для 8086 и 80286)
- 16-битный селектор (или сегмент) и 16-битное смещение.

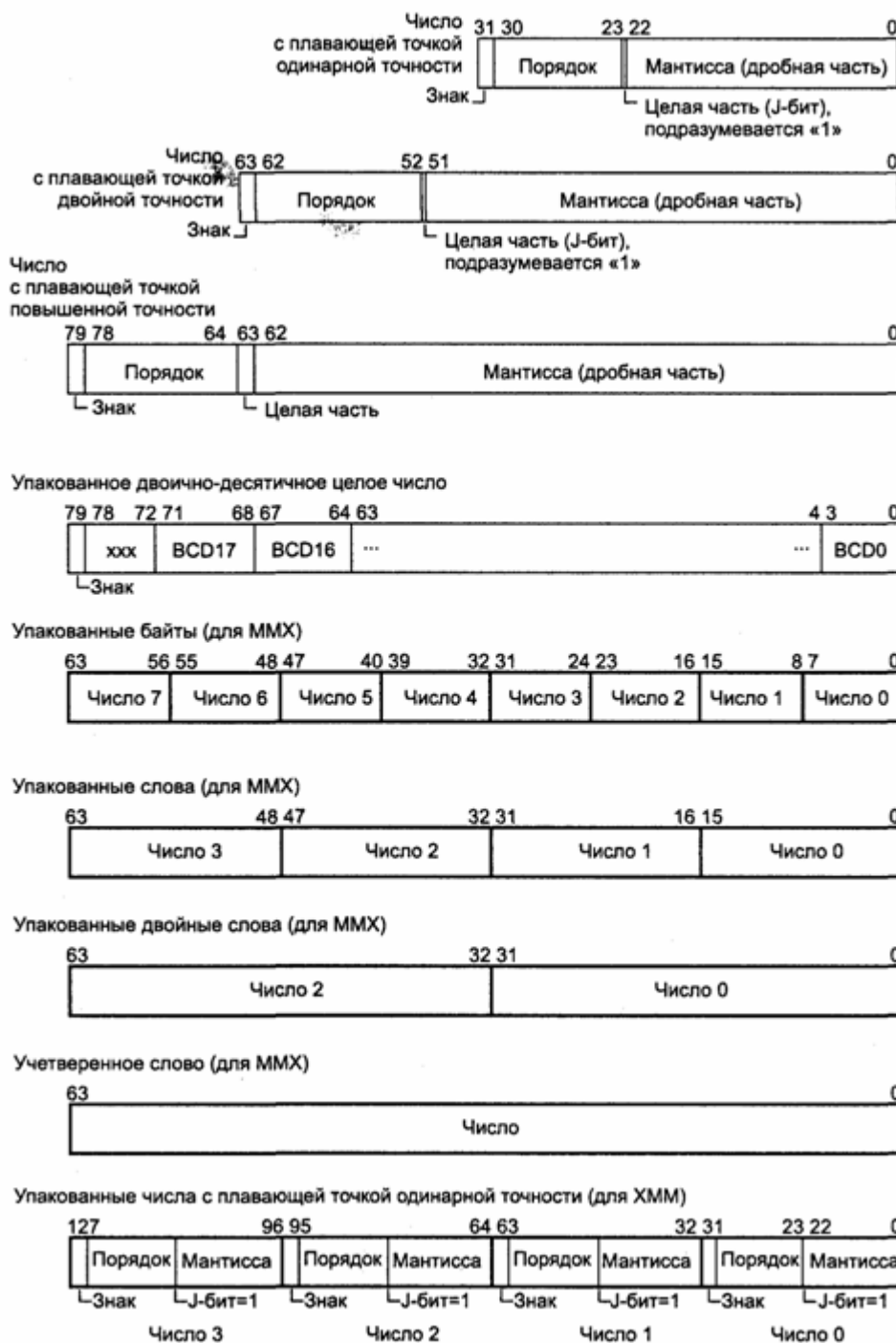
Числа в формате с плавающей точкой и упакованные 80-битные BCD-числа обрабатываются блоками FPU процессоров класса 486 и выше, а также сопроцессорами 8087/287/387. Упакованные 64-битные и 128-битные данные обрабатываются процессорами с MMX и SSE.

- Действительные числа в формате с плавающей точкой:
 - одинарной точности (Single Precision), 32 бит — 23 бит мантисса, 8 бит порядок;
 - двойной точности (Double Precision), 64 бит — 52 бит мантисса, 11 бит порядок;

- повышенной точности (Extended Precision), 80 бит — 64 бит мантисса, 15 бит порядок.
- Двоично-десятичные 80-битные упакованные числа (18 десятичных разрядов и знак)
- Упакованные действительные числа одинарной точности в формате с плавающей точкой, обрабатываются блоком XMM.
- Упакованные целые числа, знаковые и беззнаковые, обрабатываются блоком MMX:
 - упакованные байты (Packed byte) — восемь байт;
 - упакованные слова (Packed word) — четыре слова;
 - упакованные двойные слова (Packed doubleword) — два двойных слова;
 - учетверенное слово (Quadword) — одно слово.



Форматы данных, обрабатываемых целочисленным блоком АЛУ
всех 32-разрядных процессоров Intel



Типы данных, обрабатываемых блоками FPU/MMX и XMM

Математический сопроцессор предназначен для расширения вычислительных возможностей центрального процессора – выполнения арифметических операций, вычисления основных математических функций (тригонометрических, экспоненты, логарифма) и т. д. В разных поколениях процессоров он назывался по-разному – FPU (Floating Point Unit – блок чисел с плавающей точкой) или NPX (Numeric Processor eXtension – числовое расширение процессора). Сопроцессор IA-32 поддерживает семь типов данных: 16-, 32-, 64-битные целые числа; 32-, 64-, 80-битные числа с плавающей точкой и 18-разрядные числа в двоично-десятичном формате. Формат чисел с плавающей точкой соответствует стандартам IEEE 754 и 854.

С программной точки зрения сопроцессор и процессор выглядят как единое целое. В современных (486+) процессорах FPU располагается на одном кристалле с центральным процессором. Для процессоров 386 и ниже сопроцессор был отдельной микросхемой, подключаемой к локальной нише основного процессора. В любом случае сопроцессор исполняет только свои специфические команды, а всю работу по декодированию инструкции и доставке данных осуществляет CPU. Сопроцессор может выполнять вычисления парал-

тельно с центральным процессором, независимо от переключения задач в защищенном режиме. Как и основной процессор, сопроцессор может работать в реальном или защищенном режиме и переключать разрядность — 16 или 32. Переключение режимов влияет на формат отображения регистров сопроцессора в оперативной памяти, при этом формат используемых внутренних регистров не изменяется.

Сопроцессор оперирует данными в формате с плавающей точкой, который позволяет представлять существенно больше действительных чисел, чем целочисленное АЛУ центрального процессора. Арифметические операции (здесь под арифметическими понимаются операции, изменяющие значения операндов, а также операции сравнения) в FPU выполняются над 80-битными числами, представленными во внутреннем формате расширенной точности.

Форматы вещественных чисел представляют только множество дискретных значений множества чисел, расположенных на непрерывной бесконечной числовой оси. Диапазон и плотность значений зависят от выбранного формата представления. Заметим, что не все десятичные дроби могут быть представлены точно в двоичном коде. Так, например, дробь $1/10$ не имеет точного двоичного представления (аналогично тому, что $1/3 = 0,33333(3)$).

Сопроцессор контролирует числа, участвующие в арифметических операциях. При загрузке денормализованного операнда в регистр FPU и попытке выполнения арифметических инструкций хотя бы с одним денормализованным операндом сопроцессор фиксирует условие исключения #D. Денормализованные числа могут появляться при выполнении вычислений, в этом случае сопроцессор фиксирует факт исчезновения значащих разрядов и генерирует исключение #U. При попытке выполнения арифметических операций с нечислами, а также с недопустимыми значениями операндов (например, извлечение квадратного корня из отрицательного числа) вырабатывается исключение #I. При переполнении вырабатывается исключение #O, при попытке деления на нуль ненулевого операнда вырабатывается исключение #Z. Вместо автоматического выполнения округления сопроцессор может вырабатывать исключение #P.

3.1.2. Биты, байты, слова

Байт (byte) — это основная единица хранения и обработки цифровой информации. В большинстве вычислительных архитектур байт — это минимальный независимо адресуемый набор данных. В современных вычислительных системах байт считается равным восьми битам, в этом случае он может принимать одно из $256 (2^8)$ различных значений. Естественно, так было не всегда, и ряд ЭВМ 1950-х и 1960-х годов (БЭСМ-6, М-220) использовали 6-битовые символы в 48-битовых или 60-битовых машинных словах. В некоторых моделях Voughts размер символа был равен 9 битам. В советской ЭВМ Минск-32 использовался 7-битный байт, а в байте PDP-10 вообще 36 битов.

В результате, иногда в компьютерных стандартах и официальных документах для однозначного обозначения 8-битного слова используется термин «октет» (*octet*). Байтовая адресация памяти была впервые применена в системе IBM System/360. В более ранних компьютерах адресовать можно было только целиком машинное слово, состоявшее из нескольких байт, что затрудняло обработку текстовых данных. 8-битные байты были приняты в System/360, вероятно, из-за использования BCD-формата представления чисел: одна десятичная цифра (0-9) как раз требует 4 бита (тетраду) для хранения; один 8-битный байт может представлять две десятичные цифры. 6-битные байты могут хранить только по одной десятичной цифре, два бита остаются незадействованными. По другой версии, 8-битный размер байта связан с 8-битным же числовым представлением символов в кодировке EBCDIC.

Постепенно 8-битные байты стали стандартом де-факто, и с начала 1970-х в большинстве компьютеров байты состоят из 8 бит и размер машинного слова кратен 8 битам. Из соображений удобства единицы нетекстовых типов данных также делают кратными 8 битам, например:

- размер одного сэмпла в звуковых файлах равен 8, 16 или 24 битам,
- размер пикселя в системе RGB равен 24 битам (по 8 бит на цвет).

Ниббл (*nibble*, *nybble*), или полубайт — единица измерения информации, равная четырём двоичным разрядам (битам), удобна тем, что представима одной шестнадцатеричной цифрой, то есть является одним шестнадцатеричным разрядом. Переменная размера «ниббл» может принимать $2^4=16$ различных значений. В русском языке используется синоним «тетрада».

Машинное слово — машинно-зависимая величина, измеряемая в битах или байтах, равная разрядности регистров процессора и/или разрядности шины данных (обычно некоторая степень двойки). На ранних компьютерах размер слова совпадал также с минимальным размером адресуемой информации (разрядностью данных, расположенных по одному адресу); на современных компьютерах минимальным адресуемым блоком информации обычно является байт, а слово состоит из нескольких байтов.

В программировании слово обозначает естественную единицу данных, используемую конкретной машиной. Слово, как правило, имеет фиксированный размер, оно образуется группой битов, которые обрабатываются как единое целое аппаратными средствами компьютера. Количество битов в слове (также называемый размер слова, ширина или длина) является важной характеристикой любой конкретной архитектуры.

Размер слова отражается во многих аспектах структуры компьютера и доступных ему операций. Большинство регистров процессора, как правило, имеют размер слова или кратный ему, слово – как правило, максимальный блок данных, который может быть передан в память, прочитан или обработан за один такт работы. Также очевидно, слово определяет максимальное адресное пространство, доступное машине, т.к. длина адреса не может превышать длину слова. Размер слова, в подавляющем большинстве случаев, кратен степени двойки, современные процессоры (в т.ч. микроконтроллеры) имеют размер слова 8, 12, 16, 24, 32, 64 бита, компьютеры общего назначения – 64 бита. При этом размер внутренних регистров и ширина шины памяти может быть значительно больше – 2 или 4 слова, это делается в целях распараллеливания работы. DSP могут иметь нестандартный размер слова, зависящий от битности конкретного обрабатываемого сигнала – например, 9, 18, 39, 48, 60 бит. Некоторые старые машины использовали десятичные слова или BCD-код, некоторые вообще не имели слов фиксированной длины. В целях обратной совместимости, иногда процессоры поддерживают несколько длин слов.

Машинное слово напрямую определяет следующие параметры: размер целого числа, размер вещественного числа, максимальную длину адреса, количество данных, переданных за один такт, минимально выбираемое количество данных, длину инструкции. Таким образом, выбор слова является существенным параметром при проектировании архитектуры. Есть конструктивные соображения, которые поощряют использование слов разной длины, например, вопросы совместимости или ширина шины, но принцип экономии приводит к тому, что выбирается какой-то фиксированный размер.

Одним из классических факторов, влияющих на размер слова, например, было количество используемых символов в стандартной кодировке. До середины 1960-х годов байт чаще всего состоял из шести бит, что позволяло использовать не более 64 символов, но этого было достаточно. Поскольку слово чрезвычайно неудобно делать не кратным размеру байта, чаще всего оно имело длину 36 бит, что позволяло считать и передавать 6 символов за такт.

После IBM System/360, который использовал 8-битные символы, что позволяло различать регистр команд, стандартный размер символа стал равен 8 битам. Естественно, слово после этого стало 16, 32, 64-х битным. Многие ранние конструкции ЭВМ использовали то, что часто называют переменной длиной слова. При такой архитектуре числовой операнд не имел фиксированной длины, а его конец обозначался специальным символом-маркером. Такие машины часто использовали BDC-код. Среди них были IBM 702, 705, 7080, 7010, 1401 и 1620 и UNIVAC 1050.

Большинство из этих машин работали лишь с одним банком памяти, и каждая команда обычно читалась за несколько циклов, что сильно замедляло работу. Например, чтение инструкции из памяти IBM 1620 занимало 8 циклов, чтобы просто прочитать 12-битную команду. Усовершенствование следующего поколения ограничило число циклов шестью или четырьмя, если не требовалось читать оба поля адреса.

Модели памяти архитектуры ЭВМ также сильно зависят от размера слова. В частности, наименьший адресуемый блок часто выбирается в качестве такового. При таком подходе адреса, отличающиеся на 1, хранят в памяти смежные слова. Это естественная архитектура для машин, которые оперируют преимущественно словами, она позволяет минимизировать длину поля адреса при выборке соседних команд. Когда машина оперирует преимущественно байтами (а размер слова, как мы помним, кратен байту, но не всегда ему равен!), то более выгодно использовать байтовую адресацию, а значит, слово может и не совпадать с минимально адресуемым участком ОЗУ. Этот подход был использован в IBM 360 и с тех пор стал наиболее распространенным. Есть еще одно важное соображение. Если слово будет иметь размер, кратный степени двойки, то вместо умножения и деления во многих случаях можно будет использовать простой сдвиг.

В середине 1970-х гг. DEC VAX стал преемником знаменитого PDP-11. Он использовал слово длиной 16 бит, а для данных размером 32 бита использовался термин «длинное слово» или «двойное слово». Это отличало его от предыдущих машин, где словом назывался максимальный блок данных, а его половина – соответственно полусловом. В современной терминологии под словом обычно понимают 16-битную величину, двойное слово (DWORD) – 32 бита, учетверенное слово (QWORD) – 64 бита. В старом ассемблере x86 до сих пор многие мнемоники команд используют сокращения D и Q соответственно, чтобы явно указать длину инструкции.

В таблице приведено все разнообразие размеров машинного слова и байтов на разных платформах, а также размер чисел и длина инструкций. Из нее можно извлечь огромное количество подробностей о той или иной архитектуре и ее эволюции.

Год	Машина	Слово <i>w</i>	Целое	Вещественное	Инструкция	Адрес	Байт
1837	Babbage Analytical engine	50 <i>d</i>	<i>w</i>	-----	5 различных перфокарт с инструкциями, размер неизвестен	<i>w</i>	-----
1941	Zuse Z3	22 <i>b</i>	-----	<i>w</i>	8 <i>b</i>	<i>w</i>	-----
1942	ABC	50 <i>b</i>	<i>w</i>	-----	-----	-----	-----

1944	Harvard Mk I	23d	w	-----	24b	-----	-----
1946 1948 1953	ENIAC + Panel #16 + Panel #26	10d	w, 2w w w	-----	(2d, 4d, 6d, 8d) {2d, 4d, 6d, 8d}	----- ---- w	-----
1951	UNIVAC I	12d	w	-----	1/2 w	w	d
1952	IAS	40b	w	-----	1/2 w	w	5b
1952	Fast Universal Digital Computer M-2	34b	w	w	34 b = 4 b код операции + 3·10 b адрес	10b	-----
1952	IBM 701	36b	1/2 w, w	-----	1/2 w	1/2 w,	6b
1952	UNIVAC 60	n d	1d, ..., 10d	-----	-----	-----	2d, 3d
1953	IBM 702	n d	0d, ..., 511d	-----	5d	d	d
1953	UNIVAC 120	n d	1d, ..., 10d	-----	-----	-----	2d, 3d
1954 1955	IBM 650 + IBM 653	10d	w	----- w	w	w	2d
1954	IBM 704	36b	w	w	w	w	6b
1954	IBM 705	n d	0d, ..., 255d	-----	5d	d	d
1954	IBM NORC	16d	w	w, 2w	w	w	----- -
1954	IBM 305	n d	0d, ..., 100d	-----	10d	d	d
1957	Autonetics Recomp I	40b	w, 79b, 8d, 15d	-----	1/2 w	1/2 w,	5b
1958	UNIVAC II	12d	w	-----	1/2 w	w	d
1958	SAGE	32b	1/2 w	-----	w	w	6b
1958	Autonetics Recomp II	40b	w, 79b, 8d, 15d	2w	1/2 w	1/2 w,	5b
1958	Сетунь	6 trit	до 6 trit	?	до 3 trit	?	4 trit
1959	IBM 1401	n d	1d, ...	-----	d, 2d, 4d, 5d, 7d, 8d	d	dd
1959	IBM 1620	n d	2d, ...	4d, ..., 102d	12d	d	2d
1960	LARC	12d	w, 2w	w, 2w	w	w	2d
1960	CDC 1604	48b	w	w	1/2 w	w	6b
1960	IBM 1410	n d	1d, ...	-----	d, 2d, 6d, 7d, 11d, 12d	d	d
1960	IBM 7070	10d	w	w	w	w, d	2d
1960	PDP-1	18b	w	-----	w	w	6b
1960	Elliott 803	36b + 1b CRC	?	?	?	?	?
1961	IBM 7030 (Stretch)	64b	1b, ..., 64b 1d, ..., 16d	w	1/2 w, w	b, 1/2 w , w	1b, ..., 8b
1961	IBM 7080	n d	0d, ..., 255d	-----	5d	d	d
1962	GE-6xx	36b	w, 2w	w, 2w, 80b	w	w	6b, 9b
1962	UNIVAC III	25b, 6d	w, 2w, 3w, 4w	-----	w	w	6b

1962	Autonetics D-17B *	27b	11b, 24b	-----	24b	w	6b
1962	UNIVAC 1107	36b	$\frac{1}{6}w, \frac{1}{3}w, \frac{1}{2}w, w$	w	w	w	-----
1962	IBM 7010	<i>nd</i>	1d, ...	-----	<i>d, 2d, 6d, 7d, 11d, 12d</i>	<i>d</i>	<i>d</i>
1962	IBM 7094	36b	w	w, 2w	w	w	6b
1963 1966	PDP-6 PDP-10	36b	w	w, 2w	w	w	6b, 9b
1963	Gemini *	39b	26b	-----	$\frac{1}{3}w$	$\frac{1}{3}w, w$	-----
1963 1966	Apollo *	15b	w	-----	w, 2w	w	-----
1963	Saturn Launch Vehicle *	26b	w	-----	$\frac{1}{2}w$	w	-----
1964	CDC 6600	60b	w	w	$\frac{1}{4}w, \frac{1}{2}w$	w	6b
1964	Autonetics D-37C *	27b	11b, 24b	-----	24b	w	4b, 5b
1965	IBM 360	32b	$\frac{1}{2}w, w, d, \dots, 16d$	w, 2w	$\frac{1}{2}w, w, \frac{1}{2}w$	8b	8b
1965	UNIVAC 1108	36b	$\frac{1}{6}w, \frac{1}{4}w, \frac{1}{3}w, \frac{1}{2}w, w, 2w$	w, 2w	w	w	6b, 9b
1965	PDP-8	12b	w	-----	w	w	8b
1970	PDP-11	16b	w	2w, 4w	w, 2w, 3w	8b	8b
1971	Intel 4004 **	4b	w, d	-----	2w, 4w	w	-----
1972	Intel 8008	8b	w, 2d	-----	w, 2w, 3w	w	8b
1972	Calcomp 900	64b	w	-----	w, 2w	w	-----
1974	Intel 8080	8b	w, 2w, 2d	-----	w, 2w, 3w	w	8b
1975	ILLIAC IV ***	64b	w	$\frac{1}{2}w, w$	w	w	8b
1975	Motorola 6800	8b	w, 2d	-----	w, 2w, 3w	w	8b
1975	MOS Tech. 6501 / 6502	8b	w, 2d	-----	w, 2w, 3w	w	-----
1976	Cray-1 ***	64b	24b, w	w	$\frac{1}{4}w, \frac{1}{2}w$	w	8b
1976	Zilog Z80	8b	w, 2w, 2d	-----	w, 2w, 3w, 4w, 5w	w	8b
1978 1980	Intel 8086 + Intel 8087	16b	$\frac{1}{2}w, w, 2d, w, 2w, 4w$	2w, 4w, 5w, 17d	$\frac{1}{2}w, w, \dots, 7w$	w	8b
1978	VAX-11/780 ***	32b	$\frac{1}{4}w, \frac{1}{2}w, w, 1b, \dots, 32b, 1d, \dots, 31d$	w, 2w	$\frac{1}{4}w, \dots, 14\frac{1}{2}w$	8b	8b
1979	Motorola 68000	32b	$\frac{1}{4}w, \frac{1}{2}w, w$	-----	$\frac{1}{2}w, w, \dots, 7\frac{1}{2}w$	8b	8b
1982 1983	Motorola 68020 + Motorola	32b	$\frac{1}{4}w, \frac{1}{2}w, w$	----- w, 2w, $2\frac{1}{2}w$	$\frac{1}{2}w, w, \dots, 7\frac{1}{2}w$	8b	8b

	68881						
1985	Intel 80386	32b	$1/2 w, w, 2w, 4w, 2d$	$2w, 4w, 5w, 17d$	$1/2 w, w, \dots, 7w$	8b	8b
1985	ARM1	32b	w	-----	w	8b	8b
1985	MIPS	32b	$1/4 w, 1/2 w, w$	w, 2w	w	8b	8b
1989	Motorola 68040	32b	$1/4 w, 1/2 w, w, 2d$	$w, 2 w, 2\frac{1}{2}w$	$1/2 w, w, \dots, 7\frac{1}{2}w$	8b	8b
1991	Alpha	64b	$1/4 w, 1/2 w, w$	w, 2w	$1/2 w$	8b	8b
1991	Cray C90 ***	64b	32b, w	w	$1/4 w, 1/2 w, 48$	w	8b
1991	PowerPC	32 – 64b	$1/4 w, 1/2 w, w$	w, 2w	w	8b	8b
2000	IA-64	64b	$8b, 1/4 w, 1/2 w, w$	$1/2 w, w$	41b	8b	8b
2002	XScale	32b	w	w, 2w	$1/2 w$	8b	8b

* - бортовые компьютеры ракет,

** - начало эпохи микропроцессоров,

*** - суперкомпьютеры, все остальное – микропроцессоры.

Здесь: b – бит, d – децит (десятичная цифра), w – размер машинного слова, n – переменное значение.

Помимо длины слова, значение имеет, в каком порядке производить чтение данных. Существует несколько основных порядков.

Порядок от старшего к младшему или (big-endian, дословно: «тупоконечный») – запись начинается со старшего и заканчивается младшим. Этот порядок является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому порядок байтов от старшего к младшему часто называют сетевым порядком байтов (network byte order). Этот порядок байтов использовался такими машинами, как IBM 360/370/390, процессорами Motorola 68000 и SPARC (отсюда третье название — Motorola byte order, MBO).

В этом же виде (используя представление в десятичной системе счисления) записываются числа индийско-арабскими цифрами в письменностях с порядком знаков слева направо (латиница, кириллица). Для письменностей с обратным порядком (арабская) та же запись числа воспринимается как «от младшего к старшему». Порядок байтов от старшего к младшему также применяется во многих форматах файлов — например, PNG, FLV, EBML.

Порядок от младшего к старшему или (little-endian, дословно: «остроконечный») – запись начинается с младшего и заканчивается старшим. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют Intel byte order, IBO. В некоторых кругах используется название VAX order, например в документации Perl. В противоположность «тупоконечному» порядку, IBO поддерживает куда меньшее количество кроссплатформенных протоколов и форматов данных. Заметные исключения составляют спецификация USB и шины PCI, таблица разделов GUID и рекомендации FidoNet.

Многие процессоры могут работать и в порядке от младшего к старшему, и в обратном, например, ARM, PowerPC (но не PowerPC 970), DEC Alpha, MIPS, PA-RISC и IA-64. Обычно порядок байтов выбирается программно во время инициализации операционной системы, но может быть выбран и аппаратно переключателями на материнской плате. В этом случае правильнее говорить о порядке байтов операционной системы. Переключаемый порядок байтов иногда называют bi-endian.

Смешанный порядок байтов (middle-endian) иногда используется при работе с числами, длина которых превышает машинное слово. Число представляется последовательностью машинных слов, которые записываются в формате, естественном для данной архитектуры, но сами слова следуют в обратном порядке. Классический пример middle-endian — представление 4-байтных целых чисел на 16-битных процессорах семейства PDP-11 (известен как PDP-endian). Для представления двухбайтных значений (слов) использовался порядок little-endian, но четырехбайтное двойное слово записывалось от старшего слова к младшему. В процессорах VAX и ARM используется смешанное представление для длинных вещественных чисел.

Существенным достоинством little-endian по сравнению с big-endian порядком записи считается возможность «невной типизации» целых чисел при чтении меньшего объема байт (при условии, что читаемое число помещается в диапазон). Так, если в ячейке памяти содержится число 0x00000022, то прочитав его как int16 (два байта), мы получим число 0x0022, прочитав один байт — число 0x22. Однако это же может считаться и недостатком, потому что провоцирует ошибки потери данных.

Обратно считается, что у little-endian, по сравнению с big-endian есть неочевидность значения байтов памяти при отладке – последовательность байтов A1, B2, C3, D4 на самом деле значит 0xD4C3B2A1, для big-endian же эта последовательность читалась бы естественным для арабской записи чисел образом – 0xA1B2C3D4. Наименее удобным в работе считается middle-endian формат записи; он сохранился только на старых платформах.

Для записи длинных чисел (чисел, длина которых существенно превышает разрядность машины) обычно предпочтительнее порядок слов в числе little-endian (поскольку арифметические операции над длинными числами производятся от младших разрядов к старшим).

3.2. Польская запись и стековые машины

3.2.1. Принципы бесскобочной записи

Помимо порядка чтения бит, еще одним важным вопросом является последовательность выполнения арифметических действий. Дело в том, что одними элементарными операциями обходиться можно, но до них еще надо дойти, особенно, если перед нами сложное выражение. Каким образом объяснить компьютеру иерархию скобок в выражении и порядок выполнения действий? В математике существует древняя традиция помещать оператор между операндами, например, вот так: $(3 + 5) \cdot (5 + 3) \cdot 5$. В итоге, компьютер не может начать вычисления, не проведя разбор всего выражения и только после установления правильного порядка действий, контроля количества открывающих и закрывающих скобок и прочего он может вычислить результат.

Такая форма записи именуется инфиксной, и она имеет много неудобств. Во-первых, наличие скобок, явно задающих порядок выполнения операций, приводит к лишнему количеству символов и повышенной вероятности ошибок при их интерпретации. Во-вторых, инфиксная запись совершенно не подходит для работы с такой удобной моделью памяти, как стек, а на нем основаны многие алгоритмы, целые языки программирования и даже железная часть некоторых компьютеров. Наконец, в-третьих, инфиксная запись по большей части произвольна и основывается на простых договоренностях и традициях, что также нелогично и нежелательно, так как плохо переносится в новые реалии.

Например, все знают, что умножение имеет приоритет перед сложением, а значит, эту операцию можно не «огораживать» скобками. А вот имеет ли приоритет сдвиг влево на два бита перед конъюнкцией? А сдвиг вправо на 4? В годы проектирования первых ЭВМ вопрос был далеко не праздным - этого никто не знал, поскольку соответствующая математическая традиция еще не успела выработаться.

Знаменитый математик и логик Ян Лукасевич задумался над подобными вопросами еще в начале XX века, когда об ЭВМ речи не было, но его наработки оказали решающее влияние на программирование и архитектуру многих машин. Он изобрел т.н. нотацию Лукасевича для записи математических операций или просто польскую запись.

Помните разделимые коды, которые можно однозначно декодировать по мере поступления, благодаря свойству пост- или префиксной записи? Аналогичную идею предложил Лукасевич – давайте выкинем скобки, переместим знаки и поставим их спереди от операторов, а читать будем по порядку.

Например, запись $(3 + 5) \cdot (5 + 3) \cdot 5$ в прямой или префиксной записи превратится в $.. +35 + 535$. Так как любая простая арифметическая операция является бинарной, то ее префиксное представление не может быть интерпретировано двояко, поэтому нет необходимости использовать скобки. Вычисление умножения задерживается до тех пор, пока не будут считаны оба операнда (результаты сложения 3 и 5 и 5 и 3). Как и в любой другой нотации, выражения, находящиеся в глубине, вычисляются первыми, но в польской записи глубина выражения определяется порядком, а не скобками.

Префиксная запись в простой арифметике представляет в значительной степени академический интерес. Она была использована для некоторых коммерческих вычислительных машин (например, HP-11C), но активнее всего получила развитие в программировании. Изучение префиксной записи часто является первым шагом в области конструирования компиляторов.

Префиксная запись широко применяется в s-выражениях в языке программирования Lisp, где скобки необходимы, поскольку арифметические операторы обладают различной арностью. Язык программирования Ambi использует польскую запись для арифметических операций и структуры программы. Также важно отметить, что количество операндов в выражении должно быть на один больше, чем количество операций, иначе выражение не имеет смысла (учитывая, что в выражении используются только бинарные операции). Этому можно не придавать значения при работе с длинными, сложными выражениями, что повлечет за собой ошибки. Поэтому необходимо обращать внимание на количество операций и операндов при использовании префиксной нотации.

Префиксная запись особенно популярна в стековых языках благодаря свойственной им возможности легко различать порядок операций без использования скобок. Для выявления порядка вычисления операторов в префиксной нотации даже нет необходимости запоминать всю операционную иерархию, как при инфиксной нотации. Вместо того, чтобы анализировать выражение для обнаружения оператора, который нужно вычислить первым, нужно считывать выражение слева направо, рассматривая оператор и ближайшие к нему два операнда. Если среди этих операндов находится еще один оператор, то вычисление первого оператора

откладывается до тех пор, пока не будет вычислен новый оператор. Итерации этого процесса повторяются до тех пор, пока оператор не будет вычислен, что должно, в конечном счете, произойти, если в выражении количество операндов на один больше, чем количество операций. Как только оператор вычислен, он и его два операнда заменяются полученным значением (операндом). Поскольку оператор и два операнда заменяются вычисленным операндом, то выражение становится на один оператор и операнд короче. После этого в выражении также остается N операторов и N+1 операнд, что позволяет итеративно продолжать процесс.

Большую славу снискала иная разновидность такой нотации, предложенная уже во времена первых ЭВМ – обратная польская нотация – форма записи математических выражений, в которой операнды расположены перед знаками операций. Также она именуется обратной польской записью, обратной бесскобочной записью, постфиксной нотацией или, самое популярное, польской инверсной записью – ПОЛИЗ.

Она настолько хорошо ложится на архитектуру машин со стековой памятью, что даже одно из их определений гласит – стековой машиной называется ЭВМ, использующая ПОЛИЗ. Обратная польская нотация была разработана австралийским философом и компьютерным инженером Чарльзом Хэмблином в середине 1950-х на основе польской нотации.

Первыми компьютерами, поддерживающими обратную польскую нотацию, были KDF9 от English Electric Company, который был анонсирован в 1960 и выпущен (появился в продаже) в 1963, и Burroughs B5000, выпущенный в том же 1963. Один из проектировщиков B5000, Р. С. Бартон позже написал, что разработал обратную польскую запись независимо от Хэмблина примерно в 1958, в процессе чтения книги по символической логике и до того, как познакомился с работой Хэмблина.

Компания Friden перенесла ПОЛИЗ в настольные калькуляторы, выпустив в июне 1964 модель EC-130. А в 1968 инженеры Hewlett-Packard разработали настольный калькулятор 9100A с поддержкой ПОЛИЗ. Этот калькулятор сделал обратную польскую нотацию популярной среди учёных и инженеров, даже несмотря на то, что в ранней рекламе 9100A ПОЛИЗ не упоминалась. В 1972 калькулятор HP-35 с поддержкой ПОЛИЗ стал первым научным карманным калькулятором.

В 1971 году появился оригинальный язык программирования Forth, языковая машина которого имеет двухстековую структуру и где все вычисления проводятся на стеке. В этом языке ОПН является естественным способом записи любых операций с данными, хотя возможна, при желании, реализация и обычной записи.

ПОЛИЗ применялась в советском инженерном калькуляторе БЗ-19М (совместная разработка с ГДР), выпущенном в 1976 году. Все выпускаемые в СССР вплоть до конца 1980-х годов программируемые микрокалькуляторы, за исключением «Электроника МК-85», использовали ПОЛИЗ — она проще реализовывалась и позволяла обойтись в программировании вычислений меньшим числом команд, по сравнению с обычной алгебраической нотацией. Дело в том, что количество программной памяти в этих моделях всегда было критическим ресурсом (не более 105 ячеек, а команда занимала 1-2 ячейки). ПОЛИЗ используется в современных российских программируемых калькуляторах «Электроника МК-152» и «ЭЛЕКТРОНИКА МК-161», что обеспечивает их совместимость с программами, написанными для советских калькуляторов.

Автоматизация вычисления выражений в обратной польской нотации основана на использовании стека. Алгоритм вычисления для стековой машины элементарен:

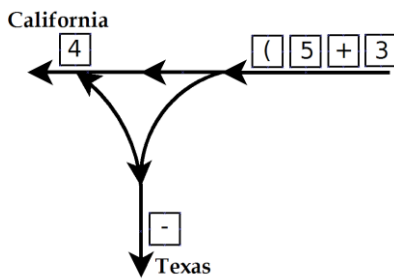
- Обработка входного символа:
 - Если на вход подан операнд, он помещается на вершину стека,
 - Если на вход подан знак операции, то соответствующая операция выполняется над требуемым количеством значений, извлечённых из стека, взятых в порядке добавления,
 - Результат выполненной операции кладётся на вершину стека.
- Если входной набор символов обработан не полностью, перейти к шагу 1.
- После полной обработки входного набора символов результат вычисления выражения лежит на вершине стека.

Реализация стековой машины, как программная, так и аппаратная, чрезвычайно проста и может быть очень эффективной. Обратная польская запись совершенно унифицирована – она принципиально одинаково записывает унарные, бинарные, тернарные и любые другие операции, а также обращения к функциям, что позволяет не усложнять конструкцию вычислительных устройств при расширении набора поддерживаемых операций, поэтому обратная польская запись используется почти во всех научных и программируемых микрокалькуляторах.

Перевод выражения в обратную польскую запись осуществляется т.н. алгоритмом Дейкстры, названным им «сортировочная станция». Представим, что у нас есть состав с вагонами, где каждый везет какой-нибудь символ. Это наша изначальная инфиксная формула. Далее мы проходим стрелку, на которой вагоны с цифрами не задерживаются и едут, как шутил Дейкстра, прямо в Калифорнию, а вагоны с символами должны заехать на сортировочную станцию в Техас по определенным правилам.

	↓	+	−	×	÷	(	)
↓	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
×	2	2	2	2	2	1	2
÷	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Вот описание всех возможных ситуаций на такой дороге, первый вагон (помеченный символом $\perp$) всегда едет в Техас. Числа соответствуют следующим ситуациям:



1. Вагон на стрелке отправляется в Техас,
2. Последний вагон, направившийся в Техас, разворачивается и направляется в Калифорнию,
3. Вагон, находящийся на стрелке, и последний вагон, отправившийся в Техас, угоняются и исчезают,
4. Остановка. Символы, находящиеся на Калифорнийской ветке, представляют собой формулу в обратной польской записи, если читать слева направо,
5. Остановка. Произошла ошибка. Изначальная формула была некорректно сбалансирована.

После каждого действия производится новое сравнение вагона, находящегося у стрелки (это может быть тот же вагон, что и в предыдущем сравнении, а может быть следующий вагон), и вагона, который на данный момент последним ушёл на Техас. Этот процесс продолжается до тех пор, пока не будет достигнут шаг 4. Отметим, что линия на Техас используется как стек, где отправка вагона в Техас – это помещение элемента в стек, а разворот отправленного в Техас вагона в сторону Калифорнии – это выталкивание элемента из стека. Порядок следования переменных в инфиксной и постфиксной записи одинаков. Однако порядок следования операторов не всегда один и тот же. В обратной польской записи операторы появляются в том порядке, в котором они будут выполняться.

Как работает физическая реализация такого устройства? Ветка, ведущая в Техас, как вы уже догадались, это стек. В общем случае, стек – это структура данных, представляющая список элементов, организованных по принципу LIFO (last in – first out, «последним пришёл – первым вышел»). Аппаратный стек в ЭВМ часто называется магазином – по аналогии с магазином огнестрельного оружия. Идею стека предложил Тьюринг ещё в 1946 г.

Чтобы структура стала стеком, для нее должны быть определены две операции, условно называемые pop и push – вытолкнуть из стека и поместить в стек. С этой точки зрения, в некоторых языках (например, Lisp, Python) стеком можно назвать любой список, так как для него эти операции определены по умолчанию. В C++, например, стандартная библиотека имеет класс с реализованной структурой и методами.

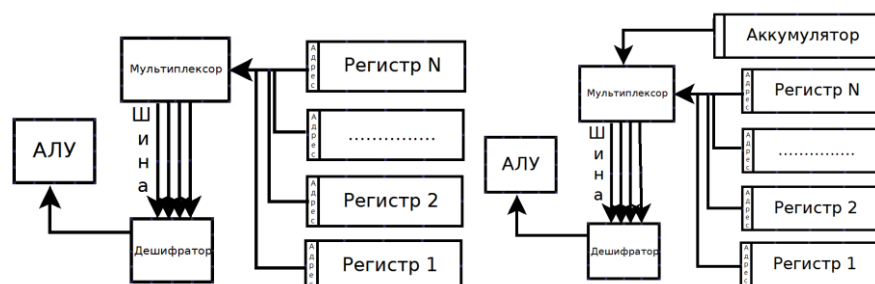
3.2.2. Стековая машина

Аппаратный стек – непрерывная область памяти, адресуемая специальными регистрами ESP (указатель стека) и SS (селектор сегмента стека). До использования стека он должен быть инициализирован так, чтобы регистры SS:ESP указывали на область реальной оперативной памяти (стек в ПЗУ, естественно, работать не может). Прикладные программы, как правило, от операционной системы получают готовый к употреблению стек. В защищенном режиме сегмент состояния задачи содержит четыре селектора сегментов стека (для разных уровней привилегий), но в каждый момент используется, естественно, только один стек. Аппаратный стек применяется в случаях, когда необходимо организовать прерывания вызовов и возвратов, либо в случаях, когда нужно организовать временное хранилище данных места в памяти (переменные, параметры функции).

Программный вид стека используется для обхода структур данных, например, дерево или граф. При использовании рекурсивных функций также будет применяться стек, но аппаратный его вид. Для отслеживания точек возврата из подпрограмм используется стек вызовов. Арифметические сопроцессоры, программируемые микрокалькуляторы и язык Forth используют полностью стековую модель вычислений.

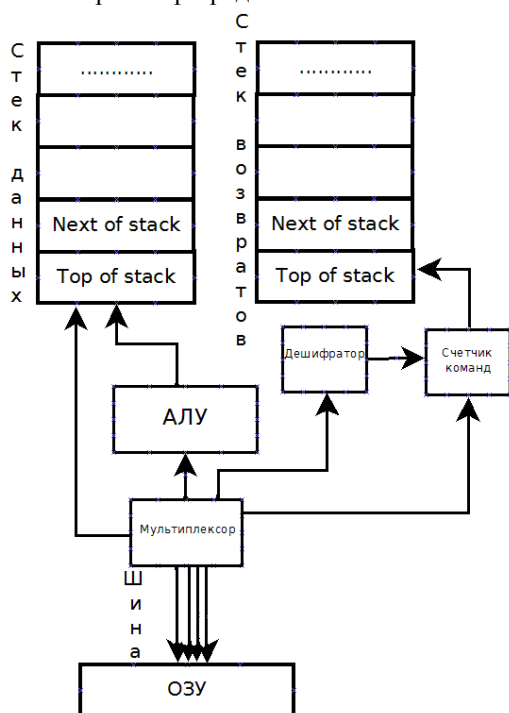
Для нас же интересен другой аспект стека – можно ли создать компьютер, вообще не имеющий традиционных регистров, хранящих информацию, и организовать вычисления чисто на стеке? Конечно, их понадобится несколько, как минимум, стек команд и стек данных, но технически это возможно и, более того, неоднократно воплощено в железе. Чистой стековой ЭВМ (иногда Forth-машиной) называется компьютер, выполняющий посредством аккумулятора операции только над данными в стеке.

Чем физически отличается стековая машина? Вообще, реализаций модели памяти ЭВМ не так уж и много. Самой простой идеей является разделить всю память на независимые части – регистры, кратные машинному слову, каждый из которых имеет адрес и хранит команду или данные. Тогда взаимодействие памяти и вычислителя (арифметико-логическое устройство, АЛУ) в ЭВМ будет выглядеть примерно вот так:



Типичный регистровый процессор и процессор с аккумулятором

Достоинство этого подхода заключается в простоте программирования. Кроме этого, чем больше регистров имеет процессор, тем меньше выполняется пересылка данных при решении задач и возрастает производительность. Однако при этом обилие регистров усложняет схему, ведь нужны не только они, но и дешифратор команд, и узлы, вырабатывающие разрешение на запись данных. Кроме того, увеличение числа регистров затрагивает и разрядность полей операндов, и это еще не все. Считанные данные нужно прогнать через мультиплексор для подачи на общую выходную шину. При увеличении числа регистров пропорционально растет разрядность шины и сложность мультиплексора, а это совсем плохо.



Следующим подходом является использование специального регистра – аккумулятора, который играет роль своеобразного кэша. Достоинством этого метода является то, что в систему команд можно ввести специальные инструкции для записи в аккумулятор, и их длина существенно уменьшится за счет того, что не нужно будет снова передавать адрес. Таким образом, во многих случаях в работе процессор будет использовать аккумулятор для хранения промежуточного результата, без обращения к основным регистрам. Естественно, процессор может иметь несколько аккумуляторов, причем их адрес уже будет закодирован в самой команде и по-прежнему не потребует дополнительной нагрузки на шину и мультиплексор.

Впервые такой подход был применен в архитектуре Intel 8080, но неудобство его заключается в повышенной сложности программирования – необходимо каждую часть программы начинать с команд пересылок содержимого аккумуляторов, чтобы не потерять эти значения.

Альтернативой такой схеме является та самая Forth-машина или стековая архитектура. В такой модели необходимо минимум два стека – один для хранения данных, другой для хранения меток возвратов – чтобы иметь возможность полноценной работы с подпрограммами. Ее обобщенную схему вы можете видеть слева. Данные могут быть считаны только из вершины стека (как правило, командой TOS) или сле-

дующей ячейки (NOS). Так же, конечно, должны быть предусмотрены команды загрузки и выгрузки и обмена содержимым ячеек.

Стековая архитектура активно использует ОЗУ, а также крайне экономична – требует минимум блоков, состоящих из минимального числа вентилях. Для работы со стеком не нужно указывать адреса операндов, поэтому иногда при ее описании используется термин «0-операндная» или «безоперандная архитектура», часто сокращается до ROSC (Removed Operand Set Computer).

По существу, мы получаем машину для исполнения выражений, записанных в постфиксной нотации. Помимо экономичности, в стековой машине не нужно сохранять контекст при прерываниях, меньше времени тратится на вызов подпрограмм, команды не имеют операндов и потому объектный код программы намного компактнее, при компиляции программы отпадает необходимость в распределении регистров, управление процессором становится намного проще.

Однако не все так хорошо. Помимо проблем с производительностью, такие машины сложны для системного программирования из-за непривычной логики языка Forth и его ограничений, всегда существует опасность переполнения стека и, конечно, отсутствуют любые формы параллельности. Такая архитектура обычно обладает средним быстродействием, которое практически невозможно увеличить из-за интенсивного использования ОЗУ. В результате получаем дешевый процессор, программируемый на языке, близком к машинному, обладающий умеренным быстродействием. Для полноценных вычислений его использовать невыгодно, тем не менее, такие процессоры нашли широкое применение в качестве контроллеров и встроенных компьютеров.

Хотя Forth и C были задуманы почти в одно и то же время, их происхождение значительно отличалось. C, как и UNIX, был рожден в одном из оплотов информатики — Bell Labs. В течение долгого времени качественные C-компиляторы было доступны только в среде UNIX и компания AT фактически отдала C университетам. Таким образом, многие поколения студентов были воспитаны на UNIX и C.

Язык же Forth был изобретен Чарльзом Муром, внештатным программистом, который писал программы для обработки данных на фабрике по производству ковров в Stanford Linear Accelerator Center. Мур был знаком со многими языками программирования, включая диалекты ассемблера для разных машин, Fortran, COBOL, PL/1 и APL, но чувствовал, что все они не вполне отвечали его потребностям. Он экспериментировал с различными трансляторами и в начале 1960-х пришел к идее использовать стек, чтобы передавать параметры и результаты между интерпретируемыми командами.

Его первый интерпретатор был написан на Fortran для IBM 1130 во время его работы на фабрике Mohasco Industries в 1968 г. Несколько позже Мур перенес Forth на Honeywell 316. В 1973 Мур, Элизабет

Ратэр и еще несколько человек сформировали FORTH Inc., чтобы продавать язык как коммерческое изделие. Вскоре он был портирован на ряд систем, из которых наиболее важным был легендарный PDP-11, известный у нас как «Электроника-60». Он оставался основной Forth-машиной долгое время.

Затем Forth был портирован на Intel 8080, TI 9900, 6502, PACE и даже Motorola 6800. Впоследствии было разработано несколько специализированных процессоров с поддержкой Forth, например, Novix NC 4016 Чарльза Мура, MF1600 Алана Винфилда и WISC CPU/32 Фила Купмана (Writable Instruction Set Computer — компьютер с перезаписываемой системой команд). Проект чипа Novix был в конечном счете продан фирме Harris Semiconductor, где он был перепроектирован в RTX 2000; эта линейка микроконтроллеров производится до сих пор.

СССР традиционно испытывал проблемы с производством сложных и дорогих компьютеров, в результате простота стековой архитектуры очень привлекла отечественных разработчиков, и здесь Forth пережил второе рождение. Идеи стековой архитектуры причудливо скрестились с концепцией теговой ЭВМ, позаимствованной у машин семейства Burroughs, породив настоящего монстра с соответствующей легендой — суперкомпьютер «Эльбрус». История его сложна и содержит слишком много лжи и темных пятен, и прочих, не относящихся к архитектуре компьютеров, вещей, к тому же это была более теговая, нежели стековая, машина, поэтому рассказ о ней мы отложим до обсуждения низкоуровневого программирования.

Если переместиться от сложных архитектур к изначально простым чисто стековым машинам, то в СССР их производством занимался один из первых отечественных кооперативов, кооператив «Форт-Инфо», организованный в 1988 году и преобразованный в 1991 году в инновационно-техническую фирму «Технофорт».

В результате трехлетней работы в 1991 г. был создан самый быстрый в СССР микропроцессор — 16-рядный Форт-процессор Дофин-1610. Предназначенный для систем реального времени, он превосходил производимые в то время в СССР микропроцессоры по быстродействию в 50 раз⁵² и выпускался опытными партиями в минском НПО «Интеграл». Основными заказчиками фирмы до 1995 года были предприятия Санкт-Петербурга, в первую очередь ВПК, стремившиеся применить новые отечественные устройства на базе Дофин-1610 в своих разработках и системах автоматизации. Была спроектирована новая, усовершенствованная архитектура процессора — Дофин-1620 (совместно с минской фирмой НКТ, организованной в 1992 г.). «Технофорт» выполнил ряд НИРовских работ, на основании которых было подготовлено решение о постановке Форт-процессора на серийное производство в Зеленограде. К сожалению, экономический кризис и фактический упадок ВПК привели к свертыванию этих работ.

Первые два варианта чипа — Дофин-1610 и 1620, были, по традиции, клонами, только уже не процессоров, а микроконтроллеров Novix NC 4016. Разработчики повторили его, пользуясь при этом лишь спецификацией на архитектуру и систему команд⁵³. В результате получился совместимый кристалл, показавший хорошие параметры быстродействия для своего времени. Характерной особенностью Дофин-1610 стало использование дополнительной шины для подключения памяти стеков параметров и возвратов.

Во втором процессоре были добавлены внутренние стеки и урезаны шины для подключения внешних стеков. Кроме того, в него введена развитая система прерываний и аппаратный умножитель 16·16. В результате получился процессор, очень близкий по архитектуре с RTX 2000. Кристалл выполнен по КМОП-технологии 2,2 микрона и заключен в 100-выводной корпус PQFP.

Следующий контроллер Дофин-1630 (1997 г.) был уже совершенно оригинальной разработкой. Главной целью была поддержка DSP и языка C. Для этого была использована мультистековая архитектура. К фортовским стекам добавились второй стек данных и стек адресов. Введены дополнительные методы адресации через стек адресов и стек возвратов. К сожалению, поддержка Форты (по сравнению с Дофин-1620) пострадала.

В настоящее время компания производит микроконтроллеры TF16, активно использующиеся в отечественной встраиваемой технике, например, кассовых аппаратах.

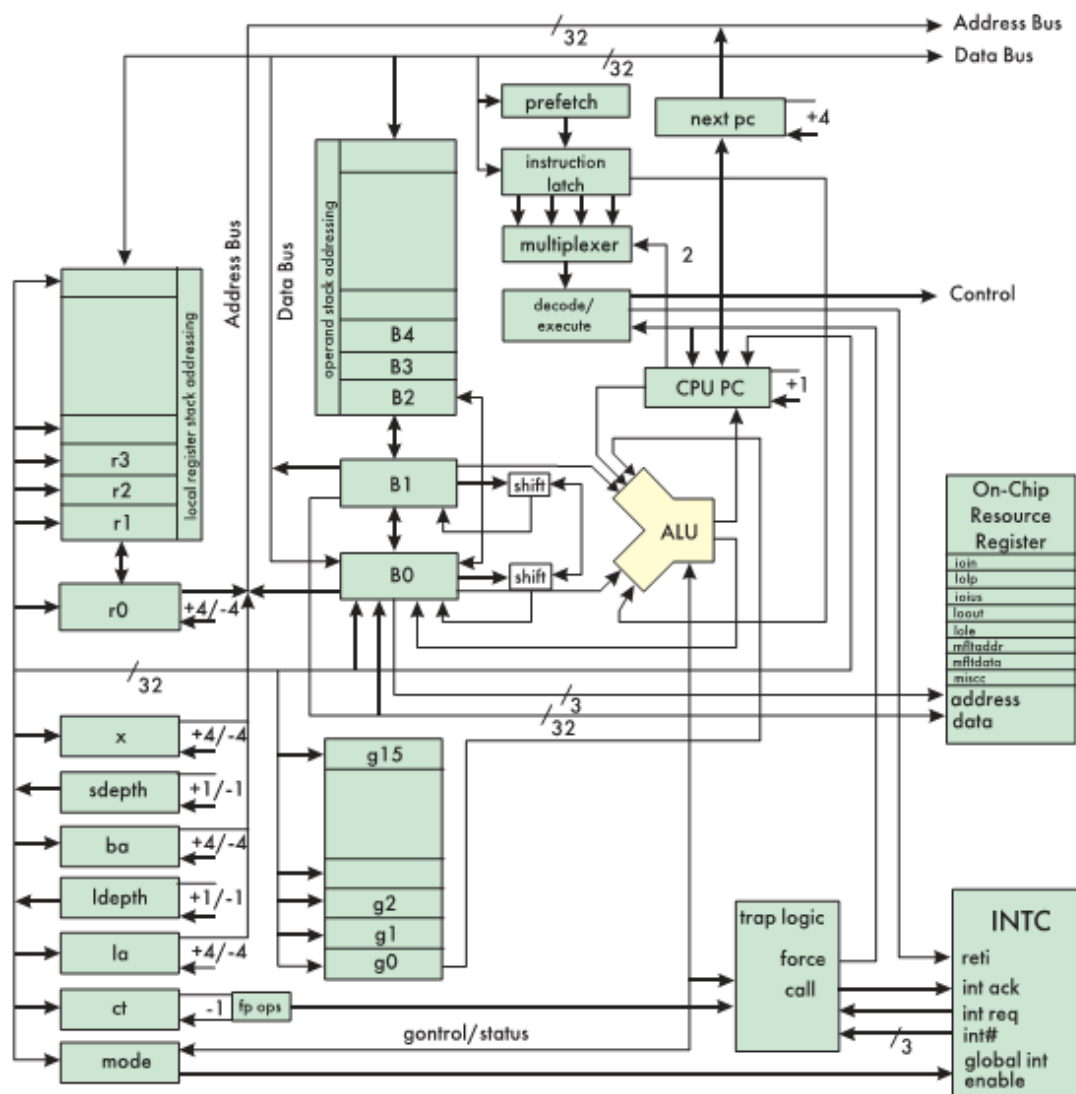


Знаменитые Forth-машины. Слева направо Novix Mostek MKGD02TG, Novix NC4016C (оба 1987 г.), Harris RTX2001G1-8 (1992 г.)

⁵² Учитывая, что это были клоны допотопного Intel 8086, подвиг невелик, скажем честно, но в те времена и это было большим достижением.

⁵³ А вот это уже подвиг!

Среди современных западных производителей также можно отметить Triangle Digital Services с их контроллером TDS9092 FORTH CHIPS, Patriot Scientific Corporation и их IGNITE. Последний уже не просто контроллер, а мощный 32-разрядный микропроцессор с очень высокой производительностью и плотностью упаковки команд, прекрасно исполняющий байт-коды Java, Forth и сгенерированные компиляторами языков C/C++ бинарные программы.



Блок-схема IGNITE

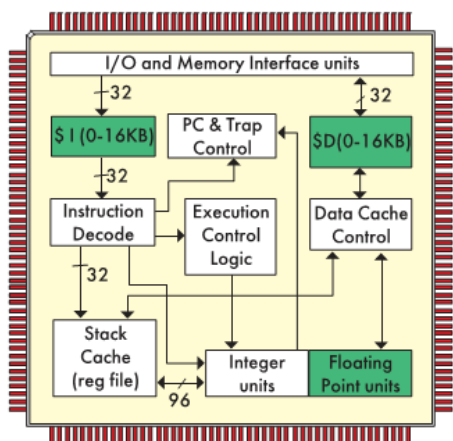
Операции этого микропроцессора занимают 8 бит, за один цикл чтения 32-битного слова выполняется 4 команды. Кроме того, применение стека также позволяет сократить число регистров, выполнять сохранение в стеке и извлечение данных из стека в пределах одной процедуры, позволяя таким образом, получить короткие последовательности команд и повысить быстродействие.

Еще одним неожиданным направлением, связанным с простотой разработки стековых машин, стало массовое участие в этом энтузиастов, использующих FPGA-матрицы. Это, по сути, заготовка процессора стоимостью, в среднем, от 50 до 300\$, которая позволяет с помощью специального программного обеспечения и программатора разработать и изготовить модель практически любой архитектуры. Этими экспериментами занимаются и серьезные фирмы, например, Altera выпускает чип Nios.

Среди любителей можно отметить разработки Бернда Пайсана (Bernd Paysan) b16, известный проект 4stack. Дальнейшей эволюцией стало использование других ЯВУ, в частности Java. Наиболее известен здесь стековый процессор с аппаратной поддержкой Java разработки Sun, который так и называется – picoJava. Он появился в начале 2000 г. и выполняет чистый байт-код Java, непосредственно заменяя собой Java Virtual Machine.

На процессорах, созданных по этой спецификации, код Java работает примерно в 20 раз быстрее, чем на стандартном процессоре с использованием JVM. Также эти процессоры способны исполнять код на C/C++

со скоростью, сопоставимой с процессорами архитектуры RISC. Такой подход позволяет существенно увеличить производительность программ на Java, не снижая при этом скорость выполнения программ на других языках. Спецификация pJava не описывает логику памяти и интерфейсов ввода-вывода, поэтому разработчики могут использовать логику по своему усмотрению. Sun Microsystems никогда не выпускала продуктов на основе этой технологии. Однако Sun лицензировала её некоторым компаниям, таким как Fujitsu, NEC и Siemens.



Ядро pJava-II

Примером применения технологии pJava-II может служить микропроцессор Fujitsu MB86799. Он состоит из ядра pJava-II, внешней интерфейсной шины и интерфейса шины PCI. Он имеет кэш команд 8 кбайт, кэш данных 8 кбайт, кэш стека с 64 входами и сопроцессор для операций с плавающей точкой. MB86799 может работать с внешней максимальной частотой 33 МГц и внутренней частотой 66 МГц. Соотношение внутренней и внешней частоты может быть от 2 до 5. Чип потребляет 360 мВт от источника питания 2,5 В при тактовой частоте 66 МГц.

Еще один известный проект - LavaCORE — 32-разрядное ядро Java-процессора, предназначенное для реконфигурируемых аппаратных средств и оптимизированное для применения в чипах Xilinx Virtex-II. Оно разработано фирмой Derivation Systems, Inc.⁵⁴ (), являющейся партнером фирмы Xilinx и может быть загружено с сайта Xilinx IP Center⁵⁵. Ядро процессора непосредственно выполняет байт-код Java аппаратным способом. LavaCORE выполнено как soft-ядро процессора с набором про-

граммных инструментальных средств для выполнения Java-приложения и конфигурирования процессора.

После pJava компанией ARM была разработана еще одна мощная Java-архитектура – Jazelle. Jazelle была изначально разработана для архитектуры ARMv5TE, и первым процессором, использующим ее, был ARM926EJ-S. Доступность технологии Jazelle в названии процессора обозначается буквой 'J', за исключением моделей, для которых данная технология является неотъемлемой частью архитектуры.

В настоящее время на рынке присутствует разновидность Jazelle DBX (Dynamic Bytecode eXecution — динамическое выполнение байт-кодов). Основным элементом этой технологии является специальный сопроцессор, встраиваемый в стандартную архитектуру микропроцессоров ARM9. Сопроцессор аппаратно преобразует байт-код Java в соответствующую последовательность команд основного процессора, который немедленно их выполняет. Отсюда и название DBX — «динамическое выполнение».

В июне 2005 г. ARM анонсировал новое поколение технологии Jazelle под названием RCT (Runtime Compiler Target – поддержка динамических компиляторов). Jazelle RCT вводит дополнительный набор инструкций в стандартную архитектуру процессоров ARM. Новые команды специально разработаны для оптимального выполнения Java байт-кода. Таким образом, простой компилятор может быстро преобразовать байт-код в команды основного процессора, при этом эффективно используя новые типы команд. В большинстве случаев каждому байт-коду будет соответствовать единственная 16-разрядная команда из нового набора. В результате сгенерированный код будет иметь практически тот же размер, что и исходный байт-код, но будет выполняться аппаратно и, следовательно, намного быстрее.

Особенностями технологии RCT могут воспользоваться, прежде всего, разработчики так называемых динамических компиляторов (Just-In-Time Compilers), поскольку преобразование байт-кода в набор команд RCT может быть выполнено очень быстро, и результирующий код не будет требовать излишне большой памяти для хранения. Набор команд RCT не привязан жестко к языку Java и может использоваться для компиляции байт-кодов прочих интерпретируемых языков, таких как Perl, Python, а также языков, поддерживаемых технологией Microsoft .NET.

Еще одной примечательной Forth-архитектурой является знаменитое семейство процессоров SEAFoth – это очень необычная и интересная архитектура, заслуживающая более детального рассмотрения. Отметим, что все Forth, Java и прочие описанные здесь машины, можно отнести формально и к теговым ЭВМ, т.е. таким, в которых ассемблером служит разновидность ЯВУ, но в этой главе мы делаем упор на их стековую архитектуру. К теговым машинам мы еще вернемся (стек для их реализации не обязателен, как, впрочем, и язык Forth для стека, просто они очень хорошо сочетаются), но будем понимать под ними немного более сложную архитектуру.

Наиболее экзотическим проектом стековых машин с ЯВУ, пожалуй, стал отечественный процессор «Кронос», использовавший в качестве внутреннего языка Модула-2. «Кронос» – семейство 32-разрядных процессоров, которые были предназначены для создания микро- и мини-ЭВМ. Архитектура процессоров «Кронос» ориентирована на поддержку языков высокого уровня ANSI C, Модула-2 и Оссам. Это семейство

⁵⁴ www.derivation.com

⁵⁵ www.xilinx.com/ipcenter

было разработано в новосибирском Вычислительном центре СО АН СССР в рамках проекта MAPC (Модульные Асинхронные Развиваемые Системы) во второй половине 1980-х гг. исследовательской группой KRONOS (Kronos research group) временного научно-технического коллектива (ВНТК) «СТАРТ» под руководством д-ра ф.-м. наук В. Е. Котова.

Впоследствии, в 1990 году на базе ВНТК «СТАРТ» был образован Институт систем информатики имени А. П. Ершова СО РАН. Процессоры «Кронос» выпускались опытными партиями в основном для целей построения инструментальных машин для создания и отладки ПО. Основное применение нашли процессоры «Кронос» 2.6 при мелкосерийном производстве рабочей станции Кронос-2.6WS. Первый образец этой рабочей станции был продемонстрирован на выставке «Наука-88» в Москве в 1988 году. Рабочие станции Кронос-2.6WS нашли применение как инструментальные ЭВМ на ряде предприятий оборонного комплекса СССР, в частности, использовались для создания бортового программного обеспечения для спутников в Научно-производственном объединении прикладной механики им. академика М. Ф. Решетнёва (НПО ПМ, Красноярск-26).

В настоящее время образцы рабочей станции Кронос-2.6WS сохранились в Музее науки в Лондоне (Великобритания), в Политехническом музее в Москве, в музее Сибирского отделения Российской Академии наук (г. Новосибирск), в Новосибирском государственном университете (НГУ) и в ИСИ им. А. П. Ершова СО РАН в Новосибирске.

ЭВМ с процессорами «Кронос» 2.2, «Кронос» 2.5 и «Кронос» 2.6 работают под управлением фирменной ОС Excelsior. Система команд процессоров «Кронос» базируется на системе команд компьютера Lilith, разработанного в Швейцарской высшей технической школе Цюриха — ЕТН проф. Никлаусом Виртом. Стековая архитектура процессоров, схожих с Lilith и «Кронос», в дальнейшем повлияла на систему команд и архитектуру виртуальной машины Java.

Взяв за основу работы швейцарской группы под руководством Никлауса Вирта, новосибирцы сделали 32-разрядный процессор, ориентированный на Modula-2, и использовали его впоследствии для реализации базовых компонентов транспьютерной архитектуры. Промежуточным результатом оказалось создание платы «Кронос» для «Электроники-60», в отличие от швейцарцев, и распространение, и серийное производство наладить было куда проще.

История «Кроноса» началась летом 1983 г., когда в Новосибирске появились материалы по Modula-2 и Lilith. В это время Алексей Недоря закончил третий курс механико-математического факультета НГУ и начал делать диплом — компилятор с языка Edison (придуманного Пером Бринч Хансенем, автором языка Concurrent Pascal). В то же время Дмитрий Кузнецов приступил к созданию компилятора Modula-2 (под руководством И. В. Поттосина). Схожие задачи и общая творческая атмосфера способствовали образованию неформального коллектива: вместе собрались несколько человек, которым было интересно программирование вообще и компиляторы в частности. Новосибирцы даже организовали свой семинар в общежитии, он назывался Intruders Club. Основным заводилой всей клубной деятельности стал Д. Кузнецов.

Весной 1984 г. начала формироваться более узкая группа; в этот момент, как вспоминает Алексей, «мы нахально решили сделать себе нечто похожее на Lilith: каждому хотелось заполучить домой собственную машину».

Работа была организована так: Кузнецов писал компилятор на Burroughs 6700 и интерпретатор «Кроноса», Недоря реализовывал маленькую ОС под управлением интерпретатора, а Тарасов с Васекиным делали процессор. Решение было принято самое простое — создать его в виде платы для «Электроники-60» (bit-slice). В июле 1984 г. Недоря написал на языке Паскаль микроассемблер для микропрограммирования «Кроноса» и к тому времени уже защитил диплом (компилятор Edison работал и порождал код для «Эльбруса», о котором подробно ниже).

В начале декабря 1984 г. процессор «Кронос-1» был готов. Для организации кросс-разработки по последовательному каналу «Кронос» был подключен к Burroughs, и начался мучительный процесс отладки. На «Кроносе» работала пультовая микропрограмма. На Burroughs была написана маленькая связанная утилита, которая позволяла загрузить программу в «Кронос» и запустить ее. Программы писались с самого начала на Modula-2.

Поскольку работал процессор кое-как, было принято решение изменить систему команд, процессор, компилятор и все остальное. Так началась разработка «Кроноса-2». Насколько успешны были их усилия, можно судить, например, по одному из применений этой машины.

В 1984–1989 гг. в Московском авиационном институте занимались созданием ПО для нескольких стендов технологического оборудования, предназначенного для научных экспериментов. Создание систем реального времени всегда отличалось особой сложностью, прежде всего из-за необходимости соединять несоединимое — эффективность и надежность. Подавляющее большинство работ выполнялось на ассемблере, при этом требовалась не просто реализация специфических драйверов устройств, а увязывание всех программных элементов в единую целостную систему, работающую при минимальном вмешательстве человека.

В 1985 г., когда команда Kronos Group уже перешла к созданию «Кроноса-2», разработчики решили использовать Modula-2. За пару месяцев удалось разыскать работающий компилятор (это были преемники виртовой системы для PDP-11: M22 и M23 для RT-11). Язык Modula-2, таким образом, стал своеобразным каркасом, в который, в случае необходимости, органично вписывались ассемблерные части.

Все бы ничего, даже не особенно страшны были и ограничения в 64 Кбайта для целевой машины (это была «Электроника-60» и другие отечественные разновидности DEC LSI-11). Однако производительность труда хотелось поднять, но как? Вот тут-то разработчики и узнали о существовании платы «Кронос». Добыть ее было непросто, но усилия окупились.

Весь комплекс в целом превосходил по удобству и быстродействию инструментария платформу PC XT/AT (при том, что XT уступала на тестах по обработке и переключению контекстов процессов даже отечественным вариантам LSI-11). Кроносковский диспетчер для DOS позволял запускать в 640 Кбайт памяти более 1000 асинхронных процессов, часть которых могла выгружаться на диск.

«Кронос-2» сильно помог, в частности, при создании в МАИ вполне конкретной встроенной системы для управления установкой плазменного напыления, точнее, осуществляющей активный контроль за процессами плазменной и магнитронной обработки поверхностей различных материалов. Этот комплекс уникального оборудования, созданный руками инженеров МАИ, позволял вести работы по изучению свойств новых перспективных покрытий для аэрокосмической индустрии. Помимо привычных для систем реального времени функций, он обеспечивал и поддержку БД режимов технологических процессов.

Отдача от архитектуры была солидной, несмотря на малое количество плат (а потом и отдельных станций), за все время было выпущено не менее 200 плат «Кронос-2». К 1987 г. был налажен выпуск модификаций «Кронос-2.5» (две платы, которые вставлялись в Labtam, шины Multibus-I/II) и «Кронос-2.6» (отдельная рабочая станция на микропроцессорных секциях серии 1802, шины Q-bus, Q-22, И-41, Multibus-I/II, 8 Мбайт ОЗУ, винчестер 40 Мбайт, растровый дисплей 480 x 360 точек, 16 цветов). При одинаковой частоте (3 МГц) производительность «Кроноса-2.5» составляла 1 млн команд/с, «Кроноса-2.6» – 1,5 млн команд/с.

Выпуском занялись СКВ Института кибернетики АН Эстонии, радиозавод в Ижевске, завод в Ленинграде, несколько предприятий в Новосибирске, Бердске, Кишиневе, Перми. Модернизировались также финские 16-разрядные компьютеры серии ELORG-DATA. Киевское НПО «Микропроцессор» приступило к разработке топологии и подготовке производства процессора «Кронос» в однокристалльном варианте. Создавался и 64-разрядный арифметический сопроцессор. Были изготовлены лабораторные образцы кристалла, однако из-за низкого качества оборудования процент брака был слишком велик, и производство так и не было налажено.

Что касается программного обеспечения, то, помимо Modula-2, были реализованы компиляторы для языков Fortran-77 и Си (а также менее известных НУТ, БАРС, ПОЛЯР), разные графические пакеты (например, bCAD), САПР печатных плат и многое другое. В частности, САПР печатных плат, заработавшая на «Кроносе-2.5», позволила заметно ускорить разработку «Кроноса-2.6», а идеи пакета bCAD оказались столь удачными, что новосибирская фирма ProPro Group создала его новое поколение (с поддержкой OpenGL, механизмов морфинга и сетевого фотореалистичного тонирования, с использованием анимированных текстур и органичного моделирования, основанного на В-сплайнах и NURBS-технологии). Со временем в Kronos Research Group был реализован и программный эмулятор для платформы Intel x86.

Предметом особой гордости стала ОС Excelsior. При ее проектировании у разработчиков возникло желание упростить обучение пользователей, а потому за прототип была взята популярная тогда ОС UNIX. Окончательная версия, полностью написанная на Modula-2, получила имя Excelsior iV. Система была многопользовательская, многозадачная, построенная на принципе динамической загрузки (каждый модуль – аналог нынешних DLL-библиотек). Ее архитектуру разработали Дмитрий Кузнецов (файловая система) и Алексей Недоря (ядро). В общей сложности ОС Excelsior создавали около 15 человек.

Следующим амбициозным проектом стал MAPC. Исследовательский проект MAPC (Модульные Асинхронные Развиваемые Системы) являлся одним из ответных шагов Советского Союза на амбициозный японский проект ЭВМ пятого поколения. Он проводился в три этапа. Вначале в ВЦ СО АН СССР под руководством В. Е. Котова и А. Г. Марчука были сформулированы общие положения концепции MAPC. На втором этапе (1985–1988 гг.) был образован ВНТК «Старт», объединивший научные и инженерные группы из ВЦ АН СССР, СКВ вычислительной техники СО АН СССР, ИК АН ЭССР, НПО «Импульс» и др. Особенно здесь выделялись два коллектива – новосибирский, формирующий базовый (системный) уровень MAPC, и эстонский, отвечавший за интеллектуализацию (средства представления знаний). Эстонцы реализовали на «Кроносе» систему объектно-ориентированного программирования, обеспечивающую в рамках языка НУТ работу с классами, объектами и продукционными правилами. Язык НУТ разрабатывался с прицелом на решение задач искусственного интеллекта. На третьем этапе проекта MAPC (1988–1990 гг.) планировалось построить экспериментальный образец системы на новой элементной базе.

С точки зрения аппаратуры, фактически были инициированы работы, которые велись параллельно с развитием линии «Эльбрус», но уже не на базе суперкомпьютеров, а на уровне кластеризации процессоров среднего класса. В основу было положено мультипроцессорное ядро на базе процессоров «Кронос» с использованием транспьютерной организации. При этом транспьютерные узлы, обладая собственной локальной памятью, соединялись между собой асинхронными каналами и на основе системной шины комплектовались в транспьютерные модули. Это позволяло достигать гибкой масштабируемости и смещать акцент на аппаратную поддержку распределенных потоковых вычислений.

В отличие от английской концепции транспьютера, разработанной в компании Inmos, в случае MAPC вычислительный элемент имел не четыре, а произвольное число соседей. Основным строительным блоком

служил процессор «Кронос», точнее, МАРС-Т. Цель проекта МАРС-Т сводилась к тому, чтобы построить сеть процессоров, связанных быстрыми каналами. Память в системе была только локальная. Скорость передачи по каналам довольно высокая – порядка 2 Мслов, что примерно в два раза хуже, чем у транспьютера Inmos T800. Толчок идеям дал язык Оссат, но в случае МАРС все было сделано по-другому.

Макет МАРС-Т состоял из четырех процессоров «Кронос-2.6», связанных каналами. В процессоры были добавлены дополнительные команды для приема/передачи данных из каналов. На одну машину (головную) загружалась ОС Excelsior iV с модифицированным ядром, на все остальные – небольшое специализированное ядро. Подсистема ввода/вывода работала только на головном процессоре. Все остальные могли лишь передавать данные по каналам. Программы передавались тоже по каналам. В ядро входила система поддержки виртуальных каналов. Любую программу можно было запустить на одном процессоре (при этом использовались виртуальные каналы) или на нескольких. Правда, система автоматического распределения нагрузки так и не была доведена до конца. Для демонстрации возможностей МАРС-Т команда Kronos Group написала несколько тестовых примеров. Один пример был графический. На экране вращалась надпись, причем один или несколько процессоров ее обсчитывали, а головной отрисовывал. К сожалению, приемная комиссия лишь благосклонно посмотрела демонстрации возможностей архитектуры, на том все и закончилось.

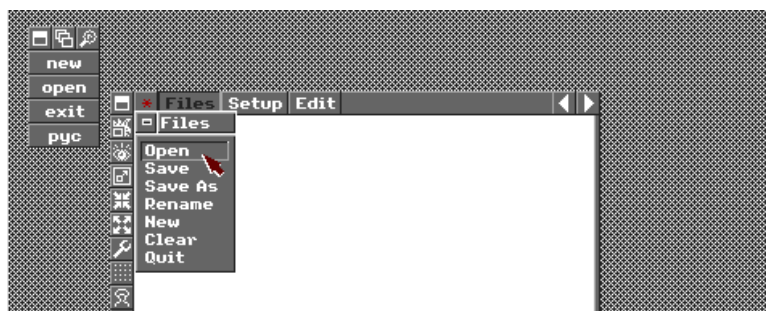
После завершения проекта МАРС группа специалистов Института систем информатики СО РАН, которую возглавлял И. В. Поттосин и куда входили некоторые члены Kronos Research Group, занялась реализацией другого проекта – СОКРАТ. Он предусматривал создание средств кросс-разработки (в частности, для языка Modula-2), которые были предназначены для бортового ПО российских спутников. В течение 1988–1991 гг. в рамках этого проекта закладывались основы новой технологии генерации объектного кода, которая была использована для создания в 1991–1994 гг. целой серии компиляторов для языков Modula-2 и Oberon-2. Тогда же из членов Kronos Group была образована первая компания – xTech Ltd.



Процессор «Кронос-2.5» и «Кронос-2.6» – АЛУ, ОЗУ и БМУ



Рабочая станция «Кронос-2.6»



ОС Excelsior

В 1994 г. была спроектирована инструментальная многоплатформная среда кросс-разработки, получившая название XDS (extensible Development System) и давшая впоследствии имя новой фирме. Характерным отличием ее было наличие переносимой исполняющей системы (RTS), реализующей для Oberon-2 эффективный механизм метапрограммирования (трактовка данных как кода) и сборки мусора. Участие новосибирцев в работе международных комитетов по стандартизации Modula-2 (ISO) и Oberon-2 оказало большую помощь в получении качественных реализаций соответствующих компиляторов. Компиляторы серии XDS стали главным тиражируемым продуктом компании.

Как не жаль, развитие линии PC (в отличие от «Кроноса») пошло такими темпами, что многие прелести отечественного чуда на их фоне постепенно исчезли. Тем не менее, стековые машины до сих пор живее всех живых и чувствуют себя отлично. Простота разработки, дешевизна, возможность использовать различные языки – вот основные причины, которые позволяют им оставаться востребованными уже много лет.

Стековая машина в лоб реализуется как регистровая с большим регистровым пулом и регистром – указатель номера регистра (из пула) – текущей вершины стека. Разница будет только в том, что в обычной регистровой машине регистры доступны непосредственно, потому могут использоваться как кеш данных. Такое положение дел у регистровой машины тормозит переключение тредов исполнения, поскольку приходится спасать весь пул регистров. В стековой по регистру StackTop всегда видно, сколько регистров занято, и спасаются только занятые, на стековой переключение тредов реализуется куда проще.

При большом количестве регистров возможна хорошая оптимизация исполнения кода, как на этапе компиляции, так и на этапе исполнения (параллельное исполнение нескольких инструкций). Но каждый регистр – это его адрес в инструкции процессора. Если регистров 8, то на каждый регистр надо 3 бита, и на три регистра надо 9 бит. Если регистров 16, то на три регистра надо 12 бит и т.д. А ещё нужны биты для задания операции с регистрами, инструкции процессора увеличиваются и не вмещаются в память. Как с этим борются популярные регистровые машины?

- MIPS – простые инструкции, много регистров. Много вырожденных команд, скажем, чтобы взять элемент массива 32-битных целых чисел, надо отдельной командой сдвинуть регистр индекса, и отдельной командой загрузить значение из памяти;
- ARM – более сложные инструкции с дополнительными операциями для одного из регистров. Скажем, можно во время операции сдвинуть регистр индекса и взять значение из 32-битного массива целых за одну команду. Меньше регистров (номера регистров банально не помещаются в инструкцию процессора);
- ARM Thumb – 16-битные команды, за счёт упрощения системы команд, уменьшения числа регистров, 2-х регистровых команд – зато команды занимают меньше места в памяти и грузятся быстрее.

Есть и другие способы борьбы (специализированные регистры, плавающий размер инструкций процессора и пр.). Можно добавить совсем много регистров, увеличивать размер инструкции (особенно, когда не надо память экономить) – но начинаются проблемы с переключением контекстов, размером файлов регистров для всех конвейеров и пр., а сильного увеличения производительности уже нет. Все они имеют свои недостатки и достоинства, но однозначно лучшей среди них нет.

Стековые архитектуры стоят особняком – у них другой выбор. Они жертвуют возможностью оптимизации (и выполняются значительно медленнее), но выигрывают в размере команд. По одному байту на команду. Регистры вообще не надо адресовать.

В результате, для регистровых архитектур имеем сложный дизайн процессора, большой размер кода — но быстрое исполнение команд. Для стековых архитектур – простой дизайн процессора и небольшой размер кода, но медленное исполнение команд. Увеличение сложности процессора позволяет ускорить исполнение кода и там и там, но максимальная скорость регистровых архитектур для стековых недостижима.

Что лучше? Зависит от задачи и прочих условий. Сейчас мы имеем ситуацию, когда скорость процессоров растёт медленно, а количество транзисторов растёт быстро. И плюс – большинство приложений однопоточны. Очевидно, что для данных условий регистровая модель лучше. Можно сделать достаточно сложный процессор, и он будет быстро исполнять наши потоки. Для массового параллелизма лучше выглядит стековая модель. Можно сделать много очень простых ядер, и хотя каждое будет работать медленнее, чем регистровое ядро – но в сумме они бы работали быстрее.

3.3. Ассемблер

Изучение ассемблера – дело и простое и сложное одновременно. Простое, так как объективно – это один из самых примитивных языков, поэтому формально ознакомиться с ним можно легко. Тем не менее, он считается весьма трудным по той причине, что успешно писать на нем сложные программы можно, лишь хорошо разбираясь в самых низкоуровневых вещах – базовой архитектуре процессора, что в настоящее время встречается нечасто.

Проблема же в том, что для глубокого понимания архитектуры ЭВМ ассемблер надо бы, в свою очередь, уже неплохо знать. В результате мы оказываемся в своеобразном методическом тупике – изучить одно без другого очень сложно, и не знаешь, с чего начать. Потому эта небольшая глава посвящена не глубоким тайнам машинных языков, а скорее, носит характер ознакомления с их принципами, облегчающими дальнейшее изучение структуры компьютера. Тем более, что, как и любой приличный язык, ассемблер целиком описывается не в одной книге, многие из которых весьма объемны и нет смысла переписывать их здесь.

Типичная команда компьютера хранится в слове памяти, иногда в нескольких словах. Она содержит код команды, определяющий ее тип – операцию, выполняемую командой, а также ноль или более требуемых аргументов, которые могут быть адресами или значениями. Для примера рассмотрим команду компьютера с архитектурой процессора IBM Power PC, длина слова – 32 бита. Команда занимает целое слово с нумерацией битов, начинающейся с нуля:

0	6	11	16	21	22	31	Позиция бита
0 1 1 1 1 1	0 0 1 0 1	0 0 0 1 1	0 0 1 0 0	0 0 1	0 0 0 0 1 0 1 0		Значение бита
Код команды	Результат	Операнд 1	Операнд 2		Вторичный код (266)		Роль

Код команды является комбинацией первичного кода 31 (двоичное 11111), заданного битами от 0 до 5, и вторичного кода 266 (двоичное 10001010) в битах от 22 до 31. Результат помещается в регистр 5 (двоичный код 101), а операнды читаются из регистров 3 и 4.

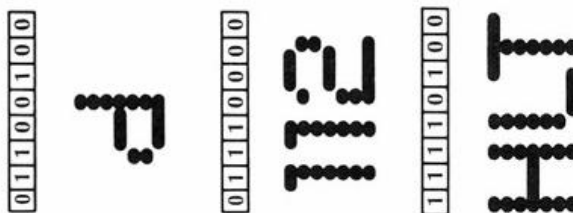
Побитовое представление неудобно для практического использования человеком. Обычная нотация заменяет двоичное представление более коротким – шестнадцатеричным. Выше представленное слово 01111100101000110010000100001010 равно 7CA3210A в шестнадцатеричной системе, это и есть полное описание команды, которую мы хотим выполнить⁵⁶.

Компьютеры, как правило, обладают командами трех разных типов.

- команды передачи данных - загрузка значений из оперативной памяти в регистры процессора, запись данных из регистров в память, перемещение данных между регистрами;
- команды обработки данных - арифметические и логические операции над данными, содержащимися в регистрах процессора;
- команды передачи управления - условные и безусловные переходы, вызовы подпрограмм и возвраты из них, эти команды явно изменяют значение в регистре команды.

Каждая команда имеет свой код, такой, как 31 в примере. Естественно, писать прямо в бинарных кодах, даже и в шестнадцатеричных, очень неудобно. Первые компьютеры вообще не имели общепринятых устройств ввода, типа ленты или даже перфокарт, и программу приходилось устанавливать переключателями на передней панели вычислительного устройства. Очевидно таким способом можно было составить только небольшие программы.

После того как появились перфокарты, бинарный, десятичный или шестнадцатеричный код стало возможным пробивать на бумаге, примерно так:



Первым проблему решил Тьюринг, изобретая сокращенную запись – в этом коде для представления длинных строк двоичных нулей и единиц использовались комбинации телетайпных символов, которые затем интерпретировались и превращались в двоичный код специальным устройством – то, что сейчас бы мы назвали компилятор.

В 1949 г. отец ENIAC Джон Мочли, независимо от Тьюринга, пришел к своей мнемонической системе - Short Code. Первая реализация была написана Вильгельмом Шмидтом для компьютера BINAC в том же году, на следующий год Шмидтом была разработана новая версия для UNIVAC I. Конечный вариант Short Code был предложен в 1952 году для компьютера Univac II А.Б. Тоником и Дж. Р. Логаном.

Сейчас сложно назвать это языком, но в те годы это был прорыв. Например, сложение вида $A = B + C$ уже записывалось не длинными рядами 0 и 1, а как 00 S0 03 S1 07 S2, где переменные A, B, C обозначены S0, S1, S2, а операции равенства и сложения - 03 и 07 (комбинация 00 определяет номер строки программы). Все операции по преобразованию в код производились руками, в машинные инструкции он транслировался автоматически.

Предложенная Мочли система кодирования, несмотря на все ее неоспоримые преимущества по сравнению с программированием в двоичном машинном коде, отнюдь не была панацеей:

- поскольку язык этой системы был весьма далек от машинного, для его реализации требовалась некая программа-посредник, которая осуществляла бы перевод программ в двоичный код компьютера;

⁵⁶ Для Power PC мнемоника команды сложения с кодом 31 – add. На языке ассемблера Power PC рассмотренная нами команда может быть записана так: add r5, r3, r4

- программу, записанную в системе кодирования Мочли, надо было просматривать и выполнять строка за строкой. Это делалось при помощи интерпретатора. Таким образом, каждая отдельная операция программы перед ее выполнением должна была подвергаться полному посимвольному разбору, даже если до этого она уже не раз использовалась;

- при переносе подпрограммы в другую программу приходилось менять все числа, задающие адреса конкретного месторасположения программы в памяти машины. Обычно эти адреса образовывали группу последовательных номеров. Поэтому для внесения подпрограммы в другую программу требовалось прибавить к каждому адресу подпрограммы некоторую константу.

Результатом дальнейших изысканий стало появление уже полноценного языка ассемблера, и приложила к этому руки математик и программист Грейс Хоппер (Grace Murray Hopper). С детства увлекавшаяся техникой, она закончила Гарвард и была направлена на работу в отдел, занимающийся обслуживанием Harvard Mk I – одного из первых компьютеров мира.

Ввиду того что основной задачей Mk I было вычисление артиллерийских баллистических таблиц, в нем предусматривались многие математические операции общего назначения. Однако Грейс Хоппер и ее коллеги убедились, что для решения конкретных задач требуются более специфические средства. Она предложила запоминать многократно используемые последовательности команд, построенные таким образом, чтобы из них можно было компоновать крупные сегменты, а затем собирать из них более сложные программы⁵⁷.

В октябре 1951 г. Грейс Хоппер получила задание создать набор стандартных математических подпрограмм для UNIVAC I. В процессе работы она окончательно пришла к мысли, что подпрограммы и Short Code вещь хорошая, но работа с ними руками – напрасная трата сил и времени. Так у нее родилась идея совместить сборщик подпрограмм, транслятор и мнемоническую запись кодов в одно целое – и первый ассемблер A-0 был рожден.

Данный тип языков получил своё название от транслятора (компилятора) с этих языков – ассемблера (assembler – сборщик). Название обусловлено тем, что программа «автоматически собиралась», а не вводилась вручную покомандно непосредственно в кодах. При этом наблюдается путаница терминов: ассемблером нередко называют не только транслятор, но и соответствующий язык программирования.

В результате сейчас этот термин, как правило, употребляют в трех смыслах: собственно программа-ассемблер, преобразующая все, что мы написали в машинный код, язык команд конкретной машины и класс низкоуровневых языков в целом, безотносительно платформ.

Проиллюстрируем разницу между машинным кодом и ассемблером следующим примером. Рассмотрим древний настолько, что уже стал классическим, процессор Intel 8086. Все современные процессоры семейства x86, что очевидно, имеют систему команд, основанную именно на нем и принципиально не отличающуюся.

Система команд 8086 достаточно обширна. Команда может иметь длину от одного до шести байт. Первый байт команды всегда содержит код операции и может включать поля признаков. Коды некоторых команд не умещаются в один байт, оставшиеся биты записываются во второй байт команды. Второй байт обычно содержит поля, указывающие способ формирования адреса операнда (если он не определяется первым байтом). Следующие байты, если они есть, содержат адреса и данные. Вот как выглядит программирование непосредственно в бинарных кодах процессора.

Для того чтобы загрузить значение в регистр, вы должны подать на вход устройства управления такую комбинацию:

1011 W DST LB [HB]

Для лучшего осознания мы пока используем человеческие сокращения – бит W указывает, что должно быть загружено, байт (0) или слово (1), три бита DST определяют целевой регистр по следующим правилам:

DST			DST		
000	AX	AL	100	SP	AH
001	CX	CL	101	BP	CH
010	DX	DL	110	SI	DH
011	BX	BL	111	DI	BH

Итак, мы хотим, например, положить число 158 в регистр AX. В двоичной системе оно равно 10011110, что дает ровно байт. Соответственно, мы должны послать устройству управления процессора последовательность

1011000010011110

это и есть искомая бинарная команда. Разбив ее на части, для лучшего понимания, легко выделим:

1011	0	000	10011110
код операции	признак значения	адрес	значение

⁵⁷ Она же подарила миру термин «баг», в значении «ошибка программы». В один из дней Mk I остановил вычисления, т.к. залетевший внутрь мотыль замкнул одно из многочисленных реле. Мотыль был извлечен, машина почищена, а на вопрос контролирующего офицера, что они там копаются, Хоппер пошутила – жуков вычищаем (debugging), так родился классический термин «отладка».

Несложно? Вот так и программировались первые ЭВМ – сначала программу писали на бумаге, потом скрупулезно руками переносили в двоичные коды, набивали их на перфокарте и посылали в процессор размером со шкаф.

Вот еще несколько команд 8086:

1000101 W 00 REG PTR – загрузка значения из памяти в регистр, W снова признак, три бита REG указывают регистр аналогично DST, три бита PTR указывают тип адресации, например PTR=111 указывают, что для получения адреса надо сложить содержимое регистров DS и BX.

0000001 W 11 DST SRC – сложение значений в двух регистрах, синтаксис аналогичен вышеописанным.

0010001 W 11 DST SRC - операция И.

11001101 N и 11001111 – вызов подпрограммы обработки прерываний (достаточно сложный вызов – для вычисления прерывания процессор загружает в стек PSW, CS и IP, затем загружает в CS:IP значения из ячеек с 0000:04000·N по 0000:04000·N + 3) и выход из прерывания.

Надеемся, этого достаточно, чтобы понять, каким спасением оказалась мнемоника команд, вместе с интерпретатором их понимающим. До этого пробовали писать команды в шестнадцатеричной системе, так было короче, но не проще. Вот, например, «Hello, world!» в шестнадцатеричных машинных кодах 8086 и он же, на языке ассемблера:

Hex-адрес	Hex-код	Метка	Ассемблер
0100	1E		push DS
0101	B4 09		mov AH,09
0103	BA 10 01		mov DX, offset @H
0106	0E		push CS
0107	1F		pop DS
0108	CD 21		int 21H
010A	1F		pop DS
010B	B8 00 4C		mov AX,4C00H
010E	CD 21		int 21H
0110	48	@H:	Db “Hello, world!\$”
0111	65		
0112	6C		
0113	6C		
0114	6F		
0115	2C		
0116	20		
0117	77		
0118	6F		
0119	72		
011A	6C		
011B	64		
011C	21		
011D	24		

Как видим, на несколько десятков шестнадцатеричных чисел машинного года приходится несколько строчек ассемблера, это при том, что сама строка Hello, world! составляет почти половину программы. Все что нужно сделать - это поместить в регистр AH номер функции DOS (9), адрес строки поместить в DS:DX и вызвать 21-е прерывание (символ доллара является концом строки).

Заметим, что эта программа написана под DOS, аналогичная под Win32 будет весить порядка 4 Кб и набрать ее руками уже весьма затруднительно. Таким образом, очевидно, что если бы не Мочли и Хоппер, то программистам пришлось бы очень туго.

Современный ассемблер основывается на следующих принципах:

1. Команды языка ассемблера один к одному соответствуют командам процессора. Фактически, они и представляют собой более удобную для человека символьную форму записи — мнемокоды — команд и их аргументов. При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процессора, более того, современные процессоры уже очень давно не выполняют команды буквально. Например, регистров там много, а адресуемых среди них в разы меньше, таким образом, запись на ассемблере становится ненамного менее условной, чем на любом другом языке.

2. Ассемблер позволяет использовать символические метки вместо адресов ячеек памяти, которые при ассемблировании заменяются на вычисляемые ассемблером или компоновщиком абсолютные или относительные адреса, без участия программиста.

3. Ассемблер позволяет использовать называемые директивы препроцессора (команды ассемблера, не переводимые в машинные коды, а выполняемые самим ассемблером).

Каждая модель (или семейство) процессоров имеет свою систему команд и соответствующий ему язык ассемблера. Из всего же многообразия синтаксисов этого языка сейчас остались только два — Unix-программисты предпочитают по традиции запись в стиле AT&T, Windows-программисты, соответственно, нотацию Intel.

Существуют компьютеры, реализующие в качестве машинного языка программирования высокого уровня (Forth, Lisp, Эль-76). Фактически, в таких компьютерах они выполняют роль языков ассемблера. Частично об этом мы уже говорили, ниже продолжим это обсуждение теми же машинами.

Достоинств у ассемблера много:

- Язык ассемблера, теоретически, позволяет писать самый быстрый и компактный код, какой вообще возможен для данного процессора. Дело в том, что если код программы достаточно большой, — данные, которыми он оперирует, не помещаются целиком в регистрах процессора, то есть частично или полностью находятся в оперативной памяти, то искусный программист, как правило, способен значительно оптимизировать программу по сравнению с ЯВУ. Скорость же достигается за счёт оптимизации вычислительного алгоритма и/или более рационального обращения к ОЗУ. Объём кода минимизируется за счёт тонких приемов оптимизации, на которые автоматический транслятор не способен. Пример: IM-клиент Faim 0.21 — 12.61 Кб, ОС Menuet целиком на ассемблере занимает 1 дискету (!), при этом включая: графическую оболочку, интегрированную среду разработки, TCP/IP, FTP/HTTP/SMTP — сервер, поддержка нескольких языков и офисный пакет.

- Обеспечение максимального использования специфических возможностей конкретной платформы, что также позволяет создавать более эффективные программы, — в том числе менее ресурсоемкие.

- При программировании на языке ассемблера возможен непосредственный доступ к аппаратуре, и, в частности, портам ввода-вывода, регистрам процессора и др. Потому язык ассемблера часто применяется для создания драйверов оборудования и ядра операционной системы (или машинозависимых подсистем ядра ОС) или написания BIOS. Также с его помощью часто создаются машинозависимые подпрограммы компиляторов и интерпретаторы языков высокого уровня, а также реализуется совместимость платформ.

- С помощью дизассемблера возможно исследовать существующие программы при отсутствии исходного кода.

- Он позволяет использовать новые команды, доступные в новых процессорах. Компилятор хоть и оптимизирует код высокого уровня при компиляции, но практически не способен генерировать инструкции из расширенных наборов команд типа AVX, CTX, XOR и т.п., так как команды в процессоры, обычно, добавляются быстрее, чем в компиляторах появляется логика для генерации этих команд.

- Используется для написания кода, создание которого невозможно (или затруднено) на языках высокого уровня, например, получение дампа памяти/стека. Даже когда аналог на языке высокого уровня возможен, выгода от языка ассемблера может быть значительной. Например, реализация подсчета среднего суммы двух чисел с учётом переполнения для x86 процессоров занимает всего 2 команды (сложение с выставлением флага переноса и сдвиг с займом этого флага). Аналог на языке высокого уровня может не работать в принципе или при компиляции конвертироваться в намного большее число команд.

- Полезен в изучении структуры ЭВМ.

- Используется для написания инфекторов, вирусов, руткитов и т.п.

Конечно, у ассемблера есть недостатки:

- Сложную программу написать на языке ассемблера намного сложнее — вплоть до полной невозможности или это потребует многих лет разработки (более 12 лет в случае MenuetOS).

- В силу машинной ориентации человеку сложнее читать и понимать программу на нём по сравнению с программой на языке высокого уровня.

- Подлежащий редактированию ассемблерный код сравнительно большой — программа на языке ассемблера состоит из слишком мелких элементов. Соответственно, в разы усложняются программирование и отладка, растёт трудоёмкость разработки, сравнительно велика вероятность программных ошибок.

- Для наилучшего результата требуется высокая квалификация программиста. Код на ассемблере выполняется быстрее, но написанный неопытным программистом обычно оказывается намного хуже сгенерированного компилятором с языка высокого уровня.

- Программы на языке ассемблера не переносимы на компьютеры с другой архитектурой и системой команд.

- В результате, на языке ассемблера пишут подпрограммы (в настоящее время довольно редко программы пишутся на ассемблере целиком), для которых критически важны:

- быстродействие (драйверы, отдельные элементы программ для GPU, самые затратные части видео- и аудиокодеков);

- объём используемой памяти (загрузочные секторы, встраиваемое ПО, прошивки для микроконтроллеров и процессоров с ограниченными ресурсами, вирусы, программные защиты).

Типичными командами языка ассемблера являются (большинство примеров даны для Intel-синтаксиса архитектуры x86):

- Команды пересылки данных (mov и др.);
- Арифметические команды (add, sub, imul и др.);
- Логические и побитовые операции (or, and, xor, shr и др.);
- Команды управления ходом выполнения программы (jmp, loop, ret и др.);
- Команды вызова прерываний (иногда относят к командам управления): int, into;
- Команды ввода/вывода в порты (in, out).

Для микроконтроллеров и микрокомпьютеров характерны также команды, выполняющие проверку и переход по условию, например:

- cbne — перейти, если не равно,
- dbnz — декрементировать, и если результат ненулевой, то перейти,
- cfsneq — сравнить, и если не равно, пропустить следующую команду.

Команды обычно имеют следующую структуру: [метка:] опкод [операнды] [;комментарий].

Вот основные отличия AT&T-ассемблера (GAS) от Intel-ассемблера (MASM, TASM, FASM, NASM).

AT&T:

- Директивы начинаются с точки.
- Многострочные комментарии обозначаются как в Си (/* ... */).
- Регистры обозначаются префиксом % и регистр-приёмник указывается после источника; например код `movl $42, %eax` помещает значение 42 в регистр `eax`.
- Отсутствие префикса операнда указывает на адрес в памяти; поэтому `movl $F,%eax` помещает адрес переменной `F` в регистр `%eax`, а `movl F, %eax` помещает в `%eax` содержимое переменной `F`.
- Порядок операндов — вначале источник, затем приёмник `movl %ebx,%eax`.
- Числовые константы имеют следующую форму записи `$0x20` (20).
- Для записи/считывания значения из определённого адреса в регистр знак доллара отсутствует `movl 0xffff,%eax`.
- Указания на индексные методы адресации отличаются тем, что используются круглые, а не квадратные скобки: `subl -0x20(%ebx,%ecx,0x4),%eax`.
- Размер операнда определяется как суффикс имени инструкции. Суффиксы:
 - b (от byte) — операнды размером в 1 байт,
 - w (от word) — операнды размером в 1 слово (2 байта),
 - l (от long) — операнды размером в 4 байта,
 - q (от quad) — операнды размером в 8 байт,
 - t (от ten) — операнды размером в 10 байт,
 - o (от octo) — операнды размером в 16 байт.

Ассемблер от Intel:

- Название регистров зарезервировано (нельзя использовать метки с именами `eax`, `ebx` и т. д.).
- Порядок операндов — вначале приёмник, затем источник.
- Числовые константы имеют следующую форму записи `20h`.
- Указания на индексные методы адресации: `sub eax,[ebx+ecx*4h-20h]`.

Программа на языке ассемблера может содержать директивы препроцессора: инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой компилятора. Набор и синтаксис их значительно разнятся и зависят не от аппаратной платформы, а от используемого транслятора (порождая диалекты языков в пределах одного семейства архитектур). В качестве «джентльменского набора» директив можно выделить следующие:

- определение данных (констант и переменных),
- управление организацией программы в памяти и параметрами выходного файла,
- задание режима работы компилятора,
- всевозможные абстракции (то есть элементы языков высокого уровня) — от оформления процедур и функций (для упрощения реализации парадигмы процедурного программирования) до условных конструкций и циклов (для парадигмы структурного программирования),
- макросы.

Сравнивая трансляторы между собой, стоит обратить внимание на ряд параметров. Начнем с генерации отладочной информации, без которой отладка программы сложнее, чем «hello, world», превращается в настоящую пытку. Несмотря на то, что мнемоники машинных команд, что в листинге, что в отладчике — одни и те же, работать с ними трудно, не имея меток, структуры, имен функций и т.п. Без них и так не самый простой код станет совершенно нечитаемым.

Проблема в том, что формат отладочной информации не стандартизован, и различные трансляторы используют различные форматы, что ограничивает нас в выборе отладчиков или вынуждает использовать конвертеры сторонних производителей, а некоторые ассемблеры (например, FASM) не генерируют отладочной информации вообще. Но если формат отладочной информации – это задний двор транслятора, то формат выходных файлов – это его лицо. Вроде бы, снова, какой там формат? Обыкновенный obj, из которого с помощью линкера можно изготовить все, что угодно – от exe до dll. На самом деле обыкновенных объектных файлов не бывает. Есть omf (в редакциях от Microsoft и IBM), coff, elf, aout и куча разной экзотики в стиле as86, rdf, ieeе и т.д. Также заслуживает внимания возможность сквозной генерации двоичных файлов, не требующая помощи со стороны линкера. А некоторые ассемблеры (например, FASM) даже позволяют вручную генерировать исполняемые файлы и динамические библиотеки различных форматов, полностью контролируя процесс их создания и заполняя ключевые поля по своему усмотрению.

Впрочем, обычно на ассемблере пишутся лишь системно-зависимые компоненты или модули, критичные к быстродействию, которые затем линкуются к основному проекту и, если ассемблер генерирует только omf, а компилятор – coff, то возникает проблема сборки разных форматов воедино. Так что все-таки лучше, чтобы и ассемблер, и компилятор генерировали одинаковые форматы объектных файлов.

Другой немаловажный критерий – количество поддерживаемых процессорных архитектур, которых в линейке x86 уже больше десятка. На самом деле ни один из трансляторов не поддерживает набор команд x86-процессоров в полном объеме. Например, на MASM невозможно написать jmp 0007h:00000000h, и приходится прибегать к различным ухищрениям: либо реализовать команду через DB, что ненаглядно и неудобно, либо заталкивать в стек сегмент/смещение, а потом делать getf, но это длинно и к тому же, воздействует на стек, которого у нас может и не быть.

Программирование на смеси 16- и 32-разрядного кода с кратковременным переходом в защищенный режим и возвращением обратно в реальный – это вообще отдельная проблема, к которой на MASM не знаешь как подступиться, но, на счастье, такое почти никому не нужно.

А вот что реально нужно – так это интеграция в мировое сообщество. Программистам обычно приходится писать под уже существующие системы и, если в состав DDK входит MASM с кучей исходных текстов драйверов, то пытаться собрать их под другим транслятором означает впустую тратить время. Опять-таки, если компилятору Microsoft Visual C++ задать ключ /FA, то он выдаст ассемблерный листинг в стиле MASM, точно так же поступит и IDA Pro, Borland C++ выберет TASM, а GCC – GNU Assembler (он же GAS). Чистый ассемблер сам по себе мало кому интересен и практически всегда он становится приложением к чему-то еще.

Качество документирования играет весьма важную роль и еще важнее – кому принадлежит проект. Трансляторы, созданные энтузиастами-одиночками, могут умереть в любой момент. Коммерческие ассемблеры крупных компаний выглядят намного более стабильными и непоколебимыми, однако никаких гарантий, что в один момент компания не прекратит поддержку, у нас нет (достаточно вспомнить историю с TASM). Открытые трансляторы, поддерживаемые независимой группой лиц, наиболее живучие. Стоило коллективу NASM чуть-чуть приостановить его развитие, как тут же появился YASM, позаимствовавший исходные тексты и добавивший все необходимое (поддержку x86-64, формат отладочной информации CodeView и т.д.).

Последнее, на что хотелось бы обратить внимание – это макросредства. Отношение к ним у программистов двоякое. Одни пользуются плодами прогресса, программируя на смеси ассемблера, бейсика и процессора Си, другие презируют их, ратуя за чистоту кода. Макросы упрощают программирование, зачастую позволяя невозможное (например, шифровать код программы еще на стадии ассемблирования), но переносимость программы от этого резко ухудшается и переход на другой транслятор становится труднореализуемым. Но, как бы то ни было, поддержка макросов совсем не обязывает этими макросами пользоваться.

Некоторые наиболее популярные ассемблеры Intel-типа:

Microsoft Macro Assembler (MASM) — ассемблер для процессоров семейства x86. Первоначально был произведен компанией Microsoft для написания программ в операционной системе MS-DOS и был в течение некоторого времени самым популярным ассемблером, доступным для неё. В 2000 г. MASM 6.15 был выпущен как часть пакета разработки Visual C++, и все версии Visual C++ после 6.0 включали в себя версию MASM. Стив Хатчессон собрал последние версии транслятора MASM, линкер от Microsoft, включаемые файлы, библиотеки, обширную документацию, статьи разных авторов, посвященные ассемблеру, и даже простенькую IDE в один дистрибутив, известный как «пакет Хатча» (MASM Hutch), бесплатно раздаваемый всем желающим на вполне лицензионной основе. MASM посвящено множество книг, что упрощает процесс обучения, а в сети можно найти много исходных текстов ассемблерных программ и библиотек, освобождающих программиста от необходимости изобретать велосипед. Также MASM является выходным языком для многих дисассемблеров (Sourcer, IDA Pro). Все это делает MASM транслятором номер один в программировании под Windows/Intel. Основным его недостатком следует признать невообразимое, чудовищное количество ошибок, которые, по традиции, Microsoft и не думало исправлять. Вот только несколько примеров: dwtoa и atodw_ex не понимают знака и по скорости очень тормозят, хотя в документации написано: A high speed ascii decimal string to DWORD conversion for applications that require high speed streaming of conversion data; ucFind не находит в строке подстроку, если длина подстроки равна 1 символу; функции BMHBinsearch и SBMBinSearch (поиск алгоритмом Бойера-Мура) реализованы с ошибками; некоторые функции обрушивают программу (если передать ustr2dw строку длиннее пяти байт – программа падает), расписывать можно до бесконечности.

Turbo Assembler (TASM) — программный пакет компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. Кроме того, TASM может работать совместно с трансляторами с языков высокого уровня фирмы Borland, такими, как Turbo C и Turbo Pascal. Как и прочие программные пакеты серии Turbo, Турбо Ассемблер больше не поддерживается.

High-Level Assembler (HLASM) — ассемблер фирмы IBM высокого уровня для операционных систем z/OS, z/VSE, z/VM и z/TPF, которые выполняются на компьютерах z/Architecture универсальной ЭВМ. Есть также версия (предназначенная прежде всего для систем, установленных на z/Architecture), которая выполняется в Linux (эта окружающая среда иногда упоминается как z/Linux).

Flat assembler (FASM) — свободно распространяемый многопроходной ассемблер, написанный Томашем Грыштаром (Tomasz Grysztar). FASM написан на самом себе, обладает небольшими размерами и очень высокой скоростью компиляции, имеет богатый и ёмкий макро-синтаксис, позволяющий автоматизировать множество рутинных задач. Поддерживаются как объектные форматы, так и форматы исполняемых файлов — это позволяет в большинстве случаев обойтись без компоновщика. В остальных случаях нужно использовать сторонние компоновщики, поскольку таковой вместе с fasm не распространяется. Помимо базового набора инструкций процессора и сопроцессора, FASM поддерживает наборы инструкций MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4a, AVX и 3DNow!, а также EM64T и AMD64 (включая AMD SVM и Intel SMX). Компиляция программы в fasm состоит из 2 стадий: препроцессирование и ассемблирование.

Теперь поговорим о недостатках. Ни на что не похожий синтаксис FASM сложен в восприятии. Категорическая несовместимость с MASM чрезвычайно затрудняет разработку Windows-драйверов (в большинстве своем создаваемых на основе примеров из DDK). Прикладным задачам, в свою очередь, требуется SDK и желательно первой свежести. Математические задачи, перемножающие матрицы, вычисляющие координаты пересечения кривых в N-мерном пространстве или трансформирующие графику, легко пишутся на FASM, поскольку не привязаны к конкретной операционной системе, никаких API-функций они не вызывают и вообще не лезут туда, где можно обойтись Си/Си++.

Netwide Assembler (NASM) — свободный (LGPL и лицензия BSD) ассемблер для архитектуры Intel x86. Используется для написания 16-, 32- и 64-битных программ. NASM был создан Саймоном Тэтхемом совместно с Юлианом Холлом и в настоящее время развивается небольшой командой разработчиков на SourceForge.net.

Yasm — ассемблер, являющийся попыткой полностью переписать ассемблер NASM. Лицензируется под лицензией BSD и в настоящее время развивается Питером Джонсоном и Майклом Ерманом.

GoAsm — ассемблер для процессоров семейства x86, созданный Джереми Гордоном для написания программ для операционных систем семейства Windows, способен создавать 32- и 64-битные версии, а также программы с поддержкой Unicode. GoAsm является проприетарным ПО и распространяется в бинарном формате. GoAsm создавался с целью создать компилятор с простым и ясным синтаксисом, производящий как можно более компактный код, скромными потребностями для обработки скриптов и возможностью добавления расширений. GoAsm не создаёт 16-разрядный код и способен работать только в плоском режиме (без сегментов), благодаря этому синтаксис очень прост. В качестве формата выходных данных используется COFF Portable Executable format, и для создания исполняемых файлов необходимо использовать дополнительный компоновщик и компилятор ресурсов. Несмотря на то, что используется Intel-синтаксис, синтаксис GoAsm несовместим ни с одним из существующих компиляторов.

Наконец, синтаксис AT&T, по умолчанию поддерживает лишь один популярный ассемблер. Это знаменитый GNU Assembler или GAS (вызывается командой as) — ассемблер проекта GNU; используется компилятором GCC почти во всех UNIX/Linux системах. Кроссплатформенная программа компилирует код для многочисленных процессорных архитектур. Распространяется на условиях свободной лицензии GPL 3.

Для ассемблера, на удивление, существует большой выбор интегрированных сред разработки, платных и бесплатных, с поддержкой разных архитектур и синтаксисов, например, RadASM, Fresh, AsmEdit, WinAsm Studio, EasyCode, GSS Visual Assembler.

3.3.1. Основы FASM

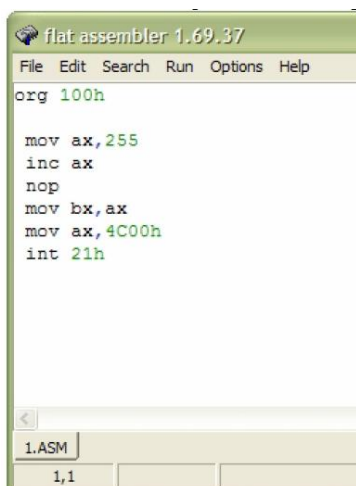
Один из самых современных и мощных ассемблеров — FASM⁵⁸.

Это новый, удобный, быстро развивающийся компилятор, написанный на себе самом. Его преимущества — поддержка сложных макросов и кроссплатформенность.

Учиться программировать начнем с процессора Intel 8086 и будем писать программы под DOS. Дело в том, что 8086 имеет относительно простую систему команд, но даже с ней разобраться начинающему нелегко, а программы под Windows и Linux имеют куда более сложную внутреннюю структуру, так что это тоже был бы не лучший выбор. Все программы без проблем будут работать под Windows, конечно, реально это будет эмуляция DOS в режиме виртуального процессора 8086, но для наших целей вполне подойдёт.

⁵⁸ Многочисленные упомянутые далее примеры расположены и подробно описаны на <http://asmworld.ru/>. Там же всегда можно скачать редактор FASM, отладчик и даже IDK FASM Editor 2.0. К сожалению, проект Asmworld не обновлялся с середины 2012 г., и его текущий статус неизвестен, что и позволило включить сюда эту важную информацию с целью сделать ее доступнее.

Начнем с простейшей .com программы под DOS. В качестве инструмента нам понадобится Pure FASM 1.69.37, который запускается без установки в весьма аскетичном виде, см. рис. ниже.



Что все это значит? Первая строка `use16` сообщает FASM, что нужно генерировать 16-битный код. Нам нужен именно такой, т.к. мы пишем программу под DOS. Вторая строка `org 100h` объясняет FASM, что следующие команды и данные будут располагаться в памяти, начиная с адреса 100h. Дело в том, что при загрузке нашей программы в память, DOS размещает в первых 256 байтах (с адресов 0000h — 00FFh) свои служебные данные. Нам эти данные изменять нежелательно.

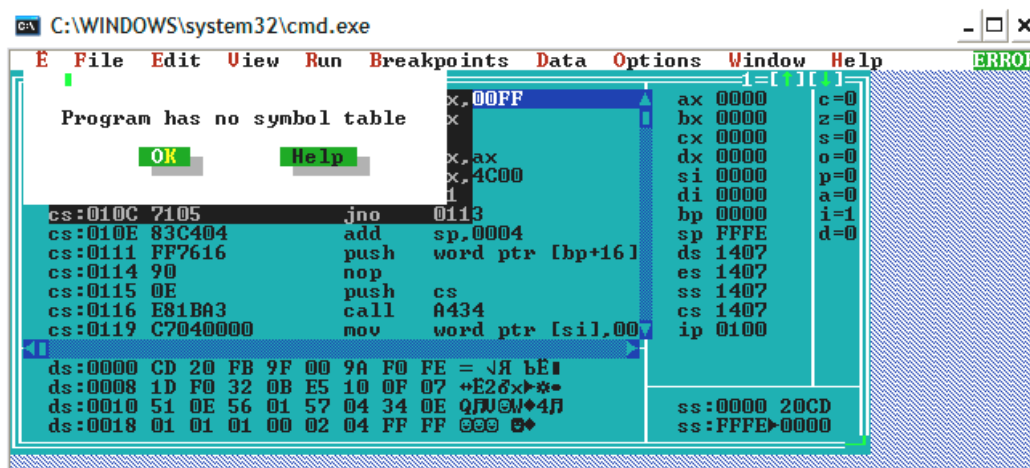
Далее идут непосредственно команды. Отступы не обязательны, но желательны — с ними код гораздо легче читать. Пустые строки игнорируются. Регистр символов значения не имеет. Можно писать большими буквами, или маленькими, или вперемешку. Четвертая строка определяет команду «поместить число 255 в регистр AX». AX — первый операнд, приёмник. 255 — второй операнд, источник. Первый операнд является регистром. Второй операнд — константа 255.

Пятая строка. Команда `inc` с одним операндом. Она заставит процессор выполнить инкремент. Единственный операнд — это регистр AX, содержимое которого и будет инкрементировано. Шестая строка. Команда `nop` без операндов. Эта команда ничего не делает и вставлена здесь шутки ради⁵⁹. Седьмая строка. Снова команда `mov`, но на этот раз оба операнда являются регистрами. Команда скопирует в BX содержимое AX.

Две последние строки — это стандартное завершение процесса в DOS. Так завершаются все DOS-программы. Вот и вся программа, пора посмотреть, что она делает!

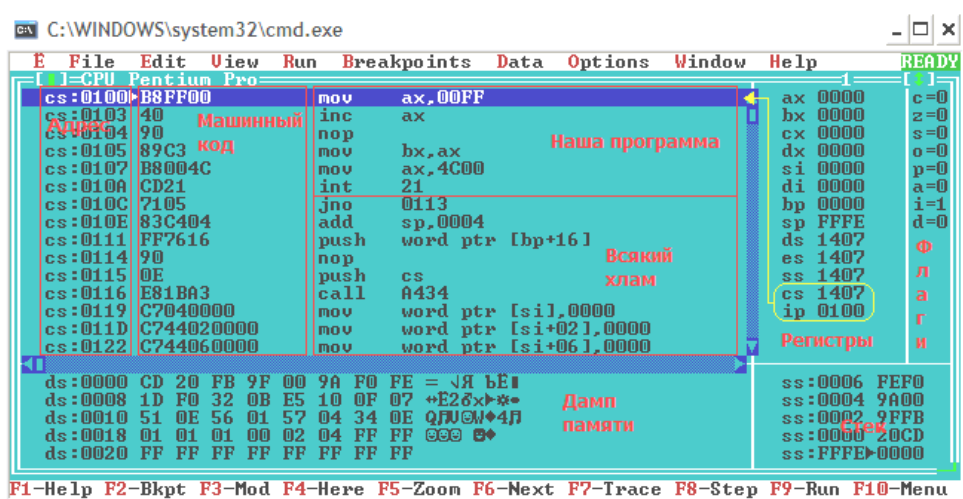
Чтобы откомпилировать программу, надо выбрать меню `Run->Compile`. FASM предложит сохранить файл, если вы этого ещё не сделали, а затем скомпилирует в .com. Файл с расширением .asm — это исходный код, обычный текстовый файл. При желании его можно открыть и редактировать блокнотом.

В результате у нас вышло 12 байт. Запустим .exe-файл и насладимся тем, что ничего не произошло! Тем не менее, программа работает, и мы в этом сейчас убедимся. То, что мы написали, не имеет графических окон и вообще никак не взаимодействует с пользователем. Программа просто временно меняет содержимое указанных регистров, значит, нам надо как-то заглянуть в них и убедиться, что их содержимое изменилось.



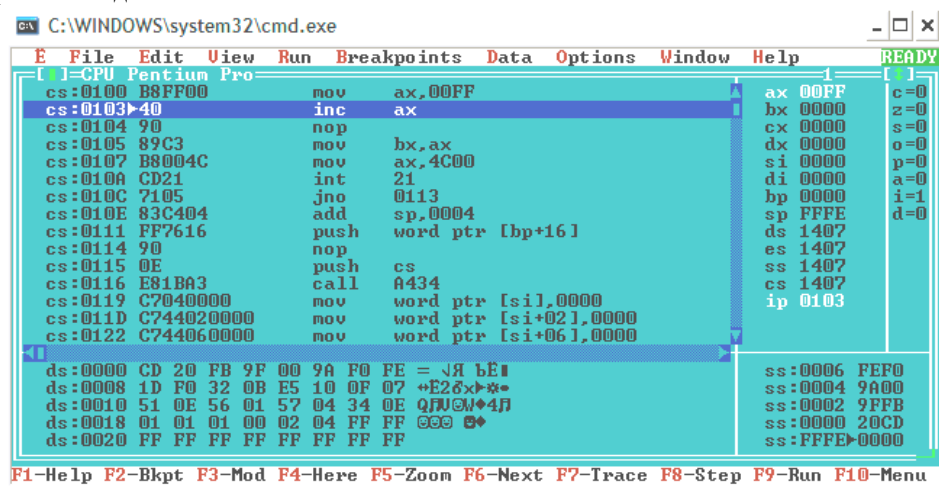
Вот тут нам пригодится еще один обязательный для любого программиста на ассемблере инструмент — отладчик, чтобы снять дамп памяти и поглядеть, что мы там натворили. Воспользуемся древним отладчиком Turbo Debugger, впрочем, в силу специфичности аудитории, которая разрабатывает такие штуки, они все выглядят пришельцами из начала 90-х. Загрузите в него наш .com файл. Сообщение означает, что в исполняемом файле нет специальных данных для отладки. Но нам эти данные и не нужны, потому что программа простая и понятная. Нажимаем OK. Turbo Debugger отображает окно CPU, в котором можно увидеть, как выполняется программа:

⁵⁹ Может закрасться подозрение, что ничего не делающая команда не нужна. Но только не в ассемблере! Это самый низкоуровневый язык, и с его помощью можно расписать выполнение программы чуть ли не по тактам процессора. А что делать, если мы синхронизируем ее выполнение программы, и надо устроить задержку в пару тактов? Не записывать же фиктивные команды типа `a=a+1`, `a=a-1`. Вот тут-то и пригодится ничего не делающая, но занимающая такт команда.



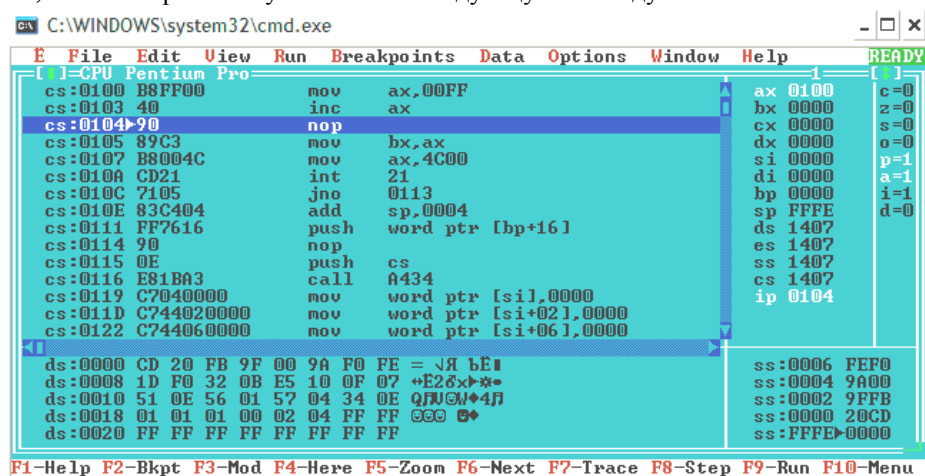
В большой области мы видим код нашей программы. Самый левый столбец — адреса, правее отображаются байты машинного кода, а ещё правее — символическое обозначение команд.

Программа размещается в памяти, начиная с адреса 0100h в сегменте кода, как мы и попросили. В нашей программе всего 6 машинных команд, а за ними в памяти находится случайный мусор (точные значения неизвестны и всякий раз будут разными). Обратите внимание, что отладчик показывает адреса и значения в шестнадцатеричном виде.



В правой части окна CPU отображаются регистры процессора и флаги. В нижней части можно увидеть дамп области памяти и стек. Адрес текущей машинной команды определяется регистрами CS и IP, эта команда показана выделенной строкой и стрелкой. Теперь нажмите F8, чтобы выполнить первую команду, и стрелка станет указывать на вторую. Изменившиеся регистры выделены белым цветом.

Регистр AX теперь содержит значение 00FFh (то есть 255, чего мы и хотели от команды «mov ax, 255»). Также изменилось значение регистра IP — оно увеличилось на размер выполненной машинной команды, а именно, на 3. Теперь CS:IP указывает на следующую команду. Снова нажимаем F8.



Значение регистра AX увеличилось на 1 и стало равным 0100h (256). Значение IP тоже увеличилось на 1, потому что длина команды «inc ax» — 1 байт. Процессор выполняет программу последовательно, одну команду за другой.

The screenshot shows the CPU Pentium Pro debugger window. The assembly code is as follows:

```

cs:0100 B8FF00 mov ax,00FF
cs:0103 40 inc ax
cs:0104 90 nop
cs:0105 89C3 mov bx,ax
cs:0107 B8004C mov ax,4C00
cs:010A CD21 int 21
cs:010C 7105 jno 0113
cs:010E 83C404 add sp,0004
cs:0111 FF7616 push word ptr [bp+16]
cs:0114 90 nop
cs:0115 0E push cs
cs:0116 E81BA3 call A434
cs:0119 C7040000 mov word ptr [si],0000
cs:011D C744020000 mov word ptr [si+02],0000
cs:0122 C744060000 mov word ptr [si+06],0000

```

The register window on the right shows the following values:

ax	0100	c	0
bx	0000	z	0
cx	0000	s	0
dx	0000	o	0
si	0000	p	1
di	0000	a	1
bp	0000	i	1
sp	FFFE	d	0
ds	1407		
es	1407		
ss	1407		
cs	1407		
ip	0105		

The memory window at the bottom shows the following data:

```

ds:0000 CD 20 FB 9F 00 9A F0 FE = Я Б Е
ds:0008 1D F0 32 0B E5 10 0F 07 = E26x
ds:0010 51 0E 56 01 57 04 34 0E = QJUCM+4J
ds:0018 01 01 01 00 02 04 FF FF = 000 0+
ds:0020 FF FF FF FF FF FF FF FF

```

Ещё раз нажимаем F8. Команда NOP ничего не делает. Меняется только значение IP — снова увеличивается на 1. Снова F8. Значение BX становится равным AX. После ещё двух нажатий F8 программа завершается.

Вот так и работает наша первая программа. Теперь перейдем к более сложным вещам, которые неоднократно пригодятся. Для того чтобы писать программы на ассемблере, нам необходимо знать, какие регистры процессора существуют и как их можно использовать. Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками древнего Intel 8086 и совместимы с его архитектурой. Это значит, что программы на ассемблере 8086 будут работать и на всех современных процессорах x86.

Все внутренние регистры процессора Intel 8086 являются 16-битными:



Всего процессор содержит 12 программно-доступных регистров, а также регистр флагов (FLAGS) и указатель команд (IP).

Регистры общего назначения (РОН) AX, BX, CX и DX используются для хранения данных и выполнения различных арифметических и логических операций. Кроме того, каждый из этих регистров поделён на 2 части по 8-бит, с которыми можно работать как с 8-битными регистрами (AH, AL, BH, BL, CH, CL, DH, DL). Младшие части регистров имеют в названии букву L (от слова Low), а старшие H (от слова High). Некоторые команды неявно используют определённый регистр, например, CX может выполнять роль счетчика цикла.

Индексные регистры предназначены для хранения индексов при работе с массивами. SI (Source Index) содержит индекс источника, а DI (Destination Index) — индекс приёмника, хотя их можно использовать и как регистры общего назначения.

Регистры-указатели BP и SP используются для работы со стеком. BP (Base Pointer) позволяет работать с переменными в стеке. Его также можно использовать в других целях. SP (Stack Pointer) указывает на

вершину стека. Он используется командами, которые работают со стеком. (Про стек я подробно расскажу в отдельной части учебного курса)

Сегментные регистры CS (Code Segment), DS (Data Segment), SS (Stack Segment) и ES (Enhanced Segment) предназначены для обеспечения сегментной адресации. Код находится в сегменте кода, данные — в сегменте данных, стек — в сегменте стека, и есть еще дополнительный сегмент данных. Реальный физический адрес получается путём сдвига содержимого сегментного регистра на 4 бита влево и прибавления к нему смещения (относительного адреса внутри сегмента). Почему все так страшно, мы узнаем, изучив архитектуру памяти x86 в соответствующей части, пока просто можете удивиться проектировщикам из Intel. COM-программа всегда находится в одном сегменте, который является одновременно сегментом кода, данных и стека. При запуске COM-программы сегментные регистры будут содержать одинаковые значения.

Указатель команд IP (Instruction Pointer) содержит адрес команды (в сегменте кода). Напрямую изменять его содержимое нельзя, но процессор делает это сам. При выполнении обычных команд значение IP увеличивается на размер выполненной команды. Существуют также команды передачи управления, которые изменяют значение IP для осуществления переходов внутри программы.

Регистр флагов FLAGS содержит отдельные биты: флаги управления и признаки результата. Флаги управления меняют режим работы процессора:

- D (Direction) — флаг направления. Управляет направлением обработки строк данных: DF=0 — от младших адресов к старшим, DF=1 — от старших адресов к младшим (для специальных строковых команд).
- I (Interrupt) — флаг прерывания. Если значение этого бита равно 1, то прерывания разрешены, иначе — запрещены.
- T (Trap) — флаг трассировки. Используется отладчиком для выполнения программы по шагам.
- Признаки результата устанавливаются после выполнения арифметических и логических команд:
- S (Sign) — знак результата, равен знаковому биту результата операции. Если равен 1, то результат — отрицательный.
- Z (Zero) — флаг нулевого результата. ZF=1, если результат равен нулю.
- P (Parity) — признак чётности результата.
- C (Carry) — флаг переноса. CF=1, если при сложении/вычитании возникает перенос/заём из старшего разряда. При сдвигах хранит значение выдвигаемого бита.
- A (Auxiliary) — флаг дополнительного переноса. Используется в операциях с упакованными двоично-десятичными числами.
- (Overflow) — флаг переполнения. OF=1, если получен результат за пределами допустимого диапазона значений.

По умолчанию, число в программе воспринимается ассемблером как десятичное. Чтобы обозначить двоичное число, необходимо к нему в конце добавить символ *b*. Восьмеричное число обозначается аналогично с помощью символа *o*. Для записи шестнадцатеричного числа FASM поддерживает 3 формы записи:

- перед числом записываются символы 0x (как в C/C++);
- перед числом записывается символ \$ (как в Pascal);
- после числа записывается символ h. Если шестнадцатеричное число начинается с буквы, необходимо добавить в начале ноль.

Этот синтаксис используется как при объявлении данных, так и в командах. Вот примеры записи чисел во всех четырёх системах:

```
mov ax, 537           ;Десятичная система
mov bl, 11010001b     ;Двоичная система
mov ch, 57o           ;Восьмеричная система
mov dl, $C2           ; \
mov si, 0x013A         ; \
mov ah, 18h           ; / Шестнадцатеричная система
mov al, 0FFh          ; /
mov al, FFh           ;Ошибка!
```

Общий синтаксис FASM

Для записи отрицательного числа в программе на ассемблере используется символ «-», например:
x db -5

Кстати, это работает и с числами в других системах счисления, и даже с символами. Со знаковыми и беззнаковыми числами нужно быть внимательным, потому что только вы знаете, какие числа используются в вашей программе! Процессору без разницы, какие данные он обрабатывает, поэтому невнимательность может привести к ошибке. Один и тот же байт может интерпретироваться по-разному, в зависимости от того, со знаком число или без. Например, числу со знаком -5 соответствует число без знака 251.

Практически любая программа, кроме машинных команд, содержит также какие-то данные. Например, числа, текстовые строки, идентификаторы, различные ресурсы и т.д. Данные могут быть как констан-

тами, не меняющими своё значение во время выполнения программы, так и переменными, в которых хранятся всякие промежуточные результаты.

Прежде всего, нужно научиться объявлять данные в программе. Для этого в ассемблере существуют директивы объявления данных.

Размер в байтах	Объявление	Резервирование
1	db	rb
2	dw du	rw
4	dd	rd
6	dp df	rp rf
8	dq	rq
10	dt	rt
N	file	----

Синтаксис объявления данных

Объявлять данные очень просто — например, чтобы объявить байт со значением, равным 5 достаточно написать:

`x db 5,`

где `x` — название нашей переменной или константы, `db` — директива объявления байта, а 5 — значение. С помощью названия в программе можно будет обращаться к ячейке памяти, содержащей наш байт. Вообще, название не обязательно, и можно его не писать, если оно не требуется: Если запустить программу в отладчике Turbo Debugger, то в окне дампа можно увидеть результат работы директивы `db`:

```
ds:0100 B8 00 4C CD 21 05 01 00  L=!*  
ds:0108 02 00 03 00 04 00 05 00  *  
ds:0110 01 01 01 01 01 48 65 6C 000000Hel  
ds:0118 6C 6F 48 65 6C 6C 6F 00 loHello  
ds:0120 48 65 6C 6C 6F 24 8D 46 Hello$HF
```

Объявление последовательностей (массивов)

Иногда в программе требуется объявить массив, то есть несколько переменных одинакового размера, расположенных в памяти друг за другом. Например, чтобы объявить массив из 5 двухбайтных чисел, можно написать:

`array1 dw 1,2,3,4,5`

где `array1` — название массива, `1,2,3,4,5` — значения элементов. Вместо `array1` компилятор FASM будет подставлять в программу адрес начала массива, то есть адрес первого элемента.

Дамп памяти будет выглядеть следующим образом (обратите внимание, младший байт каждого слова расположен перед старшим):

```
ds:0100 B8 00 4C CD 21 05 01 00  L=!*  
ds:0108 02 00 03 00 04 00 05 00  *  
ds:0110 01 01 01 01 01 48 65 6C 000000Hel  
ds:0118 6C 6F 48 65 6C 6C 6F 00 loHello  
ds:0120 48 65 6C 6C 6F 24 8D 46 Hello$HF
```

Для объявления повторяющихся элементов можно использовать такую запись (объявляем массив из 5 байтов, равных 1):

`array2 db 5 dup(1)`

и результат:

```
ds:0100 B8 00 4C CD 21 05 01 00  L=!*  
ds:0108 02 00 03 00 04 00 05 00  *  
ds:0110 01 01 01 01 01 48 65 6C 000000Hel  
ds:0118 6C 6F 48 65 6C 6C 6F 00 loHello  
ds:0120 48 65 6C 6C 6F 24 8D 46 Hello$HF
```

Объявление строк

Строка представляет собой массив байтов-символов и записывается в одинарных кавычках:

```
str1 db 'Hello'
```



Для обозначения конца строки используется специальный символ. Обычно это нулевой байт, но для функций DOS используется символ '\$'.

```
str2 db 'Hello',0 ;Обычно так
```

```
str3 db 'Hello$' ;Для DOS
```



Резервирование данных (точнее памяти для них)

Можно объявлять переменные, не имеющие определённого начального значения. Такие переменные называются неинициализированными. Например, их можно использовать в программе для хранения временного или промежуточного значения. Фактически под переменную просто резервируется место в памяти. Объявлять такие переменные можно с помощью директив *db*, *dw*, *dd*, ... и знака вопроса вместо значения.

```
x1 db ?  
x2 dw ?, ?, ?  
x3 dd 10 dup(?)
```

Кроме того, FASM поддерживает специальные директивы резервирования данных. Число после директивы обозначает количество резервируемых элементов.

То же самое можно объявить вот так:

```
x1 rb 1  
x2 rw 3  
x3 rd 10
```

С неинициализированными переменными следует быть внимательным. Не надо рассчитывать, что по умолчанию значение будет нулевым или ещё каким-то, иначе это может привести к ошибке.

Директива *file*

file — это особая директива объявления данных, которая позволяет добавить в исполняемый файл последовательность байтов из внешнего файла. Иногда это может быть очень удобно. Например, если вы хотите добавить изображение в исполняемый файл (в виде данных) или большой кусок текста, или даже код из другого файла. Директива используется следующим образом:

```
data1 file 'data.bin' ;Добавить файл data.bin целиком.
```

```
data2 file 'data.bin':20 ;Добавить байты из файла data.bin, начиная со смещения 20.
```

```
data3 file 'data.bin':20,5 ;Добавить 5 байтов из файла data.bin, начиная со смещения 20.
```

Сложение

Для сложения двух чисел предназначена команда *ADD*. Она работает как с числами со знаком, так и с числами без знака (это особенность дополнительного кода). Операнды должны иметь одинаковый размер (нельзя складывать 16- и 8-битное значение). Результат помещается на место первого операнда. В общем, эти правила справедливы для большинства команд.

После выполнения команды изменяются флаги, по которым можно определить характеристики результата:

- Флаг *CF* устанавливается, если при сложении произошёл перенос из старшего разряда. Для беззнаковых чисел это будет означать, что произошло переполнение, и результат получился некорректным.
- Флаг *OF* обозначает переполнение для чисел со знаком.
- Флаг *SF* равен знаковому биту результата (естественно, для чисел со знаком, а для беззнаковых он равен старшему биту и особо смысла не имеет).
- Флаг *ZF* устанавливается, если результат равен 0.
- Флаг *PF* — признак чётности, равен 1, если результат содержит нечётное число единиц.

Примеры:

```
add ax, 5      ;AX = AX + 5
add dx, cx     ;DX = DX + CX
add dx, cl     ;Ошибка: разный размер операндов.
```

Вычитание

Вычитание выполняется с помощью команды SUB. Результат так же помещается на место первого операнда и опять выставляются флаги. Единственная разница в том, что происходит вычитание, а не сложение. На самом деле вычитание в процессоре реализовано с помощью сложения. Процессор меняет знак второго операнда на противоположный, а затем складывает два числа. Если вам необходимо в программе поменять знак числа на противоположный, можно использовать команду NEG. У этой команды всего один операнд.

Примеры:

```
sub si, dx     ;SI = SI - DX
neg ax        ;AX = -AX
```

Инкремент и декремент

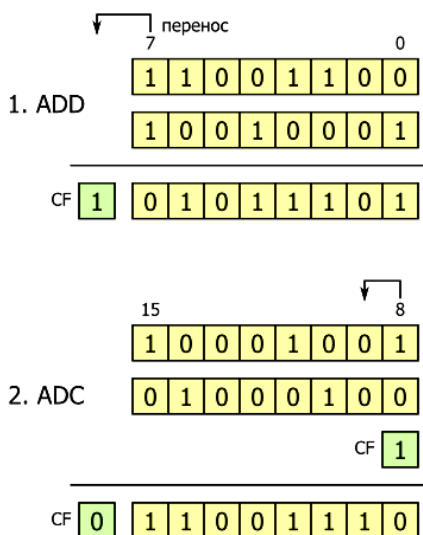
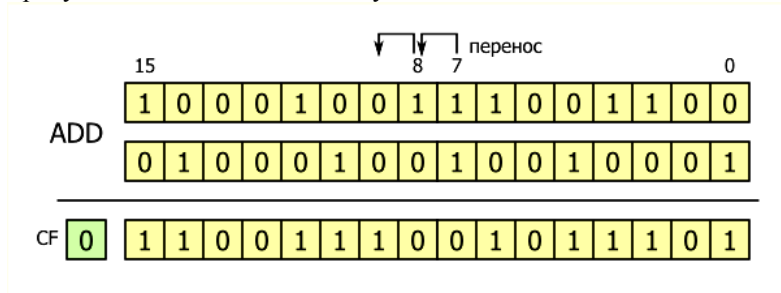
Очень часто в программах используется операция прибавления или вычитания единицы. Прибавление единицы называется инкрементом, а вычитание — декрементом. Для этих операций существуют специальные команды процессора: INC и DEC. Обратите внимание, что эти команды не изменяют значение флага CF.

Перенос

В системе команд процессоров x86 имеются специальные команды сложения и вычитания с учётом флага переноса (CF). Для сложения с учётом переноса предназначена команда ADC, а для вычитания — SBB. В общем, эти команды работают почти так же, как ADD и SUB, единственное отличие в том, что к младшему разряду первого операнда прибавляется или вычитается дополнительно значение флага CF.

Зачем нужны такие команды? Они позволяют выполнять сложение и вычитание многобайтных целых чисел, длина которых больше, чем разрядность регистров процессора (в нашем случае 16 бит). Принцип программирования таких операций очень прост — длинные числа складываются (вычитаются) по частям. Младшие разряды складываются (вычитаются) с помощью обычных команд ADD и SUB, а затем последовательно складываются (вычитаются) более старшие части с помощью команд ADC и SBB. Так как эти команды учитывают перенос из старшего разряда, то мы можем быть уверены, что ни один бит не потеряется. Этот способ похож на сложение (вычитание) десятичных чисел в столбик.

На следующем рисунке показано сложение двух двоичных чисел командой ADD:



При сложении происходит перенос из 7-го разряда в 8-й, как раз на границе между байтами. Если мы будем складывать эти числа по частям командой ADD, то перенесённый бит потеряется, и в результате мы получим ошибку. К счастью, перенос из старшего разряда всегда сохраняется в флаге CF. Чтобы прибавить этот перенесённый бит, достаточно применить команду ADC.

Аналогичная ситуация возникает с вычитанием чисел по частям. Чтобы было совсем понятно, приведу пример программы. Допустим, требуется вычислить значение формулы $k=i+j-n+1$, где переменные k , i , j и n являются 32-битными целыми числами без знака. Складывать и вычитать такие числа придётся в два этапа: сначала вычисления будут производиться с младшими словами операндов, а затем со старшими с учётом переноса.

Для прибавления единицы в данном примере нельзя использовать команду INC, так как она не влияет на флаг CF, и мы можем получить ошибку в результате!

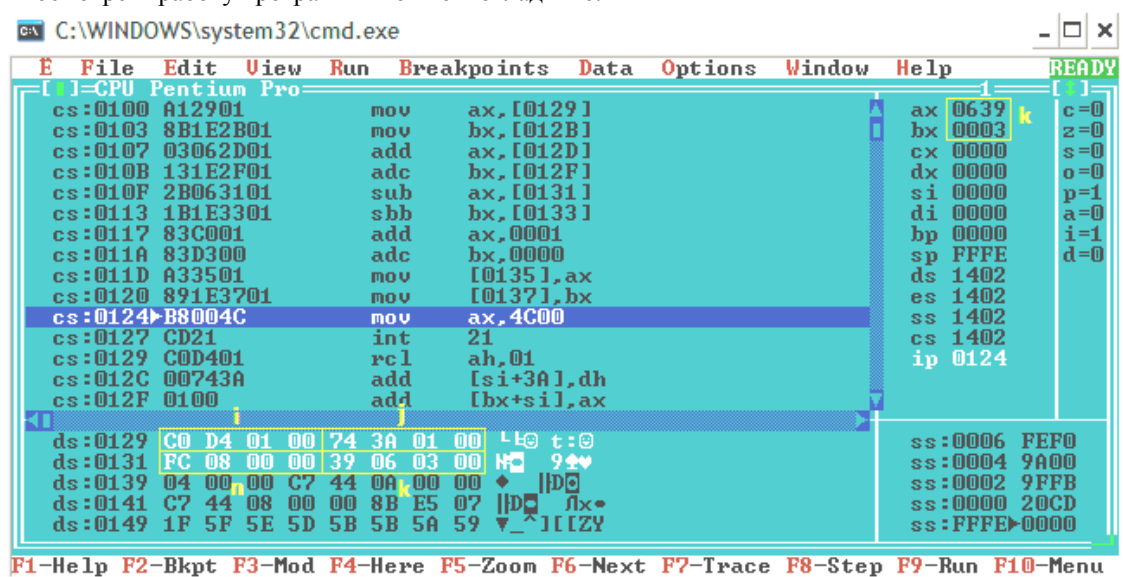
```

Пример:
use16                                ;Генерировать 16-битный код
org 100h                               ;Программа начинается с адреса 100h
mov ax,word[i]                        ;Загружаем младшую часть i в AX
mov bx,word[i+2]                     ;Загружаем старшую часть i в BX
add ax,word[j]                       ;Складываем младшие части i и j
adc bx,word[j+2]                     ;Складываем старшие части i и j
sub ax,word[n]
sbb bx,word[n+2]                     ;BX:AX = i+j-n
add ax,1                             ;Команда INC здесь не подходит!
adc bx,0                             ;BX:AX = i+j-n+1
mov word[k],ax                       ;\
mov word[k+2],bx                     ;/ Сохраняем результат в k
mov ax,4C00h                         ;\
int 21h                             ;/ Завершение программы
;-----
i dd 120000
j dd 80500
n dd 2300
k dd ?

```

Запись `word[i]` означает, что мы переопределяем размер переменной (она объявлена как `DWORD`) и обращаемся к младшему слову. Старшее слово расположено в памяти после младшего, поэтому к адресу переменной надо прибавить 2, и соответствующая запись будет иметь вид `word[i+2]`.

Посмотреть работу программы можно в отладчике:



Обратите внимание, как хранятся переменные в памяти. В процессорах Intel младший байт всегда хранится по младшему адресу, поэтому получается, что в окне дампа значения надо читать справа налево. В регистрах же числа записываются в нормальном виде. Сравните, как выглядит одно и то же значение `k` в памяти и в регистрах (старшая часть находится в `BX`, а младшая — в `AX`).

Одно из преимуществ ассемблера в том, что на нём можно реализовать работу с собственными форматами чисел, например с очень длинными целыми. А в языках высокого уровня выбор всегда ограничен компилятором.

Умножение и деление выполняются по-разному для чисел со знаком и без, поэтому в системе команд процессора x86 есть отдельные команды умножения и деления для чисел со знаком и для чисел без знака.

Умножение чисел без знака

Для умножения чисел без знака предназначена команда `MUL`. У этой команды только один операнд — второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задаётся неявно и зависит от размера операнда:

Размер операнда	Множитель	Результат
Байт	AL	AX
Слово	AX	DX:AX

Отличие умножения от сложения и вычитания в том, что разрядность результата получается в 2 раза больше, чем разрядность сомножителей. Запись «DX:AX» означает, что старшее слово результата будет находиться в DX, а младшее — в AX. Примеры:

```
mul bl      ;AX = AL * BL
mul ax      ;DX:AX = AX * AX
```

Если старшая часть результата равна нулю, то флаги CF и OF будут иметь нулевое значение. В этом случае старшую часть результата можно отбросить. Это свойство можно использовать в программе, если результат должен быть такого же размера, как множители.

Умножение чисел со знаком

Для умножения чисел со знаком предназначена команда IMUL. Эта команда имеет три формы, различающиеся количеством операндов:

- *С одним операндом* — форма, аналогичная команде MUL. В качестве операнда указывается множитель. Местоположение другого множителя и результата определяется по таблице.
- *С двумя операндами* — указываются два множителя. Результат записывается на место первого множителя. Старшая часть результата в этом случае игнорируется. Кстати, эта форма команды не работает с операндами размером 1 байт.
- *С тремя операндами* — указывается положение результата, первого и второго множителя. Вторым множителем должен быть непосредственным значением. Результат имеет такой же размер, как первый множитель, старшая часть результата игнорируется. Это форма тоже не работает с однобайтными множителями.

Примеры:

```
imul cl      ;AX = AL * CL
imul si      ;DX:AX = AX * SI
imul bx, ax   ;BX = BX * AX
imul cx, -5   ;CX = CX * (-5)
imul dx, bx, 134h ;DX = BX * 134h
```

CF = OF = 0, если произведение помещается в младшей половине результата, иначе CF = OF = 1. Для второй и третьей формы команды CF = OF = 1 означает, что произошло переполнение.

Деление чисел без знака

Деление целых двоичных чисел — это всегда деление с остатком! По аналогии с умножением, размер делителя, частного и остатка должен быть в 2 раза меньше размера делимого. Деление чисел без знака осуществляется с помощью команды DIV. У этой команды один операнд — делитель, который должен находиться в регистре или в памяти. Местоположение делимого, частного и остатка задаётся неявно и зависит от размера операнда:

Размер операнда (делителя)	Делимое	Частное	Остаток
Байт	AX	AL	AX
Слово	DX:AX	AX	DX

При выполнении команды DIV может возникнуть *прерывание*:

- если делитель равен нулю;
- если частное не помещается в отведённую под него разрядную сетку (например, если при делении слова на байт частное больше 255).

Примеры:

```
div cl      ;AL = AX / CL, остаток в AH
div di      ;AX = DX:AX / DI, остаток в DX
```

Деление чисел со знаком

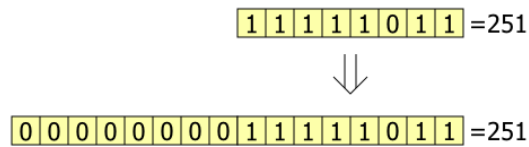
Для деления чисел со знаком предназначена команда IDIV. Единственным операндом является делитель. Местоположение делимого и частного определяется так же, как для команды DIV. Эта команда тоже генерирует прерывание при делении на ноль или слишком большом частном.

Очень часто в программах приходится выполнять действия с числами разного размера, например складывать или умножать байт и слово. Напрямую процессор не умеет выполнять такие операции, поэтому в этом случае необходимо выполнять преобразование типов. Сложность представляет преобразование меньших типов в большие (байта в слово, слова в двойное слово и т.д.).

Преобразовать больший тип в меньший гораздо проще — достаточно отбросить старшую часть. Но такое преобразование небезопасно и может привести к ошибке. Например, если значение слова без знака больше 255, то сделав из него байт, вы получите некорректное значение. Будьте внимательны, если вы используете такие трюки в своей программе.

Преобразование типов без знака

Преобразование типов выполняется по-разному для чисел со знаком и без. Для преобразования чисел без знака необходимо просто заполнить все старшие биты нулями. Например, так будет выглядеть преобразование байта в слово:



Такое преобразование можно выполнить с помощью обычной команды MOV (x объявлен как байт):

```
mov bl, [x]
mov bh, 0      ;BX = x
;Или вот так:
```

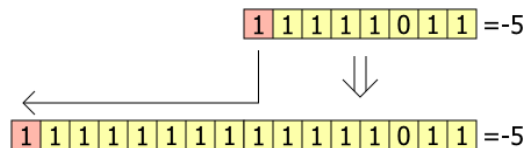
```
mov bl, [x]
sub bh, bh     ;BX = x
```

Кроме того, в системе команд процессора существует специальная команда — MOVZX (копирование с нулевым расширением). Первый операнд команды имеет размер 16 бит (слово), а второй — 8 бит (байт). Тот же результат можно получить так:

```
movzx bx, [x] ;BX = x
```

Преобразование типов со знаком

Для чисел со знаком всё немного сложнее. Если мы просто заполним старшую часть нулями, то результат будет положительным, а это не всегда верно. Поэтому преобразование выполняется путём копирования знакового бита на всю старшую часть. То есть для положительного числа со знаком старшая часть будет заполняться нулями, а для отрицательного — единицами:



Для такого преобразования предназначена команда MOVSX (копирование со знаковым расширением). Первый операнд — слово, второй операнд — байт. Например (y объявлен как байт):

```
movsx cx, [y] ;CX = y
```

Существуют ещё две команды для преобразования типов со знаком: CBW (*Convert Byte to Word* — преобразовать байт в слово) и CWD (*Convert Word to Double word* — преобразовать слово в двойное слово). У этих команд нет явных операндов. Команда CBW преобразует байт, находящийся в регистре AL, в слово в регистре AX. Команда CWD преобразует слово, находящееся в регистре AX, в двойное слово в регистрах DX:AX. Эти команды удобно использовать вместе с командами умножения и деления. Например:

```
mov al, [y]
cbw      ;AX = y
cwd      ;DX:AX = y
```

Логические операции

Для выполнения операции логического И предназначена команда AND. У этой команды 2 операнда, результат помещается на место первого операнда. Эта команда часто используется для обнуления определённых битов числа. При этом второй операнд называют *маской*. Обнуляются те биты операнда, которые в маске равны 0, значения остальных битов сохраняются.

Примеры:

```
and ax, bx      ;AX = AX & BX
and cl, 11111110b ;Обнуление младшего бита CL
and dl, 00001111b ;Обнуление старшей тетрады DL
```

Ещё одно использование этой команды — быстрое вычисление остатка от деления на степень 2. Например, так можно вычислить остаток от деления на 8:

```
and ax, 1111b   ;AX = остаток от деления AX на 8
```

Логическое ИЛИ вычисляется с помощью команды OR. У этой команды тоже 2 операнда, и результат помещается на место первого. Часто это команда используется для установки в 1 определённых битов числа. Если бит маски равен 1, то бит результата будет равен 1, остальные биты сохраняют свои значения. Примеры:

```
or al,dl           ;AL = AL | DL
or bl,10000000b    ;Установить знаковый бит BL
or cl,00100101b    ;Включить биты 0,2,5 CL
```

Инверсия выполняется с помощью команды NOT, результат помещается на место операнда. Эта команда не изменяет значения флагов. Пример:

```
not byte[bx]       ;Инверсия байта по адресу в BX
```

Для выполнения сложения по модулю два предназначена команда XOR. У команды два операнда, результат помещается на место первого. Команду можно использовать для инверсии определённых битов операнда. Инвертируются те биты, которые в маске равны 1, остальные сохраняют своё значение. Примеры:

```
xor si,di          ;SI = SI ^ DI
xor al,11110000b    ;Инверсия старшей тетрады AL
xor bp,8000h        ;Инверсия знакового бита BP
```

Обозначение операции в комментарии к первой строке используется во многих языках высокого уровня (например, C, C++, Java и т.д.). Часто XOR используют для обнуления регистров. Если операнды равны, то результат операции всегда равен 0. Такой способ обнуления работает быстрее и, в отличие от команды MOV, не содержит непосредственного операнда, поэтому команда получается короче (и не содержит нулевых байтов, что особенно нравится хакерам):

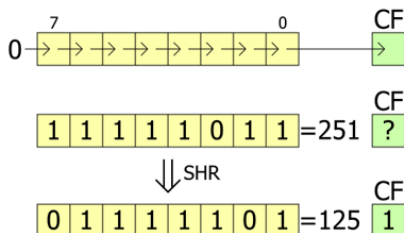
Пример. Допустим, у нас есть массив байтов. Размер массива хранится в байте без знака. Требуется в каждом байте сбросить 1-й и 5-й биты, установить 0-й и 3-й биты, инвертировать 7-й бит. А затем ещё инвертировать целиком последний байт массива.

```
use16              ;Генерировать 16-битный код
org 100h           ;Программа начинается с адреса 100h
mov bx,array       ;BX = адрес массива
movzx cx,[length]  ;CX = длина массива
mov di,cx
dec di
add di,bx          ;DI = адрес последнего элемента

ml:
mov al,[bx]        ;AL = очередной элемент массива
and al,11011101b    ;Сбрасываем 1-й и 5-й биты
or al,00001001b     ;Устанавливаем 0-й и 3-й биты
xor al,10000000b    ;Инвертируем 7-й бит
mov [bx],al        ;Сохраняем обработанный элемент
inc bx             ;В BX - адрес следующего элемента
loop ml            ;Команда цикла
not byte[di]       ;Инвертируем последний байт массива
mov ax,4C00h       ;\
int 21h            ;/ Завершение программы
;-----
length db 10
array db 1,5,3,88,128,97,253,192,138,0
```

Сдвиги — это особые операции процессора, которые позволяют реализовать различные преобразования данных, работать с отдельными битами, а также быстро выполнять умножение и деление чисел на степень 2. В этой части мы рассмотрим операции линейного сдвига, а в следующей будут циклические.

Логический сдвиг вправо (SHR)



Логический сдвиг вправо

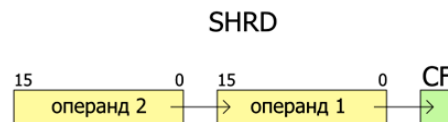
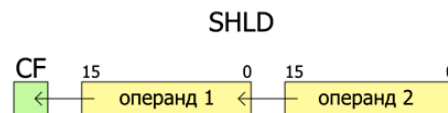
Логический сдвиг всегда выполняется без учёта знакового бита. Для логического сдвига вправо предназначена команда SHR. У этой команды два операнда. Первый операнд представляет собой сдвигаемое значение и на его место записывается результат операции. Второй операнд указывает, на сколько бит нужно осуществить сдвиг. Этим операндом может быть либо непосредственное значение, либо регистр CL.

Все биты операнда сдвигаются вправо (от старших битов к младшим). Выдвинутый бит становится значением флага CF. Старший

бит получает нулевое значение. Эта операция повторяется несколько раз, если второй операнд больше единицы. Логический сдвиг вправо можно использовать для деления целых чисел без знака на степень 2, причём сдвиг работает быстрее, чем команда деления DIV.

Пример:

```
shr ax, 1           ;Логический сдвиг AX на
1 бит вправо
shr byte [bx], cl   ;Лог. сдвиг байта по адре-
су BX на CL бит вправо
shr cl, 4           ;CL = CL / 16 (для числа без знака)
```



Арифметический сдвиг вправо

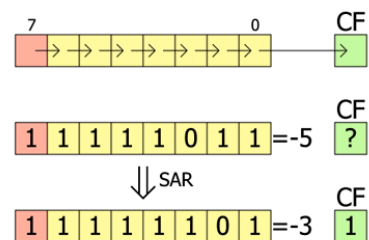
Арифметический сдвиг отличается от логического тем, что он не изменяет значение старшего бита и предназначен для чисел со знаком. Арифметический сдвиг вправо выполняется командой SAR. У этой команды тоже 2 операнда, аналогично команде SHR. Схема выполнения операции показана на рисунке.

Выдвинутый бит становится значением флага CF. Знаковый бит не изменяется. При сдвиге на 1 бит сбрасывается флаг OF. Эту команду можно использовать для деления целых чисел со знаком на степень 2 (обратите внимание, что «округление» всегда в сторону меньшего числа, поэтому для отрицательных чисел результат будет отличаться от результата деления с помощью команды IDIV).

Пример:

```
sar bx, 1           ;Арифметический сдвиг BX на 1 бит вправо
sar di, cl          ;Арифметический сдвиг DI на CL бит вправо
sar [x], 3          ;x = x / 8 (x - 8-битное значение со знаком)
```

Арифметический сдвиг вправо (SAR)



Логический и арифметический сдвиг влево

Логический сдвиг влево выполняется командой SHL, а арифметический — командой SAL. Однако на самом деле это просто синонимы для одной и той же машинной команды. Сдвиг влево одинаков для чисел со знаком и чисел без знака. У команды 2 операнда, аналогично командам SHR и SAR.

Старший бит становится значением флага CF, а младший получает нулевое значение. С помощью сдвига влево можно быстро умножать числа на степень 2. Но будьте внимательны, чтобы не получить в результате переполнение. Если при сдвиге на 1 бит меняется значение старшего бита, то устанавливается флаг OF.

Сдвиги двойной точности

Существуют ещё две команды, осуществляющие более сложные сдвиги. SHRD — сдвиг двойной точности вправо, SHLD — сдвиг двойной точности влево. У этих команд 3 операнда. Первый операнд — сдвигаемое значение и место для записи результата должен иметь размер 16 бит. Второй операнд — источник вдвигаемых битов тоже должен иметь размер 16 бит и находиться в одном из регистров. Значение второго операнда не меняется. Третий операнд — счётчик сдвигов, может быть непосредственным значением или находиться в регистре CL.

Пример программы

Программа печатает переменную размером 16-бит в двоичном виде. Используется команда сдвига влево на 1 бит, после чего анализируется значение флага CF. Если CF=0, выводим символ «0», если CF=1 выводим символ «1». Проверка битов осуществляется в цикле.

```
usel6           ;Генерировать 16-битный код
org 100h        ;Программа начинается с адреса 100h
jmp start      ;Безусловный переход на метку start
;-- Данные -----
v dw 12345
pak db 13,10,'Press any key...$'
;-----
start:
mov bx, [v]     ;BX = v
mov ah, 2       ;Функция DOS 02h - вывод символа
mov cx, 16      ;Инициализация счётчика цикла
lp:
shl bx, 1       ;Сдвиг BX на 1 бит влево
mov dl, '0'     ;dl = '0'
```

```

    jnc print      ;Переход, если выдвинутый бит равен 0
    inc dl         ;dl = dl + 1 = '1'
print:
    int 21h       ;Обращение к функции DOS 02h
    loop lp       ;Команда цикла
    mov ah,9      ;\
    mov dx,pak    ; > Вывод строки 'Press any key...'
    int 21h       ;/
    mov ah,8      ;\
    int 21h       ;/ Ввод символа без эха
    mov ax,4C00h  ;\
    int 21h       ;/ Завершение программы

```

Циклический сдвиг отличается от линейного тем, что выдвигаемые с одного конца биты вдвигаются с другой стороны, то есть движутся по кольцу. В процессоре x86 существует 2 вида циклического сдвига: простой и через флаг переноса (CF). У всех команд, рассматриваемых в этой части учебного курса, по 2 операнда, таких же, как у команд линейного сдвига. Первый операнд — сдвигаемое значение и место для записи результата. Второй операнд — счётчик сдвигов, который может находиться в регистре CL или указываться непосредственно.

Циклический сдвиг вправо выполняется командой ROR а влево — командой ROL:

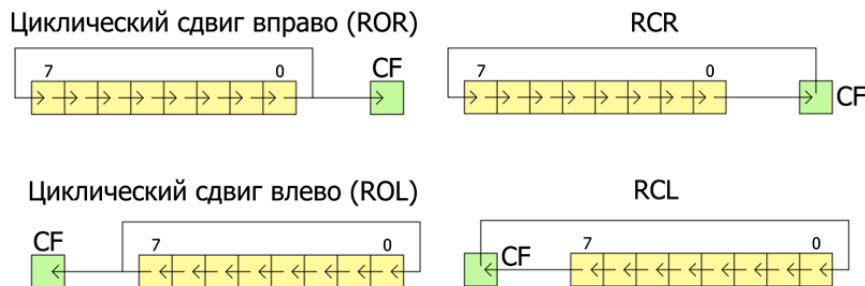
```

rol bl,1          ;Циклический сдвиг BL на 1 бит влево
ror word[si],5    ;Циклический сдвиг слова по адресу в SI на 5 бит вправо
rol ax,cl         ;Циклический сдвиг AX на CL бит влево

```

Простой циклический сдвиг и циклический сдвиг через флаг переноса

Отличие от простого циклического сдвига в том, что флаг CF участвует в сдвиге наравне с битами операнда. При сдвиге на 1 бит выдвигаемый бит помещается в CF, а значение CF вдвигается в операнд с другой стороны. При сдвиге на несколько бит эта операция повторяется многократно. Циклический сдвиг через флаг переноса выполняется командами RCR (вправо) и RCL (влево).



Опять же для сдвигов на 1 бит устанавливается флаг OF, если в результате сдвига изменяется знаковый бит операнда.

В качестве примера напомним программу для подсчёта единичных битов в байте. Для анализа битов используется команда ROL в цикле. Цикл выполняется 8 раз — по числу битов в байте. Если очередной бит равен 1, то выполняем инкремент счётчика единичных битов.

```

use16             ;Генерировать 16-битный код
org 100h          ;Программа начинается с адреса 100h
mov al,[x]        ;AL = x
xor bl,bl         ;BL = 0 (Здесь будем считать единичные биты)
mov cx,8          ;Инициализация счётчика цикла
lp:
    rol al,1      ;Цилический сдвиг AL на 1 бит влево
    jnc bit0      ;Переход, если CF=0
    inc bl        ;Инкремент счетчика единичных битов
bit0:
    loop lp       ;Команда цикла
    mov [n],bl    ;Сохраняем результат в n
    mov ax,4C00h  ;\
    int 21h       ;/ Завершение программы
;-----
x db 89h         ;Байт
n db ?           ;Количество единичных битов в байте

```

Эту программу можно немного оптимизировать. Во-первых, использовать вместо условного перехода команду ADC со нулевым вторым операндом. Это позволит прибавить 1, если CF=1 и прибавить 0, если CF=0. Во-вторых, считать единичные биты можно в регистре AH, а обнулить его в начале с помощью команды MOVZX, совместив с загрузкой байта в регистр AL. Ноль для команды ADC можно взять в регистре CH, это делает команду короче и быстрее, чем при использовании непосредственного операнда. CH равен 0 во время выполнения цикла, так как CX изменяется от 8 до 0. Вот что получилось в итоге:

```

use16                                ;Генерировать 16-битный код
org 100h                               ;Программа начинается с адреса 100h
    movzx ax, [x]                      ;AL = x, AH = 0
    mov cx, 8                          ;Инициализация счётчика цикла
lp:
    rol al, 1                          ;Цилический сдвиг AL на 1 бит влево
    adc ah, ch                         ;Прибавляем флаг CF к AH, так как CH = 0
    loop lp                            ;Команда цикла
    mov [n], bl                      ;Сохраняем результат в n
    mov ax, 4C00h                     ;\
    int 21h                           ;/ Завершение программы
;-----
x db 89h                             ;Байт
n db ?                               ;Количество единичных битов в байте

```

Адресация

Режимы адресации — это различные способы указания местоположения операндов. До этой части в учебном курсе использовались только простые режимы адресации: операнды чаще всего находились в регистрах или в переменных в памяти. Но в процессоре Intel 8086 существуют также более сложные режимы, которые позволяют организовать работу с массивами, структурами, локальными переменными и указателями. В этой части речь пойдет о всех возможных режимах адресации и примерах их использования.

1. Неявная адресация

Местоположение операнда фиксировано и определяется кодом операции.

```

cbw
mul al

```

Команда CBW всегда работает с регистрами AX и AL, а у команды MUL фиксировано положение первого множителя и результата. Такой режим адресации делает машинную команду короткой, так как в ней отсутствует указание одного или нескольких операндов.

2. Непосредственная адресация

При непосредственной адресации значение операнда является частью машинной команды. Понятно, что в этом случае операнд представляет собой константу.

```

mov al, 5
add bx, 1234h
mov dx, a

```

Обратите внимание, что в третьей строке в DX помещается *адрес* метки или переменной *a*, а вовсе не значение по этому адресу. Это особенность синтаксиса FASM. По сути, адрес метки тоже является числовой константой.

3. Абсолютная прямая адресация

В машинной команде содержится адрес операнда, находящегося в памяти.

```

mov dx, [a]

```

Вот тут уже в DX помещается значение из памяти по адресу *a*. Сравните с предыдущим пунктом. Квадратные скобки обозначают обращение по адресу, указанному внутри этих скобок.

4. Относительная прямая адресация

Этот режим используется в командах передачи управления. В машинной команде содержится смещение, которое прибавляется к значению указателя команд IP. То есть, указывается не сам адрес перехода, а на сколько байтов вперёд или назад надо перейти.

```

metka:
...
loop metka

```

У такого режима адресации два преимущества. Во-первых, машинная команда становится короче, так как она содержит не полный адрес, а только смещение. Во-вторых, такой код не зависит от адреса, по которому он размещается в памяти.

5. Регистровая адресация

Операнд находится в регистре.

```

add ax, bx

```

6. Косвенная регистровая (базовая) адресация

Адрес операнда находится в одном из регистров BX, SI или DI.

```
add ax, [bx]
mov dl, [si]
```

Размер операнда в памяти здесь определяется размером первого операнда. Так как AX — 16-разрядный регистр, то из памяти берётся слово по адресу в BX. Так как DL — 8-разрядный регистр, то из памяти берётся байт по адресу в SI. Это правило верно и для других режимов адресации.

7. Косвенная регистровая (базовая) адресация со смещением

Адрес операнда вычисляется как сумма содержимого регистра BX, BP, SI или DI и 8- или 16-разрядного смещения.

```
add ax, [bx+2]
mov dx, [array1+si]
```

В качестве смещения можно указать число или адрес метки. О размере смещения не беспокойтесь — компилятор сам его определяет и использует нужный формат машинной команды.

8. Косвенная базовая индексная адресация

Адрес операнда вычисляется как сумма содержимого одного из базовых регистров BX или BP и одного из индексных регистров SI или DI.

```
mov ax, [bp+si]
add ax, [bx+di]
```

Например, в одном из регистров может находиться адрес начала массива в памяти, а в другом — смещение какого-то элемента относительно начала.

9. Косвенная базовая индексная адресация со смещением

Адрес операнда вычисляется как сумма содержимого одного из базовых регистров BX или BP, одного из индексных регистров SI или DI и 8- или 16-разрядного смещения.

```
mov al, [bp+di+5]
mov bl, [array2+bx+si]
```

Пример программы

Допустим, имеется массив 32-битных целых чисел со знаком. Количество элементов массива хранится в 16-битной переменной без знака. Требуется вычислить среднее арифметическое элементов массива и сохранить его в 32-битной переменной со знаком.

```
use16                                ;Генерировать 16-битный код
org 100h                             ;Программа начинается с адреса 100h
    sub ax, ax                       ;AX = 0
    cwd                              ;DX = 0
    mov si, ax                       ;SI = 0 - смещение элемента от начала массива
    mov bx, array                    ;Помещаем в BX адрес начала массива
    mov di, n                        ;Помещаем в DI адрес n
    mov cx, [di]                     ;CX = n
lp1:
    add ax, [bx+si]                   ;Прибавление младшего слова
    adc dx, [bx+si+2]                 ;Прибавление старшего слова
    add si, 4                         ;Увеличиваем смещение в SI на 4
    loop lp1                          ;Команда цикла
    idiv word[di]                     ;Делим сумму на количество элементов
    cwd                              ;DX:AX = AX
    mov word[m], ax                   ;\ Сохраняем
    mov word[m+2], dx                 ;/ результат
    mov ax, 4C00h                     ;\
    int 21h                           ;/ Завершение программы
;-----
n      dw 10
array  dd 10500, -7500, -15000, 10000, -8000
      dd 6500, 11500, -5000, 10500, -20000
m      dd ?
```

Метки, циклы и ветвления

Метка представляет собой символическое имя, вместо которого компилятор подставляет адрес. В программе на ассемблере можно присвоить имя любому адресу в коде или данных. Обычно метки используются для организации переходов, циклов или каких-то манипуляций с данными. По сути, имена переменных, объявленных с помощью директив объявления данных, тоже являются метками. Но с ними компилятор допол-

нительно связывает размер переменной. Метка объявляется очень просто: достаточно в начале строки написать имя и поставить двоеточие.

Например:

```
m1: mov ax, 4C00h
    int 21h
```

Теперь вместо имени *m1* компилятор везде будет подставлять адрес команды *mov ax, 4C00h*. Можно объявлять метку на пустой строке перед командой:

```
exit_app:
    mov ax, 4C00h
    int 21h
```

Имя метки может состоять из латинских букв, цифр и символов подчёркивания, но должно начинаться с буквы. Имя метки должно быть уникальным. В качестве имени метки нельзя использовать директивы и ключевые слова компилятора, названия команд и регистров (в этом случае FASM покажет сообщение об ошибке). FASM различает регистр символов в именах меток. Можно также объявлять несколько меток на один адрес. Например:

```
no_error:
exit_app:
m1: mov ax, 4C00h
```

Синтаксис FASM не ограничивается объявлением простых меток. Сейчас мы рассмотрим дополнительную директиву для создания меток, а также научимся использовать локальные и анонимные метки. В синтаксисе FASM существует 3 основных способа объявления меток:

1. Имя метки, после которого ставится двоеточие. Это самый простой способ. Обычно так объявляются метки в коде.

```
exit_app:
    mov ax, 4C00h
    int 21h
```

2. Использование директив объявления данных. Имя переменной является, по сути, той же меткой. Отличие от первого способа в том, что дополнительно с именем метки связывается размер переменной.

```
x db 5
y dw 34, 1200, ?
z rd 1
```

3. Объявление метки с помощью специальной директивы *label*. Более сложный, но зато самый гибкий способ. Его мы рассмотрим подробнее.

Директива *label* имеет следующий формат:

```
label <имя_метки> [размер] [at адрес]
```

У директивы может быть 3 параметра. Обязательным является только первый параметр — имя метки. Второй параметр — оператор размера (*byte*, *word*, *dword* и т.д.). Он связывает с меткой размер переменной, аналогично тому, как это делают директивы объявления данных. Далее может быть указан оператор *at* и адрес метки. Адрес может представлять собой константу, числовое выражение или имя другой метки. Если адрес не указан, то для создания метки используется адрес того места, где она объявлена.

```
label m1                ;То же самое, что 'm1:'
label m2 byte           ;Похоже на 'm2 db ?', но память не резервируется
label m3 dword          ;Похоже на 'm3 dd ?', но память не резервируется
                        ;Все 3 метки указывают на один и тот же адрес!
```

Следующий пример показывает, как можно использовать в программе директиву *label*. Допустим, объявлена переменная *x* размером слово. Требуется обнулить старший байт *x*. Воспользовавшись директивой *label*, можно обратиться к старшему байту как к отдельной переменной:

```
x dw 12345              ;Переменная-слово
label xh byte at x+1    ;Объявление метки для обращения к старшему байту
...
start:
```

```
    xor al, al          ;AL=0
    mov [xh], al         ;xh=0 (старший байт x)
    mov byte[x+1], al    ;То же самое без использования метки xh
```

Кроме того, адрес может содержать базовые и индексные регистры для косвенной адресации. Такие метки можно использовать для обращения к параметрам и локальным переменным процедуры. Например:

```
label i word at bp-2    ;Локальная переменная
...
inc [i]                 ;Инкремент локальной переменной
```

Локальные метки

Локальная метка — это метка, имя которой начинается с точки. Во время генерации кода FASM автоматически добавляет к имени локальной метки имя последней объявленной «глобальной» метки. Таким образом, имена локальных меток могут повторяться, если между ними есть хотя бы одна «глобальная» метка.

Локальные метки удобно использовать, например, внутри процедуры. Можно дать им простые, понятные имена и не беспокоиться, что где-то в коде уже объявлена метка с таким именем.

Локальные метки намного улучшают читаемость кода. Если потребуется обратиться к локальной метке из другого места программы, это можно сделать, указав её полное имя.

Особым образом обрабатываются метки, имя которых начинается с двух точек. Такие метки ведут себя как глобальные, но не становятся префиксом для локальных меток.

Анонимные метки

Анонимная метка — это метка с именем `@@`. В программе можно объявлять сколько угодно анонимных меток, но обратиться получится только к ближайшей. Для этого существуют специальные имена: вместо `@b` (или `@r`) FASM подставляет адрес предыдущей анонимной метки, а вместо `@f` — адрес следующей анонимной метки. Этого, как правило, достаточно, чтобы реализовать простой цикл, переход или проверку условия. Таким образом, можно избавиться от большого количества именных меток.

Обычно рекомендуется везде, где это возможно, использовать локальные и анонимные метки. Они делают код программы более понятным и не захламляют пространство имён.

Команда LOOP

Для организации цикла предназначена команда LOOP. У этой команды один операнд — имя метки, на которую осуществляется переход. В качестве счётчика цикла используется регистр CX. Команда LOOP выполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде. Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Понятно, что максимально может быть 65535 повторений. Ещё одно ограничение связано с дальностью перехода. Метка должна находиться в диапазоне -127...+128 байт от команды LOOP (если это не так, FASM сообщит об ошибке).

Пример цикла

Вот простая программа, которая будет печатать все буквы английского алфавита. ASCII-коды этих символов расположены последовательно, поэтому можно выводить их в цикле. Для вывода символа на экран используется функция DOS 02h (выводимый байт должен находиться в регистре DL).

```
use16                                ;Генерировать 16-битный код
org 100h                             ;Программа начинается с адреса 100h
mov ah, 02h                          ;Для вызова функции DOS 02h - вывод символа
mov dl, 'A'                          ;Первый выводимый символ
mov cx, 26                           ;Счётчик повторений цикла

metka:
int 21h                             ;Обращение к функции DOS
inc dl                               ;Следующий символ
loop metka                           ;Команда цикла
mov ah, 09h                          ;Функция DOS 09h - вывод строки
mov dx, press                        ;В DX адрес строки
int 21h                             ;Обращение к функции DOS
mov ah, 08h                          ;Функция DOS 08h - ввод символа без эха
int 21h                             ;Обращение к функции DOS
mov ax, 4C00h                        ;\
int 21h                             ;/ Завершение программы

;-----
press:
db 13,10,'Press any key...$'
```

Команды `int 21h` и `inc dl` будут выполняться в цикле 26 раз. Чтобы программа не закрылась сразу, используется функция DOS 08h — ввод символа с клавиатуры без эха, то есть вводимый символ не отображается. Перед этим выводится предложение нажать любую кнопку (но *Reset* лучше не нажимать). Для примера адрес строки объявлен с помощью метки. Символы с кодами 13 и 10 обозначают переход на следующую строку. Символ 13(0Dh) называется CR — *Carriage Return* — возврат каретки, а символ 10(0Ah) LF — *Line Feed* — перевод строки. Эти символы унаследованы со времён древних телетайпов, когда текст печатался, как на печатной машинке.

Так выглядит результат работы программы:



Вложенные циклы

Иногда требуется организовать вложенный цикл, то есть цикл внутри другого цикла. В этом случае необходимо сохранить значение CX перед началом вложенного цикла и восстановить после его завершения (перед командой LOOP внешнего цикла). Сохранить значение можно в другой регистр, во временную переменную или в стек. Следующая программа выводит все доступные ASCII-символы в виде таблицы 16×16. Значение счётчика внешнего цикла сохраняется в регистре BX.

```
use16                                ;Генерировать 16-битный код
org 100h                             ;Программа начинается с адреса 100h
    mov ah,02h                       ;Для вызова функции DOS 02h - вывод символа
    sub dl,dl                         ;Первый выводимый символ
    mov cx,16                         ;Счётчик внешнего цикла (по строкам)
lp1:
    mov bx,cx                         ;Сохраняем счётчик в BX
    mov cx,16                         ;Счётчик внутреннего цикла (по столбцам)
lp2:
    int 21h                           ;Обращение к функции DOS
    inc dl                             ;Следующий символ
    loop lp2                           ;Команда внутреннего цикла
    mov dh,dl                         ;Сохраняем значение DL в DH
    mov dl,13                         ;\
    int 21h                           ; \
    mov dl,10                         ; / Переход на следующую строку
    int 21h                           ;/
    mov dl,dh                         ;Восстанавливаем значение DL
    mov cx,bx                         ;Восстанавливаем значение счётчика
    loop lp1                           ;Команда внешнего цикла
    mov ah,09h                       ;Функция DOS 09h - вывод строки
    mov dx,press                      ;В DX адрес строки
    int 21h                           ;Обращение к функции DOS
    mov ah,08h                       ;Функция DOS 08h - ввод символа без эха
    int 21h                           ;Обращение к функции DOS
    mov ax,4C00h                     ;\
    int 21h                           ;/ Завершение программы
;-----
press db 13,10,'Press any key...$'
```

Вот результат работы такой программы:



и ее код.

Безусловные переходы

Безусловный переход — это переход, который выполняется всегда. Безусловный переход осуществляется с помощью команды JMP. У этой команды один операнд, который может быть непосредственным адресом (меткой), регистром или ячейкой памяти, содержащей адрес. Существуют также «дальние» переходы — между сегментами, однако здесь мы их рассматривать не будем.

Примеры безусловных переходов:

```
jmp metka      ;Переход на метку
jmp bx         ;Переход по адресу в ВХ
jmp word[bx]   ;Переход по адресу, содержащемуся в памяти по адресу в ВХ
```

Условные переходы

Условный переход осуществляется, если выполняется определённое условие, заданное флагами процессора (кроме одной команды, которая проверяет CX на равенство нулю). Как вы помните, состояние флагов изменяется после выполнения арифметических, логических и некоторых других команд. Если условие не выполняется, то управление переходит к следующей команде.

Существует много команд для различных условных переходов. Также для некоторых команд есть синонимы (например, JZ и JE — это одно и то же). Для наглядности все команды условных переходов приведены в таблице:

Команда	Переход, если	Условие перехода
JZ/JE	нуль или равно	ZF=1
JNZ/JNE	не нуль или не равно	ZF=0
JC/JNAE/JB	есть переполнение/не выше и не равно/ниже	CF=1
JNC/JAE/JNB	нет переполнения/выше или равно/не ниже	CF=0
JP	число единичных бит чётное	PF=1
JNP	число единичных бит нечётное	PF=0
JS	знак равен 1	SF=1
JNS	знак равен 0	SF=0
JO	есть переполнение	OF=1
JNO	нет переполнения	OF=0
JA/JNBE	выше/не ниже и не равно	CF=0 и ZF=0
JNA/JBE	не выше/ниже или равно	CF=1 или ZF=1
JG/JNLE	больше/не меньше и не равно	ZF=0 и SF=OF
JGE/JNL	больше или равно/не меньше	SF=OF
JL/JNGE	меньше/не больше и не равно	SF≠OF
JLE/JNG	меньше или равно/не больше	ZF=1 или SF≠OF
JCXZ	содержимое CX равно нулю	CX=0

У всех этих команд один операнд — имя метки для перехода. Обратите внимание, что некоторые команды применяются для беззнаковых чисел, а другие — для чисел со знаком. Сравнения «выше» и «ниже» относятся к беззнаковым числам, а «больше» и «меньше» — к числам со знаком. Для беззнаковых чисел при знаке переполнения будет флаг CF, а соответствующими командами перехода JC и JNC. Для чисел со знаком о переполнении можно судить по состоянию флага OF, поэтому им соответствуют команды перехода JO и JNO. Команды переходов не изменяют значения флагов.

В качестве примера рассмотрим небольшую программу для сложения двух чисел со знаком с проверкой переполнения. В случае переполнения будет выводиться сообщение об ошибке. Вы можете поменять значения объявленных переменных, чтобы переполнение возникало или не возникало при их сложении, и посмотреть, что будет выводить программа:

```
use16      ;Генерировать 16-битный код
org 100h   ;Программа начинается с адреса 100h
mov al,[x] ;AL = x
add al,[y] ;AL = x + y
jo error   ;Переход, если переполнение
mov ah,09h ;\
mov dx,ok_msg ; > Вывод строки 'OK'
int 21h    ;/
exit:
mov ah,09h ;\
mov dx,pak ; > Вывод строки 'Press any key...'
int 21h    ;/
mov ah,08h ;\
int 21h    ;/ Ввод символа
mov ax,4C00h ;\
int 21h    ;/ Завершение программы
```

```

error:
    mov ah,09h          ;\
    mov dx,err_msg      ; > Вывод сообщения об ошибке
    int 21h             ;/
    jmp exit            ;Переход на метку exit
;-----
x          db -89
y          db -55
err_msg    db 'Error: overflow detected.',13,10,'$'
ok_msg     db 'OK',13,10,'$'
pak        db 'Press any key...$'

```

Команды CMP и TEST

Часто для формирования условий переходов используются команды CMP и TEST. Команда CMP предназначена для сравнения чисел. Она выполняется аналогично команде SUB: из первого операнда вычитается второй, но результат не записывается на место первого операнда, изменяются только значения флагов. Например:

```

cmp al,5          ;Сравнение AL и 5
jl c1             ;Переход, если AL < 5 (числа со знаком)
cmp al,5          ;Сравнение AL и 5
jb c1             ;Переход, если AL < 5 (числа без знака)

```

Команда TEST работает аналогично команде AND, но также результат не сохраняется, изменяются только флаги. С помощью этой команды можно проверить состояние различных битов операнда. Например:

```

test bl,00000100b ;Проверить состояние 2-го бита BL
jz c2             ;Переход, если 2-й бит равен 0

```

Пример программы

Простая программка, которая выводит меню и предлагает пользователю сделать выбор. Для ввода символа используется функция DOS 01h (при вводе символ отображается на экране). В зависимости от введённого символа осуществляется переход на нужный кусок кода. Для разнообразия, данные находятся в начале программы, а не в конце (кстати, обычно так и делают). Чтобы данные не выполнились как код, перед ними стоит команда безусловного перехода.

```

use16              ;Генерировать 16-битный код
org 100h           ;Программа начинается с адреса 100h
    jmp start      ;Безусловный переход на метку start
;-----
menu    db '1 - Print hello',13,10
        db '2 - Print go away',13,10
        db '0 - Exit',13,10,'$'
select  db 13,10,'Select>$'
hello   db 13,10,'Hello!',13,10,13,10,'$'
go_away db 13,10,'Go away!',13,10,13,10,'$'
;-----
start:
    mov ah,09h          ;\
    mov dx,menu         ; > Вывод меню
    int 21h             ;/
select_loop:
    mov ah,09h          ;\
    mov dx,select       ; > Вывод строки 'Select>'
    int 21h             ;/
    mov ah,01h          ;Функция DOS 01h - ввод символа
    int 21h             ;Введённый символ помещается в AL
    cmp al,'1'          ;Сравнение введённого символа с '1'
    je c1               ;Переход, если равно
    cmp al,'2'          ;Сравнение введённого символа с '2'
    je c2               ;Переход, если равно
    cmp al,'0'          ;Сравнение введённого символа с '0'
    je exit             ;Переход, если равно
    jmp select_loop     ;Безусловный переход
c1:
    mov ah,09h          ;\

```

```

mov dx,hello          ; > Вывод строки 'Hello'
int 21h               ;/
jmp start             ;Безусловный переход
c2:
mov ah,09h            ;\
mov dx,go_away        ; > Вывод строки 'Go away'
int 21h               ;/
jmp start             ;Безусловный переход
exit:
mov ax,4C00h          ;\
int 21h               ;/ Завершение программы

```

И результат работы:

```

C:\jumps.com
1 - Print hello
2 - Print go away
0 - Exit
Select>2
Go away!
1 - Print hello
2 - Print go away
0 - Exit
Select>1
Hello!
1 - Print hello
2 - Print go away
0 - Exit
Select>3
Select>8
Select>_

```

Кроме команды LOOP и команд условных переходов, существуют ещё две команды, позволяющие организовывать циклы. Это команды LOOPZ (или её синоним LOOPE) и LOOPNZ (синоним — LOOPNE). Действие этих команд очень напоминает LOOP, за исключением того, что дополнительно анализируется флаг нуля ZF.

Переход к метке цикла осуществляется в том случае, если после декремента содержимое CX не равно 0 и выполняется условие: ZF=1 (для команды LOOPZ/LOOPE) или ZF=0 (LOOPNZ/LOOPNE).

Эти команды удобно использовать в алгоритмах, где цикл должен завершаться в двух случаях:

- выполнено требуемое количество итераций;
- выполнено некоторое условие досрочного завершения цикла.

Простейший пример такого алгоритма — поиск числа или символа в массиве. Поиск завершается, если один из элементов массива совпал с искомым или если достигнут конец массива. В качестве примера рассмотрим программу для поиска символов в строке. Пользователь вводит символ, а программа определяет, содержится такой символ в строке или нет. Для выхода из программы нужно нажать ENTER.

```

use16                ;Генерировать 16-битный код
org 100h              ;Программа начинается с адреса 100h
jmp start            ;Безусловный переход на метку start
;Данные -----
string db 'Hello!',13,10,'$'
length db 6
s_entchar db 'Enter char (Press ENTER to exit):$'
s_found db ' - found!',13,10,'$'
s_nfound db ' - not found!',13,10,'$'
;-----
start:
mov ah,09h           ;\
mov dx,string        ; > Вывод строки
int 21h              ;/
main:
mov dx,s_entchar     ;\
int 21h              ;/ Вывод приглашения для ввода символа

```

```

mov ah,01h          ;\
int 21h             ;/ Ввод символа
cmp al,0Dh          ;Нажата клавиша ENTER?
je exit             ;Если да, то переход на метку exit
mov bx,string-1     ;BX = (адрес строки - 1)
movzx cx,[length]   ;CX = длина строки
search:
inc bx              ;Инкремент BX
cmp al,[bx]         ;Сравнение введённого символа с символом строки
loopne search       ;Цикл, если не равно.
je found            ;Если равно, то символ найден.

mov dx,s_nfound     ;DX = адрес строки ' - not found!'
print_result:
mov ah,09h          ;\
int 21h             ;/ Вывод результата поиска
jmp main            ;Безусловный переход на метку main
found:
mov dx,s_found      ;DX = адрес строки ' - found!'
jmp print_result    ;Безусловный переход на метку print_result
exit:
mov ax,4C00h        ;\
int 21h             ;/ Завершение программы

```

И результат работы такой программы:

```

C:\search.com
Hello!
Enter char (Press ENTER to exit):h - not found!
Enter char (Press ENTER to exit):l - found!
Enter char (Press ENTER to exit):o - found!
Enter char (Press ENTER to exit):H - found!
Enter char (Press ENTER to exit):a - not found!
Enter char (Press ENTER to exit):z - not found!
Enter char (Press ENTER to exit):! - found!
Enter char (Press ENTER to exit):

```

Обратите внимание, изначально в BX загружается значение (адрес строки — 1), так как цикл начинается с команды инкремента.

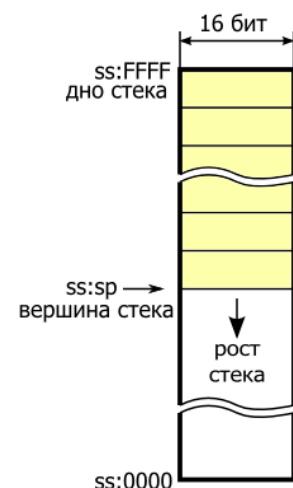
Стек

Стек является неотъемлемой частью архитектуры процессора и поддерживается на аппаратном уровне: в процессоре есть специальные регистры (SS, BP, SP) и команды для работы со стеком. Обычно стек используется для сохранения адресов возврата и передачи аргументов при вызове процедур (о процедурах в следующей части), также в нём выделяется память для локальных переменных. Кроме того, в стеке можно временно сохранять значения регистров.

Стек располагается в оперативной памяти в сегменте стека, и поэтому адресуется относительно сегментного регистра SS. Шириной стека называется размер элементов, которые можно помещать в него или извлекать. В нашем случае ширина стека равна двум байтам или 16 битам. Регистр SP (указатель стека) содержит адрес последнего добавленного элемента. Этот адрес также называется вершиной стека. Противоположный конец стека называется дном.

Дно стека находится в верхних адресах памяти. При добавлении новых элементов в стек значение регистра SP уменьшается, то есть стек растёт в сторону младших адресов. Как вы помните, для COM-программ данные, код и стек находятся в одном и том же сегменте, поэтому если постараться, стек может разрастись и затереть часть данных и кода (надеюсь, с вами такой беды не случится).

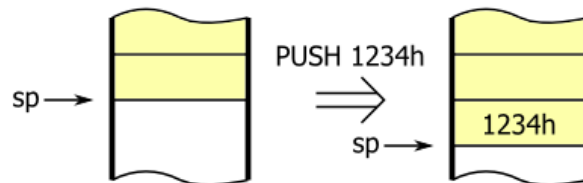
Для стека существуют всего две основные операции: добавление элемента на вершину стека (PUSH) и извлечение элемента с вершины стека (POP).



Добавление элемента в стек

У этой команды один операнд, который может быть непосредственным значением, 16-битным регистром (в том числе сегментным) или 16-битной переменной в памяти. Команда работает следующим образом:

- значение в регистре SP уменьшается на 2 (так как ширина стека — 16 бит или 2 байта);
- операнд помещается в память по адресу в SP.



Вот несколько примеров использования этой команды:

```
push -5           ;Поместить -5 в стек
push ax          ;Поместить AX в стек
push ds          ;Поместить DS в стек
push [x]           ;Поместить x в стек (x объявлен как слово)
push word [bx]     ;Поместить в стек слово по адресу в BX
```

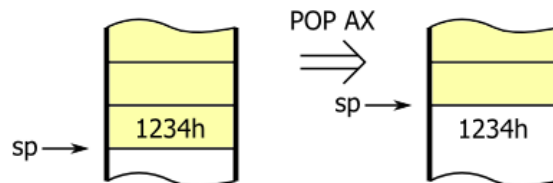
Существуют ещё 2 команды для добавления в стек. Команда **PUSHF** помещает в стек содержимое регистра флагов. Команда **PUSHA** помещает в стек содержимое всех регистров общего назначения в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI (значение DI будет на вершине стека). Значение SP помещает то, которое было до выполнения команды. Обе эти команды не имеют операндов.

Извлечение элемента из стека

У этой команды также один операнд, который может быть 16-битным регистром (в том числе сегментным, но кроме CS) или 16-битной переменной в памяти. Команда работает следующим образом:

- операнд читается из памяти по адресу в SP;
- значение в регистре SP увеличивается на 2.

Обратите внимание, что извлеченный из стека элемент не обнуляется и не затирается в памяти, а просто остаётся как мусор. Он будет перезаписан при помещении нового значения в стек.



Соответственно, есть ещё 2 команды. **POPF** помещает значение с вершины стека в регистр флагов. **POPA** восстанавливает из стека все регистры общего назначения (но при этом значение для SP игнорируется).

```
pop cx           ;Поместить значение из стека в CX
pop es           ;Поместить значение из стека в ES
pop [x]           ;Поместить значение из стека в переменную x
pop word [di]     ;Поместить значение из стека в слово по адресу в DI
```

Пример программы

Имеется двумерный массив — таблица 16-битных значений со знаком размером n строк на m столбцов. Программа вычисляет сумму элементов каждой строки и сохраняет результат в массиве *sum*. Первый элемент массива будет содержать сумму элементов первой строки, второй элемент — сумму элементов второй строки и так далее.

```
use16             ;Генерировать 16-битный код
org 100h          ;Программа начинается с адреса 100h
jmp start        ;Переход к метке start
; Данные
n db 4             ;Количество строк
m db 5             ;Количество столбцов
;Двумерный массив - таблица с данными
table:
  dw 12,45, 0,82,34
  dw 46,-5,87,11,56
  dw 35,21,77,90,-9
```

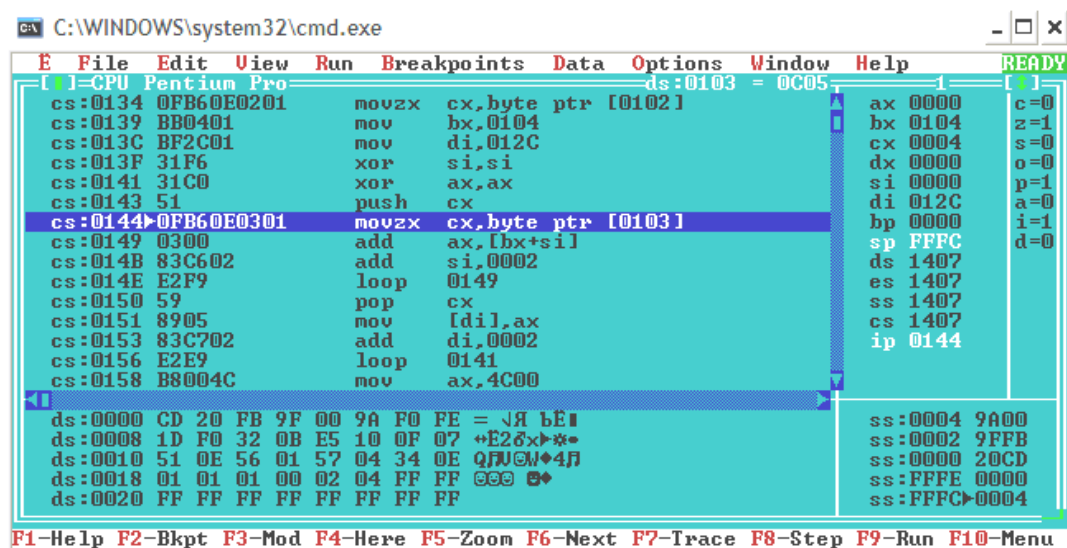
```

        dw 44,13,-1,99,32
sum rw 4                                ;Массив для сумм каждой строки
;-----
start:
    movzx cx,[n]                        ;Счётчик строк
    mov bx,table                        ;BX = адрес таблицы
    mov di,sum                          ;DI = адрес массива для сумм
    xor si,si                           ;SI = смещение элемента от начала таблицы
rows:
    xor ax,ax                           ;Обнуление AX. В AX будет считаться сумма
    push cx                             ;Сохранение значения CX
    movzx cx,[m]                        ;Инициализация CX для цикла по строке
calc_sum:
    add ax,[bx+si]                      ;Прибавление элемента строки
    add si,2                            ;SI = смещение следующего элемента
    loop calc_sum                       ;Цикл суммирования строки
    pop cx                              ;Восстановление значения CX
    mov [di],ax                         ;Сохранение суммы строки
    add di,2                            ;DI = адрес следующей ячейки для суммы строки
    loop rows                           ;Цикл по всем строкам таблицы
    mov ax,4C00h                        ;\
    int 21h                             ;/ Завершение программы

```

Turbo Debugger

В отладчике Turbo Debugger стек отображается в нижней правой области окна CPU. Левый столбец чисел — адреса, правый — данные. Треугольник указывает на вершину стека, то есть на тот адрес, который содержится в регистре SP. Если запустить программу в отладчике, то можно увидеть, как работают команды «push cx» и «pop cx».



Процедуры

Процедура представляет собой код, который может выполняться многократно и к которому можно обращаться из разных частей программы. Обычно процедуры предназначены для выполнения каких-то отдельных, законченных действий программы и поэтому их иногда называют подпрограммами. В других языках программирования процедуры могут называться функциями или методами, но, по сути, это всё одно и то же.

Команды CALL и RET

Для работы с процедурами предназначены команды CALL и RET. С помощью команды CALL выполняется *вызов* процедуры. Эта команда работает почти так же, как команда безусловного перехода (JMP), но с одним отличием — одновременно в стек сохраняется текущее значение регистра IP. Это позволяет потом вернуться к тому месту в коде, откуда была вызвана процедура. В качестве операнда указывается адрес перехода, который может быть непосредственным значением (меткой), 16-разрядным регистром (кроме сегментных) или ячейкой памяти, содержащей адрес.

Небольшой пример разных способов вызова процедуры:

```
use16                                ;Генерировать 16-битный код
org 100h                             ;Программа начинается с адреса 100h

mov ax,myproc
mov bx,myproc_addr
xor si,si
call myproc                          ;Вызов процедуры (адрес перехода - myproc)
call ax                             ;Вызов процедуры по адресу в AX
call [myproc_addr]                  ;Вызов процедуры по адресу в переменной
call word [bx+si]                   ;Более сложный способ задания адреса ;)

mov ax,4C00h                        ;\
int 21h                             ;/ Завершение программы
;Процедура, которая ничего не делает
myproc:
nop                                  ;Код процедуры
ret                                  ;Возврат из процедуры
;-----
myproc_addr dw myproc               ;Переменная с адресом процедуры
```

Возврат из процедуры выполняется командой RET. Эта команда восстанавливает значение из вершины стека в регистр IP. Таким образом, выполнение программы продолжается с команды, следующей сразу после команды CALL. Обычно код процедуры заканчивается этой командой. Команды CALL и RET не изменяют значения флагов (кроме некоторых особых случаев в защищенном режиме).

Ближние и дальние вызовы процедур

Существует 2 типа вызовов процедур. *Ближним* называется вызов процедуры, которая находится в текущем сегменте кода. *Дальний* вызов — это вызов процедуры в другом сегменте. Соответственно существуют 2 вида команды RET — для ближнего и дальнего возврата. Компилятор FASM автоматически определяет нужный тип машинной команды, поэтому в большинстве случаев не нужно об этом беспокоиться.

Передача параметров

Очень часто возникает необходимость передать процедуре какие-либо параметры. Например, если вы пишете процедуру для вычисления суммы элементов массива, удобно в качестве параметров передавать ей адрес массива и его размер. В таком случае одну и ту же процедуру можно будет использовать для разных массивов в вашей программе. Самый простой способ передать параметры — это поместить их в регистры перед вызовом процедуры.

Возвращаемое значение

Кроме передачи параметров, часто нужно получить какое-то значение из процедуры. Например, если процедура что-то вычисляет, хотелось бы получить результат вычисления. А если процедура что-то делает, то полезно узнать, завершилось действие успешно или возникла ошибка. Существуют разные способы возврата значения из процедуры, но самый часто используемый — это поместить значение в один из регистров. Обычно для этой цели используют регистры AL и AX. Хотя вы можете делать так, как вам больше нравится.

Сохранение регистров

Хорошим приёмом является сохранение регистров, которые процедура изменяет в ходе своего выполнения. Это позволяет вызывать процедуру из любой части кода и не беспокоиться, что значения в регистрах будут испорчены. Обычно регистры сохраняются в стеке с помощью команды PUSH, а перед возвратом из процедуры восстанавливаются командой POP. Естественно, восстанавливать их надо в обратном порядке. Примерно вот так:

```
myproc:
push bx                             ;Сохранение регистров
push cx
push si
...                                 ;Код процедуры
pop si                              ;Восстановление регистров
pop cx
pop bx
ret                                  ;Возврат из процедуры
```

Небольшое замечание по поводу использования отладчика. В Turbo Debugger нажимайте F7 («Trace into»), чтобы перейти к коду вызываемой процедуры. При нажатии F8 («Step over») процедура будет выполнена сразу целиком.

Для примера напишем процедуру для вывода сообщения в рамке и протестируем её работу, выведя несколько сообщений. В качестве параметра ей будет передаваться адрес строки в регистре BX. Строка должна

заканчиваться символом '\$'. Для упрощения процедуры можно разбить задачу на подзадачи и написать соответствующие процедуры. Прежде всего нужно вычислить длину строки, чтобы знать ширину рамки. Процедура *get_length* вычисляет длину строки (адрес передаётся также в BX) и возвращает её в регистре AX.

Для рисования горизонтальной линии из символов предназначена процедура *draw_line*. В DL передаётся код символа, а в CX — количество символов, которое необходимо вывести на экран. Эта процедура не возвращает никакого значения. Для вывода 2-х символов конца строки написана процедура *print_endline*. Она вызывается без параметров и тоже не возвращает никакого значения. Коды символов для рисования рамок можно узнать с помощью таблицы символов кодировки 866 или можно воспользоваться стандартной программой Windows «Таблица символов», выбрав шрифт Terminal.

```

use16                                ;Генерировать 16-битный код
org 100h                               ;Программа начинается с адреса 100h
jmp start                             ;Переход на метку start
;-----
msg1    db 'Hello!$'
msg2    db 'asmworld.ru$'
msg3    db 'Press any key...$'
;-----
start:
    mov bx,msg1
    call print_message    ;Вывод первого сообщения
    mov bx,msg2
    call print_message    ;Вывод второго сообщения
    mov bx,msg3
    call print_message    ;Вывод третьего сообщения
    mov ah,8              ;Ввод символа без эха
    int 21h
    mov ax,4C00h          ;\
    int 21h              ;/ Завершение программы
;Процедура вывода сообщения в рамке
;В BX передаётся адрес строки
print_message:
    push ax                ;Сохранение регистров
    push cx
    push dx
    call get_length        ;Вызов процедуры вычисления длины строки
    mov cx,ax            ;Копируем длину строки в CX
    mov ah,2              ;Функция DOS 02h - вывод символа
    mov dl,0xDA            ;Левый верхний угол
    int 21h
    mov dl,0xC4            ;Горизонтальная линия
    call draw_line         ;Вызов процедуры рисования линии
    mov dl,0xBF            ;Правый верхний угол
    int 21h
    call print_endline     ;Вызов процедуры вывода конца строки
    mov dl,0xB3            ;Вертикальная линия
    int 21h
    mov ah,9              ;Функция DOS 09h - вывод строки
    mov dx,bx            ;Адрес строки в DX
    int 21h
    mov ah,2              ;Функция DOS 02h - вывод символа
    mov dl,0xB3            ;Вертикальная линия
    int 21h
    call print_endline     ;Вызов процедуры вывода конца строки
    mov dl,0xC0            ;Левый нижний угол
    int 21h
    mov dl,0xC4            ;Горизонтальная линия
    call draw_line         ;Вызов процедуры рисования линии
    mov dl,0xD9            ;Правый нижний угол
    int 21h
    call print_endline     ;Вызов процедуры вывода конца строки
    pop dx                ;Восстановление регистров

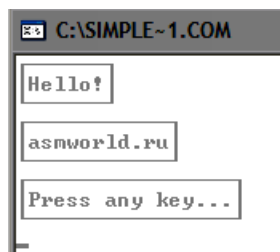
```

```

    pop cx
    pop ax
    ret                ;Возврат из процедуры
;Процедура вычисления длины строки (конец строки - символ '$').
;В BX передаётся адрес строки.
;Возвращает длину строки в регистре AX.
get_length:
    push bx            ;Сохранение регистра BX
    xor ax, ax         ;Обнуление AX
str_loop:
    cmp byte[bx], '$'  ;Проверка конца строки
    je str_end         ;Если конец строки, то выход из процедуры
    inc ax             ;Инкремент длины строки
    inc bx             ;Инкремент адреса
    jmp str_loop       ;Переход к началу цикла
str_end:
    pop bx             ;Восстановление регистра BX
    ret               ;Возврат из процедуры
;Процедура рисования линии из символов.
;В DL - символ, в CX - длина линии (кол-во символов)
draw_line:
    push ax            ;Сохранение регистров
    push cx
    mov ah, 2          ;Функция DOS 02h - вывод символа
drl_loop:
    int 21h           ;Обращение к функции DOS
    loop drl_loop     ;Команда цикла
    pop cx            ;Восстановление регистров
    pop ax
    ret               ;Возврат из процедуры
;Процедура вывода конца строки (CR+LF)
print_endline:
    push ax            ;Сохранение регистров
    push dx
    mov ah, 2          ;Функция DOS 02h - вывод символа
    mov dl, 13         ;Символ CR
    int 21h
    mov dl, 10         ;Символ LF
    int 21h
    pop dx            ;Восстановление регистров
    pop ax
    ret               ;Возврат из процедуры

```

И вот результат:



Отладчик Turbo Debugger

Небольшое замечание по поводу использования отладчика. В Turbo Debugger нажимайте F7 («Trace into»), чтобы перейти к коду вызываемой процедуры. При нажатии F8 («Step over») процедура будет выполнена сразу целиком.

Флаги

Флаги изменяются в результате выполнения арифметических и логических команд, а также команд сдвига. Регистр флагов можно сохранить в стек с помощью команды PUSHF и восстановить из стека с помощью команды POPF. Кроме того, в процессоре существуют специальные команды, которые позволяют явно установить или сбросить флаги CF, DF и IF. Это очень простые команды: у них нет операндов и результатом является только изменение значения соответствующего флага.

Флаг переноса CF

Команда CLC сбрасывает флаг CF.

Команда STC устанавливает флаг CF в единицу.

Команда CMC инвертирует значение флага CF.

Флаг направления DF

Этот флаг определяет направление обработки данных цепочечными командами (о них подробно расскажу в отдельной статье). Он должен устанавливаться или сбрасываться перед использованием этих команд.

Команда CLD сбрасывает флаг DF.

Команда STD устанавливает флаг DF в единицу.

Флаг прерывания IF

Этот флаг определяет, разрешены в данный момент прерывания или нет (о прерываниях тоже будет отдельная статья).

Команда CLI сбрасывает флаг IF (запрещает прерывания).

Команда STI устанавливает флаг IF в единицу (разрешает прерывания).

Команды LANF и SANF

Команда LANF загружает младший байт регистра флагов в АН. Её удобно использовать, когда нужно получить значения сразу нескольких флагов. Порядок расположения флагов представлен на рисунке:

7	6	5	4	3	2	1	0
SF	ZF	0	AF	0	PF	1	CF

Команда SANF выполняет обратную операцию — загружает содержимое АН в младший байт регистра флагов. Это позволяет одновременно изменить значения нескольких флагов. При этом биты 1, 3, 5 регистра АН игнорируются.

Пример программы

В качестве примера использования команды LANF дана процедура, которая получает и выводит на консоль значения флагов SF, ZF, AF, PF и CF. Подобную процедуру можно использовать в отладочных целях. Достаточно добавить её вызов в то место программы, где хочется проверить состояние флагов.

Чтобы выводить значения отдельных битов удобно написать отдельную процедуру для печати флага CF в виде символа. Нужный бит будет помещаться во флаг CF при помощи команды сдвига SHL.

;Процедура вывода значения флага CF в виде символа

`print_cf:`

`push ax`

`push dx`

`mov ah, 2`

`mov dl, '0'`

`adc dl, 0`

`int 21h`

`pop dx`

`pop ax`

`ret`

;Функция DOS 02h - вывод символа

;DL = '0'

;Если CF = 1, то в DL будет символ '1'

;Обращение к функции DOS

Значение последнего выдвигаемого бита копируется в флаг CF. Для сдвигов на 1 бит устанавливается флаг OF, если в результате сдвига изменяется знаковый бит операнда.

Отличительной особенностью FASM является очень гибкая и мощная поддержка макросов. Макросы — это шаблоны для генерации кода. Создав один раз макрос, можно использовать его во многих местах в коде программы. Макросы делают процесс программирования на ассемблере более приятным и простым, а код программы получается понятнее. Макросы позволяют расширять синтаксис ассемблера и даже добавлять собственные «команды», которых нет в процессоре.

Обработкой макросов занимается *препроцессор* FASM. Преобразование исходного кода в исполняемый код FASM выполняет в два этапа. Первый этап — *препроцессирование*, а второй — собственно *ассемблирование* или *компиляция*. На первом этапе происходит вычисление всех числовых выражений, вместо констант и названий меток подставляются их фактические значения, вместо макросов подставляется сгенерированный код. На втором этапе все данные и машинные команды преобразуются в соответствующие байты, и в результате получается исполняемый файл требуемого формата.

Синтаксис создания макроса

Для создания макроса используется директива *macro*. Эта директива имеет следующий синтаксис:

```
macro <название_макроса> [<список_параметров>]
{
    <тело_макроса>
}
```

Кстати, для коротких макросов можно писать всё в одной строке:

```
macro <название_макроса> [<список_параметров>] { <тело_макроса> }
```

Тело макроса – это код, который подставляется в то место, где макрос будет вызван. Создание макроса является, по сути, лишь его объявлением, в этом месте программы никакого кода сгенерировано не будет! Поэтому объявления макросов обычно размещают в самом начале программы или в отдельном файле.

В качестве примера рассмотрим простой макрос без параметров, который предназначен для завершения программы:

```
macro exit_app
{
    mov ax, 4C00h
    int 21h
}
```

После того как макрос объявлен, в нужном месте программы достаточно написать `exit_app`. Туда пре-процессор FASM автоматически подставит 2 команды, записанные в теле макроса.

Создадим ещё один полезный макрос, предназначенный для такой часто используемой операции, как вывод строки:

```
macro print_str str
{
    mov ah, 9
    mov dx, str
    int 21h
}
```

У этого макроса есть один параметр — адрес строки. При генерации кода вместо `str` будет подставлен тот параметр, который указан при вызове макроса. Внимание! Код будет генерироваться в месте каждого вызова макроса! В этом главное отличие макроса от процедуры. Код процедуры содержится в программе только в одном экземпляре, а вызывается она с помощью передачи управления командой `CALL`.

Добавим эти макросы в программу «hello, world!»

; Макрос выхода из программы

```
macro exit_app
{
    mov ax, 4C00h    ;Здесь только объявление макроса, код не генерируется
    int 21h
}
; Макрос вывода строки
macro print_str str
{
    mov ah, 9        ;Здесь тоже код не генерируется
    mov dx, str
    int 21h
}
```

```
use16                ;Генерировать 16-битный код
org 100h              ;Программа начинается с адреса 100h
    print_str hello ;Вывод строки
    exit_app        ;Выход из программы
hello db 'Hello, macro world!$'
```

В результате основная программа состоит всего из двух строчек, и это уже не похоже на ассемблер, это — макроассемблер FASM!

Расширение системы команд

Макросы можно использовать для расширения системы команд. Например, часто в программе приходится обнулять регистры. Создадим специальный макрос для этой цели:

; Макрос – команда обнуления регистра

```
macro clr reg { xor reg, reg }
```

Теперь обнулять регистры в программе можно так:

```
clr ax    ;AX=0
clr si    ;SI=0
clr bl    ;BL=0
```

Макросы с переменным количеством параметров

Возможно также создавать макросы с переменным количеством параметров. В этом случае имя параметра записывается в квадратных скобках. Для генерации кода макрос вызывается столько раз, сколько параметров ему было передано. Например, можно написать специальные макросы для улучшения команд `PUSH` и `POP`:

; Макрос – улучшенная команда push

```
macro push [arg] { push arg }
```

```
; Макрос – улучшенная команда pop
macro pop [arg] { pop arg }
```

Несмотря на то, что название макроса совпадает с именем команды, в теле макроса это имя считается именем команды (иначе получилась бы бесконечная рекурсия). Данные макросы позволяют помещать в стек и извлекать из стека сразу несколько операндов, что упрощает код программы. Пример использования:

```
push ax, word[si], 5
pop dx, cx, ax
```

Директива include

Можно написать собственный набор макросов и использовать их в своих программах. В этом случае удобно поместить макросы в отдельный файл и воспользоваться директивой включения файла *include*. Синтаксис директивы *include* очень прост:

```
include 'путь/к/файлу'
```

Путь к файлу указывается в одинарных кавычках и может быть относительным (по отношению к компилируемому файлу) или полным (начиная от буквы диска или корневого каталога системы). Если включаемый файл находится в той же папке, то достаточно указать только имя файла. Расширение файла может быть любым, но обычно используют «inc» или «asm».

Препроцессор FASM читает указанный файл и подставляет код из него вместо директивы *include*. В отдельный файл можно также вынести часто используемые процедуры, отдельные функциональные блоки программы или даже объявления данных.

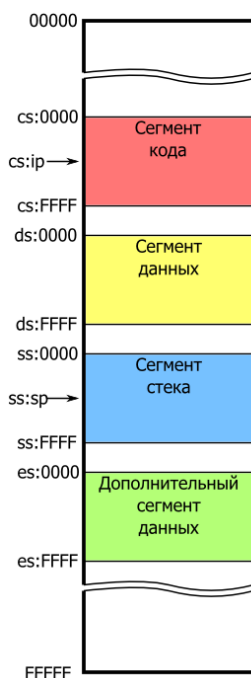
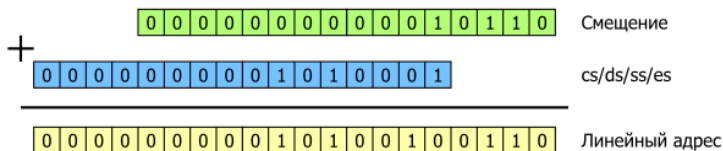
Если записать макросы в отдельный файл 'mymacro.inc', то программа «hello, world!» станет ещё короче:

```
include 'mymacro.inc'
use16 ;Генерировать 16-битный код
org 100h ;Программа начинается с адреса 100h

print_str hello ;Вывод строки
exit_app ;Выход из программы
;-----
hello db 'Hello, macro world!$'
```

Сегментная адресация

В заключение рассмотрим сегментную адресацию. Основная идея сегментной адресации в том, что адрес состоит из двух частей — сегментной и смещения. Обычно их записывают через двоеточие (например, 0100:0500). Линейный адрес любой ячейки памяти получается в результате сложения смещения и сегментной части, сдвинутой на 4 бита влево.



Начало сегмента всегда выровнено на границу параграфа (адрес кратен 16 байтам). Максимальный размер сегмента равен $2^{16} = 64$ КБайта. А всего можно адресовать $2^{20} = 1$ МБайт памяти. Конечно, сейчас такой объем памяти кажется смешным, но раньше это было очень много.

Одна из особенностей сегментной адресации — неоднозначность представления адреса. Допустим, требуется обратиться к ячейке памяти по адресу 00400. Этот адрес может быть представлен как 0000:0400, 0040:0000, 0020:0200 и т.д.

При таком способе адресации загруженная в память программа может одновременно работать с четырьмя сегментами — слева представлена типичная схема ОЗУ 8086 — она разделена на код, данные, стек и дополнительные данные. Помимо увеличения объема доступной ОЗУ — ведь адресуется она не линейно, а указанием сегмента и смещения в нем, это способствует безопасности и разделению кода и данных.

Однако сегменты могут перекрываться или даже совпадать, что запутывает программиста и может привести к трудноуловимым ошибкам с разными последствиями.

Для всех команд подразумевается сегмент «по умолчанию». Например, команды PUSH и POP работают с сегментом стека. Если операнд такой команды находится в памяти, то он берётся из сегмента данных. Команды JMP и LOOP вычисляют адрес перехода в сегменте кода.

Создание DOS EXE

Возможности сегментной адресации полностью реализуются (и нужны для создания) в исполняемом файле DOS EXE. Не путайте этот формат с исполняемым файлом Windows (PE EXE)! Расширение такое же, но файл имеет совершенно другую структуру.

```
format MZ ;Исполняемый файл DOS EXE (MZ EXE)
entry code_seg:start ;Точка входа
stack 200h ;Размер стека

;-----
segment data_seg ;Сегмент данных
hello db 'Hello, asmworld!$' ;Строка
;-----
segment code_seg ;Сегмент кода
start: ;Откуда начинается выполнение программы
    mov ax,data_seg ;\
    mov ds,ax ;/ Инициализация регистра DS
    mov ah,09h ;\
    mov dx,hello ; > Вывод строки
    int 21h ;/
    mov ax,4C00h ;\
    int 21h ;/ Завершение программы
```

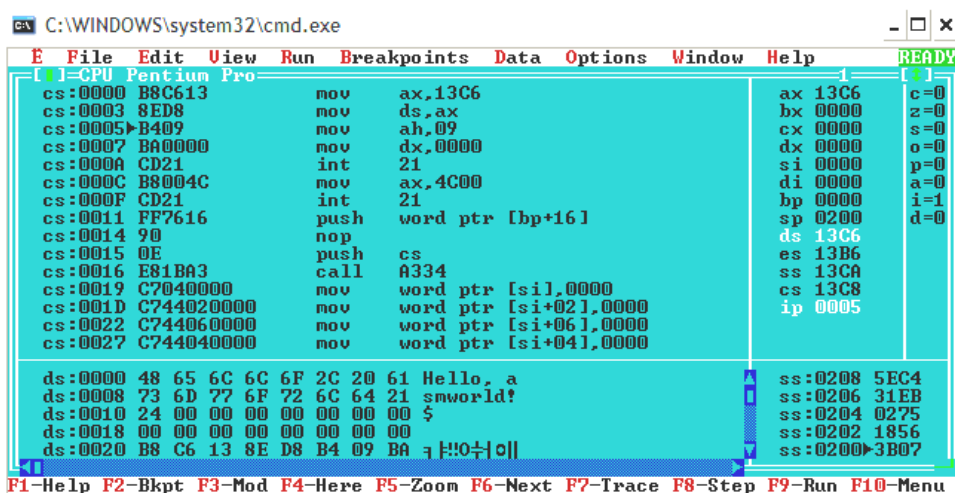
В первой строке после директивы *format* нужно поставить *MZ*, чтобы FASM сгенерировал нужный нам файл.

Во второй строке указывается точка входа, то есть метка, с которой начинается выполнение программы. Имя метки дополняется названием сегмента, в котором она находится.

После директивы *stack* можно указать требуемый размер сегмента стека в байтах (по умолчанию используется 4096). Далее файл состоит из сегментов, которые объявляются с помощью директивы *segment*. После директивы записывается название сегмента.

В приведенном примере файл состоит из двух сегментов. В первом находятся данные (а точнее, строка), во втором — код. Выполнение программы начинается с метки *start* в сегменте кода. После запуска необходимо инициализировать регистр *ds*, чтобы выбрать нужный сегмент данных. Для этого используются 2 команды, так как невозможно напрямую записать значение в сегментный регистр (нет команды *MOV ds, значение*).

Работу программы с сегментами можно увидеть в отладчике. Обратите внимание, что *cs*, *ds*, *es* и *ss* имеют разные значения:



Префиксы переопределения сегментов

Иногда нужно изменить используемый сегмент данных только для одной команды. Например, хочется прочитать байт из текущего сегмента кода или записать в сегмент стека. Для этого предназначены префиксы переопределения сегмента. Названия префиксов совпадают с названиями сегментных регистров.

Особенность синтаксиса FASM в том, что префикс пишется внутри квадратных скобок (так как по смыслу является частью адреса).

Дальние переходы, вызовы процедур и возвраты

Дальними (*far*) называются переходы в другой сегмент кода. При их выполнении меняется содержимое регистра *cs*. Они могут только безусловными. Ближние (*near*) переходы осуществляются в пределах одного сегмента. Аналогично есть дальние и ближние вызовы процедур, а также дальние и ближние возвраты.

Команда дальнего вызова процедуры сохраняет в стек не только *ip*, но и *cs*, чтобы можно было вернуться в текущий сегмент кода. Команда *RET* является синонимом ближнего возврата *RETN*. Дальний возврат осуществляется командой *RETF*. Она восстанавливает из стека регистры *ip* и *cs*.

Пример:

```
format MZ                      ;Исполняемый файл DOS EXE (MZ EXE)
entry seg1:start               ;Точка входа
;-----
segment seg1                   ;Сегмент первый
hello db 'Hello, asmworld!$'   ;Строка

start:                        ;Отсюда начинается выполнение программы
    push cs                    ;\
    pop ds                     ;/ Инициализация регистра DS
    jmp seg2:do_it

exit:
    mov ah,08h
    int 21h
    mov ax,4C00h               ;\
    int 21h                     ;/ Завершение программы
;-----
segment seg2                   ;Сегмент второй
do_it:
    mov dx,hello               ;DX = СМЕЩЕНИЕ строки в seg1
    call seg3:print_str        ;Дальний вызов (cs,ip в стек)
    jmp seg1:exit              ;Дальний переход
;-----
segment seg3
; Дальняя процедура для вывода строки (ds:dx = адрес строки)
print_str:
    mov ah,09h
    int 21h
    retf                       Дальний возврат (восстанавливает ip,cs)
```

Программа состоит из трёх сегментов. Сначала выполняется переход во второй, затем вызов процедуры в третьем сегменте. Кстати, сегмент может содержать код и данные вместе — мы поместили строку в начало первого сегмента.

3.3.2. Теговые машины

Возвращаемся к описанию теговых машин. Идея теговых компьютеров, упрощенно, заключается в создании архитектуры, способной аппаратно поддерживать в ограниченном объеме языки высокого уровня, вместо ассемблера. По идее разработчиков это должно было упростить программирование и повысить удобство, надежность и быстродействие, но, погнавшись за такой толпой зайцев, было бы удивительно поймать хоть одного.

Как известно, в различных архитектурах типы данных определяются программно, что приводит к необходимости использовать разные команды для обработки различных типов. Принцип теговой архитектуры заключается в самоопределении данных, то есть каждая ячейка памяти имеет специальное дополнительное поле (тег), которое служит для описания атрибутов его содержимого, что позволяет получить инвариантность команд к типам используемых данных. Это приводит к сокращению числа команд и превращению их в более интеллектуальные и универсальные – программист пишет *A+B*, не задумываясь, что это такое, числа, матрицы и т.п., а машина сама определяет требуемую операцию, исходя из контекста – содержимого тегов.

Таким образом, сокращается число и длина команд, определение типа данных откладывается на момент выборки этих данных, а не на момент выборки команды. В расширенном теге задается не только тип данных, но и длины операнда, занятости ячейки и т.д. с использованием дополнительных флагов, среди которых:

- Бит занятости. Определяет, свободна ячейка или занята.
- Бит захвата. Необходимость выполнения прерывания при доступе к ячейке.

- Биты защиты – определяет, какие обращения допустимы к этой «порции» данных (только чтение, только запись, чтение/запись).

- Длина данных.

В результате мы получаем огромное количество преимуществ: компактную и простую систему команд, аппаратный строгий контроль типов, элементарные компиляторы, меньший объем кода, и, самое главное, – уменьшение нагрузки шины ОЗУ-процессор. Также, в перспективе, теговые ЭВМ позволили бы писать легко переносимый между соответствующими платформами код.

Естественно, получить все эти прекрасные вещи можно не бесплатно. К ним комплектом идут снижение выразительных способностей языка из-за сильной типизации, с которой ничего не поделаешь – встроена в процессор, колоссальное усложнение блока дешифрации команды – ведь в момент ее получения необходимо прочитать все теги, огромный расход памяти на хранение теговой информации. Как результат, мы имеем очень сложный и дорогой компьютер с не лучшим быстродействием, но который удобно программировать и с хорошей защитой от ошибок. Конечно, хорошо быть умными, после того как эксперименты в железе неоднократно подтвердили эти выводы. Разработчиков же, что Burroughs, что «Эльбрус» извиняет то, что для них все это было впервые, тем более, для своего времени это были выдающиеся машины.

Фирма Burroughs Corporation относилась к одной из старейших, наряду с IBM, компаний мира, производящих вычислительную технику. Она была основана Уильямом Берроузом (дедом и тезкой известного писателя-битника) еще в 1886 г. как American Arithmometer Company.

В 1952 г. именно Burroughs создаёт память для знаменитого ENIAC. Через год одним из важных направлений её деятельности становится разработка суперкомпьютеров. Именно они впервые задумались над аппаратной реализацией в ЭВМ языка программирования высокого уровня, и в качестве такого языка был в конце 60-х годов выбран ALGOL-68 — вероятно, наиболее мощный из всех структурных языков тех лет. Что важно, ALGOL-68 был изначально рассчитан на распараллеливание вычислений. Язык был настолько мощным, что его реализации на неспециализированных системах (реализацией ALGOL-68 на ЕС ЭВМ занимался, например, Ленинградский государственный университет) не получили распространения, как слишком медленные. ALGOL-68 был вершиной структурного программирования, хотя ему на смену уже приходили объектно-ориентированные технологии. Как IBM, Burroughs пытались поставлять полную линейку продуктов для своих клиентов, в том числе собственной разработки принтеры, жесткие диски, ленточные накопители, бумагу для печати и даже ленты для пишущих машинок.

Машины Burroughs были очень мощными, но сложными и очень дорогими, на заводах фирмы к концу 1970-х трудилось больше 20.000 человек, но конкурировать с IBM, к тому времени захватившей почти весь международный рынок, было очень трудно, и в 1986 г. компания была поглощена UNISYS.

В СССР проблема суперкомпьютерных вычислений в 1970-е стояла очень остро. Машины ЕС, основанные на допотопной S/360 уже устарели, закупить суперкомпьютеры Cray не удалось никакими усилиями⁶⁰ – компания мастерски уходила от поставок своих машин в любую страну, дружественную соцлагерю, Burroughs же находилась на грани краха и не была столь идеологически щепетильна. И так, через Восточную Европу, в Союз было завезено несколько образцов их теговых машин.

Интерес к супер-ЭВМ у нас возник не просто так – в начале 1970-х г. они срочно понадобились для обновления системы ПРО Москвы. Военными разработками у нас традиционно занимался Институт точной механики и вычислительной техники Академии наук СССР (ИТМ и ВТ), где успели поработать в свое время почти все самые лучшие специалисты страны в области компьютеров, включая легендарного Лебедева. Ведущие сотрудники ИТМ и ВТ увлекались горнолыжным спортом, поэтому название у проекта появилось сразу – «Эльбрус».

В 1968 ИТМиВТ взяли заказ на разработку вычислительных средств для ЗРК С-300 и успешно решили задачу, построив трехпроцессорную ЭВМ 5Э26. Занимая объем около 2 м³, 5Э26 обладала производительностью на уровне БЭСМ-6 – около 1 млн. оп./с с фиксированной запятой. Этот комплекс был сдан в 1972–1974 гг. Следующим этапом было создание второго поколения противоракетного комплекса. Возглавлявший эту работу Г.В. Кисунько поставил задачу – разработать вычислительную систему производительностью 100 млн. оп./с., но в 1969 году это было невозможно.

Для техники СССР предел производительности тогда составлял ~1 млн. оп./с на один процессор, за рубежом ~5 млн. оп./с. Тогда родилась идея использовать многопроцессорный комплекс. «Эльбрус» заложили в 1970 г., изучив не только Burroughs, но и проект Манчестерского университета MU-5 и ОС Multics (General Electric). Взяв лучшие схемотехнические решения и увеличив число процессоров до 10, на основе 5Э26 инженеры построили «Эльбрус-1». Несмотря на работу на дискретных элементах с колоссальным по сегодняшним меркам временем задержки, производительность составила ~15 млн. оп./с. Машина была достроена и сдана заказчику в 1980 г. и установлена в системе ПРО Москвы.

К сожалению, уже в процессе постройки «Эльбрус-1» очень сильно устарел. Тогда же было принято решение собирать новую машину на интегральных схемах ИС-100 – по традиции скопированных, но не с Intel, а с более экзотической архитектуры – Motorola 10k. Делая все неофициально, без спецификаций, чисто

⁶⁰ Суперкомпьютеры Cray тоже пытались скопировать (так и не законченная БЭСМ-10), но без живого образца работа не сулила быстрого успеха и была передана в другую организацию, где вяло двигалась вплоть до развала СССР.

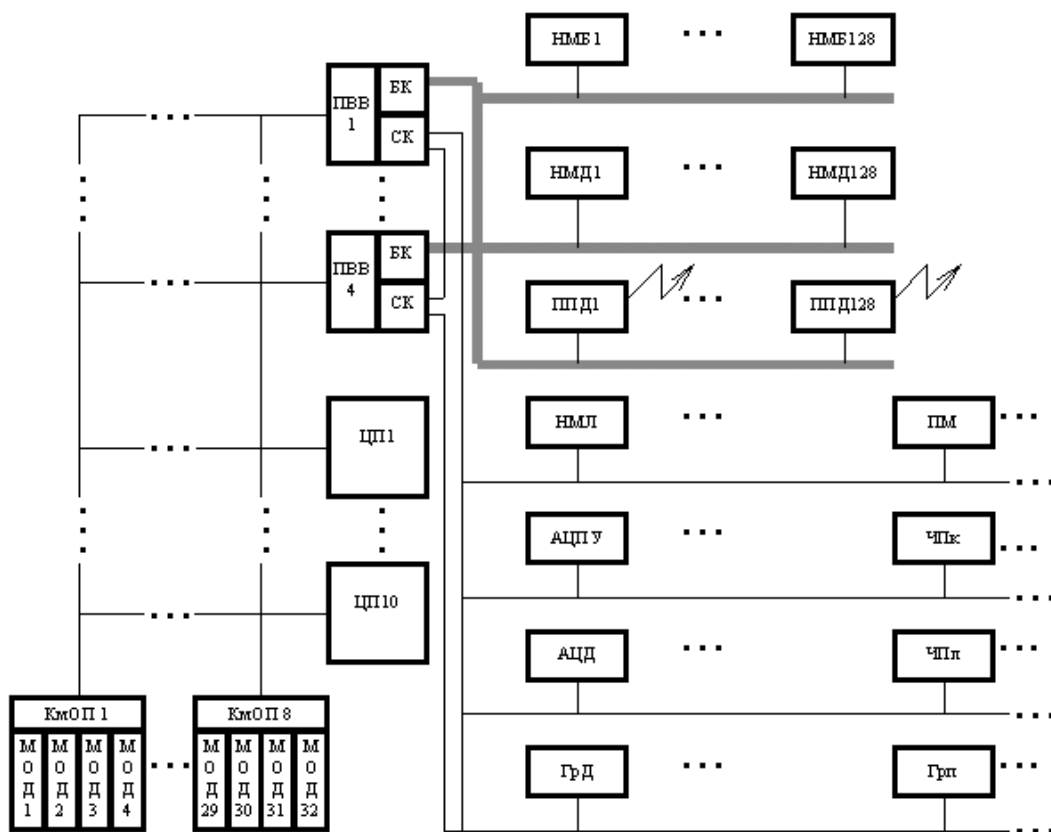
методом реверс-инжиниринга было очень сложно восстановить работу такой нетривиальной схемы. Вообще, копирование процессора – это целая история. Во времена, когда они были достаточно простые для такой процедуры, по воспоминаниям отечественных разработчиков, они просто снимали слой за слоем, изучая чип в микроскоп и пытаясь непосредственно скопировать увиденную схему, воспроизводя ее вместе со всеми потенциальными ошибками, недоработками и прочим. В результате «Эльбрус-2» довели до ума еще 5 лет и сдали только к 1985 г., получив вожделенные 125 млн. оп./с на восьми процессорах из десяти – два считались резервными. Но западные суперкомпьютеры к тому времени перешагнули порог 1 млрд. оп./с. К этому времени отечественным инженерам уже удалось ознакомиться со ставшим мифическим Cray и следующий проект планировали собрать на клонах его векторных процессоров, однако, началась перестройка и, что случилось дальше, всем известно.

Теперь о характеристиках того, что удалось построить. Как уже упоминалось, одним из принципиальных моментов разработки «Эльбрус-2» являлось создание архитектуры, позволяющей достичь значительного упрощения системы математического обеспечения за счет приближения аппаратных возможностей к нуждам программирования. Здесь имеется в виду стековая память и работа не на языке ассемблера, а на собственном ЯВУ, названном Автокод «Эль-76».

Система команд и внутренняя структура, включающая универсальный стековый механизм, приспособлены для программирования на языках высокого уровня и дают возможности рекурсивного использования процедур, работу нескольких пользователей с общими данными, динамическое распределение и защиту памяти и разветвленную систему прерываний. Машины Burroughs, кстати, также считались самыми безопасными и надежными и по тем же причинам. Уровень заслуженного доверия к ним был так высок, что задолго до появления хакеров озабочившаяся вопросами компьютерной безопасности служба Федеральной резервной системы США закупила их массово для своих банков.

Вот схема процессорного модуля «Эльбрус». Параллельная обработка данных реализуется на двух уровнях — параллельных процессов и параллельного выполнения независимых команд. Первый уровень, по сути, многопроцессорная машина, второй – конвейер в процессоре.

В процессор входят: контроллер ОЗУ (УООП), буфер команд (БК), дешифратор команд (УУ), 10 специализированных АЛУ (ИУ), регистры преобразования относительных адресов в абсолютные (БР), стек операндов (СтОп) для хранения промежуточных результатов операций и кэш (БП).



Общая схема «Эльбрус-2»

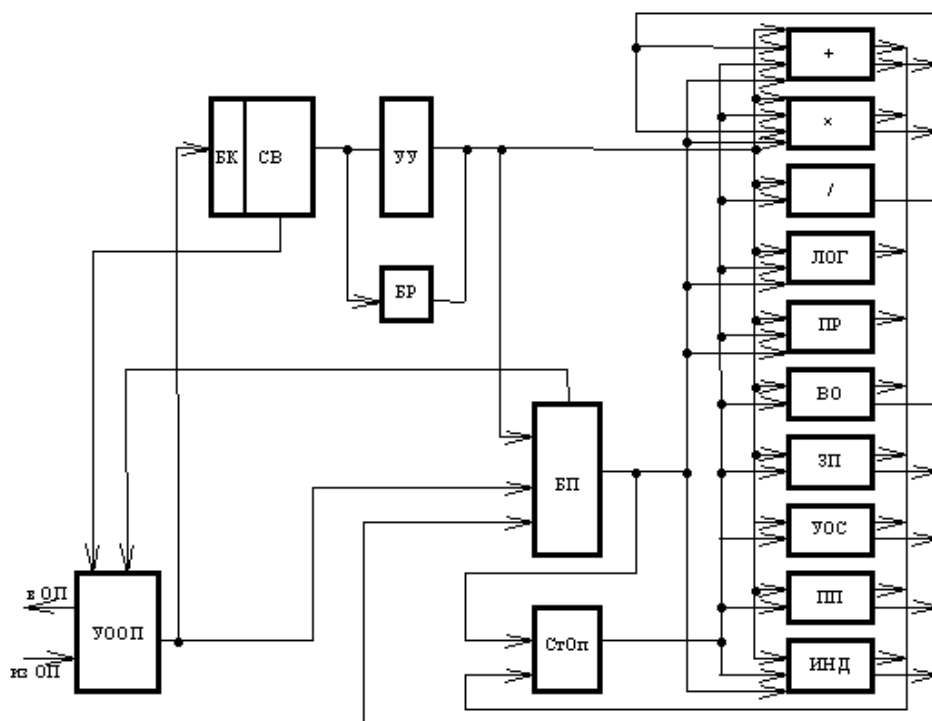


Схема теплового стекового процессора «Эльбрус-2»

Из-за высокой потребляемой мощности ЭВМ требовала мощную жидкостную систему охлаждения. Всего было выпущено порядка 30 машин «Эльбрус-2», причем некоторые из них работают до сих пор (!), включая машины Центра управления полетами.

Ну и где же миф, спросите вы? Миф как раз начинается. Как уже упоминалось, в 1986 г. все работы по векторному «Эльбрус-3» были свернуты из-за того, что один из помощников главного разработчика «Эльбрус-1,2» В.С. Бурцева⁶¹, некий Б. А. Бабаян решил провернуть, то, что делали многие во время краха СССР – «половить рыбу в мутной воде» и заработать на этом. Он решил воспользоваться тем, что по инерции разработчикам еще выделялись огромные суммы, но контроля над их тратами, уже не было.

Вот как это случилось, в описании непосредственных участников событий. ИТМ и ВТ тогда как раз разбирались с доставшейся им архитектурой Cray и планировал построить третье поколение «Эльбрус» на векторных процессорах. Помимо этого, в работе находились «Электроника ССБИС», проект модульного конвейерного процессора (МКП) разработки А.А. Соколова и несколько других систем. Однако в те же годы фирма Intel совместно с HP начала работу над процессором архитектуры VLIW, в конце концов, превратившемся в проект Itanium.

Об этом интересном, хоть и архитектурно неудачном решении еще пойдет речь. Однако, пользуясь тем, что в те годы в СССР «новое западное» означало автоматически «лучшее», Бабаян с коллегой Г.Г. Рябовым на очередном заседании АН выступил с резкой критикой векторных машин и предложил строить сразу VLIW-архитектуру под своим руководством. Надо отметить, что реальную сложность таких машин в те годы не представляла даже Intel. Их проект увенчался хоть каким-то успехом лишь после 12 лет (!) разработки, о том же, чтобы сделать что-то подобное в разваливающемся на глазах СССР и речи быть не могло. Тем не менее, академики дали себя убедить, конструктор первых «Эльбрусов» Бурцев был сослан из ИТМ и ВТ в лабораторию Г.И. Марчука, пост директора Института получил Рябов, который немедленно выбил для напарника колоссальные суммы на строительство VLIW-машины «Эльбрус-3», естественно, возглавляемое Бабаяном.

Остальные проекты были им закрыты в весьма интересной форме. Например, МКП имел возможность подключения процессоров с различной специализацией (радиолокационная обработка, структурная обработка, быстрые преобразования Фурье и т.д.), у него было несколько счетчиков команд, поэтому он мог работать с 4 потоками команд одновременно. Г.Г. Рябов, никого не предупредив, представил Госкомиссии лишь наполовину законченный проект, Бурцева и Соколова на представление, естественно, не пригласили. Комиссия, оценив наполовину готовую машину, вынесла вердикт – для серийного производства не годен, отправить в утиль. Бурцев обратился к бывшему президенту АН Г.И. Марчуку, после этого Вычислительный центр коллективного пользования (ВЦКП), который он тогда возглавил, получил 100.000 р. на доводку МКП. Счастье длилось недолго, до нового Президиума АН, где уже известными товарищами в рубрике «Разное»

⁶¹ Тот самый знаменитый теоретик dataflow-вычислений.

поднимается вопрос уже о полном расформировании (!) всего ВЦКП. Бурцева на заседание снова не приглашают, центр расформирован, деньги отбиты обратно Бабаяну на разработку VLIW-машины.

Судьба «Электроника ССБИС» еще печальнее. Институт проблем кибернетики был также ликвидирован под предлогом недостатка средств, и 4 готовые машины просто разобрали на металлолом, с целью извлечь золото и серебро из микросхем. Неутомимый Бурцев не сдался, на вырученные от продажи микросхем деньги они продолжили работу над еще более интересной разработкой, действительно оригинального направления – оптическим компьютером, организовав для этого новый Институт высокопроизводительных вычислительных систем (ИБВС).

Однако здоровье его было уже подорвано, в 1998 ему исполнился 71 год, и он был вынужден покинуть пост директора ИБВС. Несложно догадаться, кто занял его место – все тот же Б.А. Бабаян. Группа разработчиков ОЭВМ была немедленно изгнана из ИБВС в Институт проблем информатики, но все их оборудование, включая даже персональные компьютеры с САПР Mentor Graphics, на которых был проект, Бабаян забрать не разрешил. Работы в этом направлении вяло продвигаются и сейчас, но вероятность их успешного завершения чрезвычайно мала.

Что же случилось с VLIW-Эльбрусом? А ничего, в буквальном смысле слова. Прототип огромного компьютера был построен еще в 1988, но вплоть до 1994 его не удалось даже запустить – VLIW-архитектура, да еще и на дискретных схемах оказалась безумно сложной штукой. После того как разработчики из Sun привезли в Москву свой чип SPARC и продемонстрировали академиком, как один кристалл обладает функциональностью и производительностью, превышающей весь «Эльбрус-3», занимающий 5 м² и так и не заработавший, в 1994 проект был свернут, и прототип отправлен под пресс. Судьба улетевших в никуда трех миллиардов рублей, затраченных за 6 лет на разработку, деликатно не упоминалась.

Дальше начинаются чудеса. Увидев микропроцессор, Бабаян загорелся желанием сделать такой же. В результате, для удобства распила, было организовано сразу 2 компании – МЦСТ (Московский Центр Спарк Технологий), которая получила лицензию на копирование архитектуры Sun (а она свободная и бесплатная, SPARC процессоры мы еще обсудим в соответствующей части) и группа «Эльбрус».

Они занимались разработкой VLIW-микропроцессора, в общем-то, с очевидными результатами. В процессе родился «Эльбрус-90» – отечественный клон SPARC, без каких-либо ухищрений, но попытки изготовить его в Зеленограде провалились – к этому времени материально-техническое оснащение нашего процессорного производства не вытянуло бы, увы, и чипы 80-х годов, не говоря уже о современных архитектурах. Опытные образцы были изготовлены с таким количеством дефектов, что компания Sun в ужасе отозвала заказ и не приняла разработку, на этом наше сотрудничество с ними и закончилось. Бабаян же, одержимый VLIW-машинами, вложил в следующий проект – Elbrus E2k или Эльбрус 2000 – так и не состоявшийся «убийца Itanium».

Конец его был очевиден, но спустя более 10 лет заказчики (по традиции, Министерство обороны) все-таки начали робко интересоваться, куда уходят миллионы, и скоро ли Бабаян изготовит им обещанный чудо-процессор. Надо отметить, что пробная партия Эльбрус-90 была отдана армии, благо, учитывая ее общее состояние, на их качество никто и не смотрел.

Начиная с 2001 г. дело у МЦСТ пошло на лад, т.к. для компьютеров, названных в честь процессора «Эльбрус-90микро»⁶², кремниевые пластины стали производить в Израиле на фабрике Tower Semiconductor, а сборкой и тестированием процессоров занимается компания тайваньская ASE, принадлежащая AMD. Остается, конечно, вопрос, что русского (кроме улетающих как в трубу денег) в процессоре, у которого архитектура США, производство Израиль, а сборка Тайвань, но этот вопрос относится к категории философских, как и многие подобные в России.

Тем не менее, это был все-таки не обещанный «русский суперпроцессор». Что бы запутать след, МЦСТ несколько раз переименовывала Эльбрус 2000, сначала в ВК Эльбрус-3М1, потом в Эльбрус-S, потом в Эльбрус-2С+, заказала пробную партию чипов там же, на Тайване, и, в результате, в июле 2008 года, после более чем 10 лет разработки(!) свершилось чудо: процессор впервые показали прессе. Более того, при рабочей частоте 300 МГц в режиме совместимости с IA-32 он обогнал 500 МГц Intel Pentium III⁶³ в тестах SPEC.

Работы в этом направлении ведутся с тем же успехом до сих пор. ВК Эльбрус-3М1, вроде как, даже принят на вооружение, а в его вариант Эльбрус-2С+ уже уместили второе ядро. Конечно, военные заказы это всегда хорошо, но необходимо ориентироваться и на обычный рынок. В результате МЦСТ заключила союз с



Тот самый, армейский, Эльбрус-90микро

⁶² сам процессор был переименован в скромный МЦСТ R150

⁶³ который всего-то 1999 г. выпуска

известным отечественным сборщиком компьютеров Kraftway, известным мастерским продвижением собранных ими системных блоков во все государственные учреждения. Результатом союза стала miniITX-плата Monocube с процессором Эльбрус-2С+. Получившийся моноблок был представлен в 2012-2013 гг. Насчет производства пока ничего не ясно, ОС, естественно, перелицованный Linux с названием ОС Эльбрус. Цена за miniITX-систему уровня Р III по первым сведениям около 30-40 т.р. В результате можем констатировать, что с точки зрения освоения финансов Б.А. Бабаян безусловно добился заслуживающих зависти успехов, чего не скажешь о разработке архитектуры.

Что касается собственно оригинального «Эльбрус-2», то, как уже говорилось, он унаследовал все пороки своего американского прототипа, про который сами американские инженеры говорили, что его дизайн не сравнится с простыми и эффективными векторными решениями Сеймура Крэя. К тому же он был не пригоден для массового производства в принципе – каждая плата в нем должна вручную настраиваться по задержкам, и все межблочные провода так же должны иметь одинаковую задержку, подбираемую вручную.

«Эльбрус» решал следующую задачу – выжать максимум производительности и отказоустойчивости из низкокачественного, устаревшего и ненадежного железа, принеся в жертву все остальное – стоимость разработки, стоимость железа, сложность сборки и наладки. Это была не коммерческая, а государственная задача, в процессе решения которой большая толпа ученых, практически не ограниченных в финансировании, получили массу ценного опыта. Вот воспоминания одного из современных разработчиков ПО, ознакомившегося с Автокодом Эльбрус: «Помню - меня убила работа с массивами. Переключаться в режим супервизора, чтобы аллоцировать массив - это как, нормально? Это вообще нормально, что исполнительный конвейер знает про массивы? Работа с массивами через дескриптор - это что, эффективно? Выход за границы типа так быстрее проверять, да? Страшно представить, как этот ужас вообще на аппаратуру ляжет. Впрочем, тогда была другая раскладка с латентностью и скоростью работы памяти и прочих компонент - совсем не такая как сейчас. Она могла оправдывать такие смелые ходы, но такие дизайны не живут, по-любому. Собственно, они и не выжили».

Теоретически разработчики чистых теговых машин отталкивались от того, что в середине 1970-х еще не было архитектур и компиляторов, способных хоть на какое-то автоматическое распараллеливание кода, в результате чего большая часть многопроцессорных систем не могла быть эффективно загружена полностью и исполнительные блоки часто простаивали. Выходом из этого тупика стала суперскалярная архитектура или VLIW-машины, но до них еще было далеко, (хотя первый суперскалярный процессор был применен тем же Крэем в CDC 6600 еще в 1965 г.). Так и родилась идея облегчить труд программиста, переведя архитектуру на ЯВУ.

Однако стоит отметить, что на стековой архитектуре хороший суперскаляр сделать вообще не просто – для систем команд RISC он делается гораздо проще. Давайте посмотрим⁶⁴, что за суперскаляр в «Эльбрус-2»:

«Темп обработки команд в устройстве управления может колебаться от двух команд за 1 такт до одной команды за 3 такта. С максимальным темпом обрабатываются наиболее распространенные сочетания команд:

считать величину и арифметическую команду;

загрузить адрес и взять элемент массива;

загрузить адрес и записать.

В результате мы имеем суперскаляр на две инструкции за такт, причем примитивнейшие инструкции. Гордиться тут нечем, хорошо хоть чтение данных умеют на арифметику накладывать (при попадании в кэш)».

Альтернативными ЯВУ-машинами была серия МИР (Машина для Инженерных Расчётов), созданная Институтом кибернетики Академии наук Украины под руководством академика В. М. Глушкова в 1960-е годы. Оригинальная машина МИР была одной из первых в мире персональных ЭВМ, она выпускалась серийно и предназначалась для использования в учебных заведениях, инженерных бюро, научных организациях. В 1968 году МИР была модернизирована и получила название МИР-1.

МИР использовала BCD-систему, числа могли быть представлены как целые десятичные со знаком, с десятичным порядком и с плавающей запятой. Действия могут выполняться с числами произвольной разрядности и произвольной длины, ограниченные только объемом памяти в 4096 символов. Время на выполнение операции сложения — 50 мкс. Среднее быстроедействие — около 1—2 тыс. оп/с. В комплект машины входила электрическая печатная машинка Zoemtron для ввода и вывода информации со скоростью 7 знаков в секунду.

Управление машиной было организовано на более рациональном, чем у «Эльбруса», микропрограммном принципе. Микропрограммирование позволило поднять семантический уровень машинного языка и довести его до высокоуровневого языка программирования. Фактически микропрограммами выполнялось большинство арифметических действий, вычисление элементарных функций выполнялось перед трансляцией и интерпретацией входной программы. Микрокоманды машины МИР-1 — 120-разрядные и записывались на сменных микропрограммных матрицах. Это позволяло довольно сильно изменять характер использования машины, набор арифметических и логических операций, которые она может выполнять. Входной язык ма-

⁶⁴ <http://www.icmm.ru/~masich/win/lexion/elbrus2/elbrus2.htm>

шины — АЛМИР-65. Разработан коллективом в составе В. М. Глушкова, А. А. Стогния, А. А. Летичевского, В. П. Клименко, А. А. Дородницыной и др. В МИР-2 и МИР-3 используется входной язык Аналитик, сохраняющий совместимость. Аналитик развивал концепции АЛМИР-65 и дополнительно позволял непосредственно формулировать задания с аналитическими преобразованиями формул, получать аналитические выражения для производных и интегралов, что делало МИР машиной, пригодной для аналитических расчетов, по типу современных пакетов Matlab и Mathematics.

В 1967 году на выставке в Лондоне, где демонстрировалась МИР-1, она была куплена фирмой IBM — первая (и единственная) покупка советской электронной машины американской компанией. Как выяснилось позже, американцы купили машину не столько для того, чтобы считать на ней, сколько для того, чтобы доказать своим конкурентам, запатентовавшим в 1963 г. принцип ступенчатого микропрограммирования, что русские давно об этом принципе знали и реализовали в серийно выпускаемой машине. При этом реально он был применен В.М. Глушковым даже раньше — в ЭВМ «Промінь».

МИР-2 успешно соревновалась с универсальными ЭВМ обычной структуры, превосходящими ее по номинальному быстродействию и объему памяти в сотни раз. На этой машине впервые в практике отечественного математического машиностроения был реализован диалоговый режим работы, использующий дисплей со световым пером.



МИР-2

Проект МИР был закрыт, как и многие другие, в 1969 г. с началом копирования IBM S/360.

3.3.3. Ассоциативные ЭВМ

Еще одной необычной архитектурой, родственной тековым и потоковым машинам, является ассоциативная ЭВМ. Ассоциативный способ обработки данных позволяет преодолеть многие ограничения, присущие адресному доступу к памяти, за счет задания некоторого критерия отбора и проведения необходимых преобразований, только над теми данными, которые удовлетворяют этому критерию. Критерием отбора может быть совпадение с любым элементом данных, достаточным для выделения искомых данных из всех имеющихся. Поиск данных может происходить по фрагменту, имеющему большую или меньшую корреляцию с заданным элементом данных.

С течением времени были исследованы и в разной степени применяются несколько подходов, различающихся полнотой реализации модели ассоциативной обработки. Если реализуется только ассоциативная выборка данных с последующим поочередным использованием найденных данных, то говорят об ассоциативной памяти, или памяти, адресуемой по содержанию. При достаточно полной реализации всех свойств ассоциативной обработки используется термин «ассоциативный процессор».

Эти системы включают большое число операционных устройств, способных одновременно по командам управляющего устройства вести обработку нескольких потоков данных. В ассоциативных вычислительных системах информация на обработку поступает от ассоциативных запоминающих устройств (АЗУ), характеризующихся тем, что информация в них выбирается не по определенному адресу, а по ее содержанию.

Грубо говоря, ассоциативный процессор (АП) — это ассоциативная память, допускающая параллельную запись во все ячейки, для которых было зафиксировано совпадение с ассоциативным признаком. Эта особенность АП, носящая название мультизаписи, является первым отличием ассоциативного процессора от традиционной ассоциативной памяти. Считывание и запись информации могут производиться по двум срезам запоминающего массива — либо это все разряды одного слова, либо один и тот же разряд всех слов. При необходимости выделения отдельных разрядов среза лишние позиции допустимо маскировать. Каждый разряд среза в АП снабжен собственным процессорным элементом, что позволяет между считыванием информации и ее записью производить необходимую обработку, то есть параллельно выполнять операции арифметического сложения, поиска, а также эмулировать многие черты матричных ВС. Пример ассоциативной ВС — система STARAN, разработанная Goodyear Aerospace Corporation в 1972 г.

Давайте посмотрим, как работает ассоциативная память. По сути, это массив ограниченного размера из обычных строк памяти, но содержащих следующие значения – адрес A , данные D и надежность R . Надежность – это число от 0 до 1, при записи новая строка ADR заносится на место одной из наименее надежных строк. Ассоциативная память характеризуется тем, что чтение из нее возможно, даже если в ней нет строки с нужным адресом A , так как результат чтения из памяти есть вероятностная сумма всех данных, рассчитанная по их надежности R и близости адреса A к заданному адресу. Можно определять близость адреса сортировкой, можно по количеству отличающихся бит или байт.

Идея ассоциативной памяти идеально легла на понятие кэша – ADR выборка оказалась очень удобной при решении – какую строку вытеснять из памяти. Поиск нужной страницы в многоуровневой таблице страниц, требующий несколько обращений к ОЗУ при преобразовании виртуального адреса в физический, занимает много времени. Однако в соответствии со свойством локальности большинство программ в течение некоторого промежутка времени делают ссылки к небольшому числу страниц, таким образом, только небольшая часть таблицы страниц работает напряженно.

Естественное решение – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативная память, иногда также употребляют термин ассоциативные регистры (иногда *translation lookaside buffer (TLB)*).

Одна запись в таблице в ассоциативной памяти содержит информацию про одну виртуальную страницу, ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц. Отображение виртуальных страниц, хранимых в ассоциативной памяти, осуществляется быстро, однако кэш-память является дорогостоящей и имеет ограниченный размер. Число записей в TLB обычно невелико – от 8 до 2048. Такая память является ассоциативной, потому что в отличие от таблицы страниц, которая проиндексирована по номерам виртуальных страниц, здесь происходит одновременное сравнение номера виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти. Вначале он ищет виртуальную страницу в ассоциативной памяти. Если страница найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется. Если страницы нет в ассоциативной памяти, то она ищется через таблицу страниц. Происходит замена одной из страниц в ассоциативной памяти найденной страницей. В таблице такая загруженная страница помечается битом модификации, что будет учтено при следующей загрузке ассоциативной памяти из таблицы страниц. Процент, когда номер страницы находится в ассоциативной памяти, называется *hit (совпадение) ratio* (пропорция, отношение). Таким образом, *hit ratio* – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает *hit ratio*.

Предположим, что для доступа к таблице страниц необходимо 100 нс, а для доступа к ассоциативной памяти – 20 нс. С 90% *hit ratio* среднее время доступа – $0.9 \cdot 20 + 0.1 \cdot 100 = 28$ нс. Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Библиографический список

1. Акритас, А. Основы компьютерной алгебры с приложениями: пер. с англ. / А. Акритас. – М.: Мир, 1994. – 544 с.
2. Акушский, И.Я. Машинная арифметика в остаточных классах / И.Я.Акушский, Д.И.Юдицкий. – М.: Сов. Радио, 1968. – 440 с.
3. Кушнеров, А. Троичная цифровая техника. Перспектива и современность [Электронный ресурс] / Университет им. Бен-Гуриона, Беэр-Шева, Израиль; А. Кушнеров – Режим доступа: <http://314159.ru/kushnerov/kushnerov1.pdf>
4. Алексеев, В.Б. Теорема Абеля в задачах и решениях / В.Б. Алексеев. – М.: Наука, 1976. – 105с.
5. Амамия, М. Архитектура ЭВМ и искусственный интеллект / М.Амамия, Ю.Танака. – М.: Мир, 1993. – 400 с.
6. Асанов, М.О. Дискретная математика: графы, матроиды, алгоритмы / М.О. Асанов, В.А.Баранский, В.В.Расин. – Ижевск: РХД, 2001. – 288 с.
7. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Мир, 1976. – 536 с.
8. Барендрегт, Х. Лямбда-исчисление. Его синтаксис и семантика / Х.Барендрегт. – М.: Мир, 1985. – 606 с.
9. Биркгоф, Г. Современная прикладная алгебра: пер. с англ. / Г.Биркгоф, Т. Бартти. – М.: Мир, 1976. – 400 с.
10. Бурцев, В. С.. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ / В. С. Бурцев. – М.: ИВВС РАН, 1997. – 154 с.
11. Корнеев, В.Д. Параллельное программирование в MPI : учеб. пособие / В.Д. Корнеев. – Ярославль: Яросл. гос. ун-т., 2002. – 104 с
12. Малышкин, В.Э. Введение в параллельное программирование мультимикрокомпьютеров / В.Э. Малышкин. – Новосибирск: НГТУ, 2003. – 268 с.
13. Введение в криптографию / под общ. ред. В. В. Яценко. 3-е изд., доп. – М.: МЦНМО: ЧеРО, 2000 - 288 с.
14. Верещагин, Н.К. Лекции по математической логике и теории алгоритмов. Ч. I. Начала теории множеств / Н.К.Верещагин, А.Шень. – М.: МЦНМО, 1999. – 128 с.
15. Верещагин, Н.К. Лекции по математической логике и теории алгоритмов. Ч. II. Языки и исчисления / Н.К.Верещагин, А.Шень. – М.: МЦНМО, 2000. – 291 с.
16. Верещагин, Н.К. Лекции по математической логике и теории алгоритмов. Ч. III. Вычислимые функции/ Н.К.Верещагин, А.Шень. – М.: МЦНМО, 1999. –176 с.
17. Вычислительные методы и решение оптимизационных задач. – Новосибирск, 2004. – Vol. 3. – С. 107 – 112.
18. Герасимов, А. С. Курс математической логики и теории вычислимости: учеб. пособие / А. Герасимов. 3-е изд., испр. и доп. – СПб.: ЛЕМА, 2011.– 284 с.
19. Кук, Д. Компьютерная математика / Д. Кук, Г. Бейз. – М.: Наука, 1990. – 385 с.
20. Калачев, А. Процессоры семейства SEAForth / А.Калачев //Компоненты и технологии. – 2009. – № 4. – С. 66-72.
21. Кановой, В.Г. Современная теория множеств: борелевские и проективные множества/ В.Г.Кановой, В.А. Любецкий. – М.: МЦНМО, 2010. – 320 с.
22. Карпов, Ю.Г. Теория автоматов / Ю.Г. Карпов. – СПб.: Питер, 2003. – 206 с.
23. Ким, А.К. Микропроцессоры и вычислительные комплексы семейства "Эльбрус" / А.К.Ким, В.И.Перекатов, С.Г.Ермаков. – СПб.: Питер, 2013. – 272с. ил.
24. Кобринский, Н.Ф. Введение в теорию конечных автоматов / Н.Ф.Кобринский, Б.А.Трахтенброт. – М.: Физматгиз, 1962. – 404 с., ил.
25. Колмогоров, А.Н. Теория информации и теория алгоритмов / А.Н. Колмогоров. – М.: Наука, 1987. – 305 с.
26. Конспект лекций по курсу «Программирование на языке ассемблера», Digital Desing и проект ИТ-Университет [Электронный ресурс] . – Режим доступа: <http://frogs.digdes.ru/shkola-programmista/materiali/>
27. Кормен, Т. Алгоритмы: построение и анализ : пер. с англ. / под ред. И. В. Красикова; Т.Кормен, Ч. Лейзерсон, Р. Ривест, К.Штайн. – 2-е изд. – М.: Вильямс, 2005. – 263 с.
28. Котов, В. Е. Сети Петри / В. Е. Котов. – М.: Наука, 1984. – 160 с.
29. Курош, А. Г. Курс высшей алгебры / А. Г. Курош. – М.: Наука, 1968. – 431 с.
30. Лидовский, В. В. Теория информации: учеб. пособие / В. В. Лидовский. М.: Компания Спутник+, 2004. – 111 с.
31. Мински, Марвин Ли. Вычисления и автоматы / Марвин Л. Мински. – М.: Мир, 1971. – 366 с.
32. Миронов, А.М. Теория процессов [Электронный ресурс] / А.М. Миронов . – Режим доступа: <http://intsys.msu.ru/staff/mironov/processes.pdf>
33. Брусенцов, Н.П. Троичные ЭВМ “Сетунь” и “Сетунь 70”[Электронный ресурс] / Н.П. Брусенцов, Рамиль Альварес Хосе. – Режим доступа: http://www.computer-museum.ru/histussr/setun_b.htm
34. Непейвода, Н.Н. Основания программирования / Н. Н. Непейвода, И.Н.Скопин. – Москва-Ижевск: РХД. Институт компьютерных исследований, 2003. – 868 с.

35. Олишук, С.А. Анализ процессных алгебр и их использование в задачах распределенного имитационного моделирования / С.А. Олишук, М.А. Волк // Системы управління, навігації та зв'язку. – 2010. – №4(16). – С.150-153.
36. Парамонова, И. М. Симметрия в математике / И. М. Парамонова. – М.: МЦНМО, 2000. – 16 с.: ил. – (Серия: "Библиотека "Математическое просвещение"", выпуск 7).
37. Пентковский, В.М. Автокод Эльбрус. Принципы построения языка и руководство к пользованию / В.М. Пентковский. – М.: Наука, 1982. – 352 с.
38. Пентус, А. Е. Теория формальных языков: учебное пособие / А. Е. Пентус, М. Р.Пентус. – М.: ЦПИ МГУ, 2004. – 80 с.
39. Пильщиков, В.Н. Машина Тьюринга и алгоритмы Маркова: решение задач: учеб. – метод. пособие / В.Н.Пильщиков, В.Г.Абрамов, А.А.Вылиток и др. – М.: МГУ, 2006. – 47 с.
40. Поспелов, Д.А. Арифметические основы вычислительных машин дискретного действия / Д.А.Поспелов. – М. Высшая школа, 1970. –308 с.
41. Проект "Кронос"/ Сибирское отделение АН СССР Институт систем информатики [Электронный ресурс]. – Новосибирск, 1990. – Режим доступа: <http://www.kronos.ru>
42. Рутковская, Д. Нейронные сети, генетические алгоритмы и нечеткие системы / Д.Рутковская, Л. Рутковский, М.Пилиньский. – М.: Горячая линия-Телеком, 2006. – 452с.
43. Кун, С. Матричные процессоры на СБИС/ С. Кун. – М.: Мир, 1991. – 673 с.
44. Семёнов, А. Л. Математика текстов / А. Л. Семёнов. – М.: МЦНМО, 2002. – 16 с.: ил. (Серия: "Библиотека "Математическое просвещение"", вып. 22).
45. Скопин, И.Н. Ленивые и жадные вычисления в числовых задачах[Электронный ресурс] /И.Н. Скопин. – Режим доступа: <http://www2.sccc.ru/SORAN-INTEL/paper/2010/Skopin-29-04-10a.pdf>
46. Скопин, И.Н. Организация распределенных вычислений и взаимодействие процессов // Мир ПК. – №12. – 2005 (CD-приложение к журналу).
47. Стековые процессоры или Новое – это хорошо забытое новое //Компоненты и технологии. – 2009. – №4. – С. 128-133.
48. Торгашев, В.А. Средства организации параллельных вычислений и программирования в мультипроцессорах с динамической архитектурой / В.А. Торгашев, И.В. Царев // Программирование. – 2001. – № 4. – С. 53-68.
49. Торгашев, В.А. Семейство суперкомпьютеров с динамической архитектурой – концептуальные основы / В.А. Торгашев, И.В. Царев; СПб институт информатики и автоматизации РАН // Штучний інтелект. – 2009. – № 3. – С. 251-257.
50. Утешев, А. Ю. Конструктивная алгебра: алгоритмические основы обработки информации [Электронный ресурс] / А. Ю. Утешев; СПбГУ Факультет прикладной математики и процессов управления. – Режим доступа: <http://www.apmath.spbu.ru/ru/education/courses/optional/uteshev-algebra.html>
51. Хамахер, К. Организация ЭВМ / К. Хамахер, З. Вранешич, С. Заки. – 5-е изд. – СПб.: БХВ-Питер, 2003. – 846 с.
52. Хоар, Ч. Взаимодействующие последовательные процессы / Ч. Хоар; под ред. А. П. Ершова – М.: Мир, 1989. – 264 с.
53. Шень, А. Программирование: теоремы и задачи / А. Шень. – 2-е изд., испр. и доп. – М.: МЦНМО, 2004. – 296 с.: ил.
54. Шилов, Н. В. Основы синтаксиса, семантики, трансляции и верификации программ [Электронный ресурс] / Н. В. Шилов. – Режим доступа: <http://persons.iis.nsk.su/files/persons/pages/langproc.pdf>
55. Яценко, И. В. Парадоксы теории множеств / И. В. Яценко. – М.: МЦНМО, 2002. – 40 с.: ил.
56. Brey, Barry B. THE INTEL MICROPROCESSORS 8086,8088,80186,80286,80386,80486, Pentium, and Pentium Pro Processor: Architecture, Programming, and Interfacing [Электронный ресурс] / Barry B. Brey. – Режим доступа: <http://faculty.bracu.ac.bd/~faruqe/books/microprocessor/The%20Intel%20Microprocessors%2080868088,%208018680188,%2080286,%2080386,%2080486,%20Pentium,%20and%20Pentium%20Pro%20Processor%20Architecture,%20Programming,%20and%20Inter-%20facing.pdf>
57. Briais, S. A formalization of the spi calculus [Электронный ресурс] / S. Briais. – Режим доступа: http://sbriais.free.fr/talks/talk_msr.pdf
58. Dowson, B. Comparing floating point numbers [Электронный ресурс] / B. Dowson. – Режим доступа: <http://www.cygnum-software.com/papers/comparingfloats/comparingfloats.htm>
59. Petri, C.A. Introduction to general net theory / C.A. Petri // Proc. Advanced Course on General Net Theory, Processes and Systems / editor W. Brauer. – : Springer Verlag, 1980. – P. 1–20.
60. Algorithms and Architectures for Parallel Processing [Электронный ресурс]: International Conference, ICA3PP 2010, Busan, Korea, May 21-23, Proceedins: in 2 p. P. 1. / Editors Ching-Hsien Hsu Laurence T.Yang Jong Hyuk Park Sang-SooYeo. – Режим доступа: [http://books.google.ru/books?id=yJNHICYATYC&pg=PR1&lpg=PR1&dq=Ching-Hsien+Hsu+Laurence+T.Yang+Jong+Hyuk+Park+Sang-Soo+Yeo+\(Eds.\)+Algorithms+and+Architectures+for+Parallel+Processing](http://books.google.ru/books?id=yJNHICYATYC&pg=PR1&lpg=PR1&dq=Ching-Hsien+Hsu+Laurence+T.Yang+Jong+Hyuk+Park+Sang-Soo+Yeo+(Eds.)+Algorithms+and+Architectures+for+Parallel+Processing)

61. Speculative Execution in High Performance Computer Architectures [Электронный ресурс] / Editors D. Kaeli, Pen-Chung Yew. – Режим доступа: <http://books.google.ru/books?id=aokeGc8vBIQC&printsec=frontcover&dq=Speculative+Execution+in+High+Performance+Computer+Architectures>
62. Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic [Электронный ресурс] / David Goldberg. – Режим доступа: <http://www.validlab.com/goldberg/paper.pdf>
63. Sima, D. Advanced Computer Architecture: A Design Space Approach, Louisiana Tech University / D. Sima, T. Fountain, P. Kacsuk. – Louisiana: Pearson Imprint Pearson, 1997. – 792 p.
64. Fredkin, E. Conservative Logic / E. Fredkin, T. Toffol // International Journal of Theor. Physics. – 1982. – № 21. – P. 219-253.
65. Knill, E. A Scheme for Efficient Quantum Computation / E. Knill, R. Laflamme, G.J. Milburn // With Linear Optics, Nature. – 2001. – Jan. – P. 46-52.
66. Logic Array (RPLA) using Fredkin & Feynman Gates for Industrial Electronics and Applications: International Conference on Innovations and Advancements in Information and Communication Technology-2012 [Электронный ресурс] / Pradeep Singla M.Tech Scholar. – Режим доступа: http://www.academia.edu/1855384/Reversible_Programmable_Logic_Array_RPLA_using_Feynman_and_MUX_Gates_for_Low_Power_Industrial_Applications
67. Intel. Pentium® Processor Family Developer's Manual [Электронный ресурс]. Volume 1. – Режим доступа: http://datasheets.chipdb.org/Intel/x86/Pentium/241428_4.PDF
68. Bergstra, J.A. Handbook of Process Algebra / J.A. Bergstra, A. Ponse, S.A. Smolka. – Amsterdam: North-Holland, 2001. – 1356 p.
69. Connelly, J. Ternary Computing Testbed 3-Trit Computer Architecture, Computer Engineering Department [Электронный ресурс]. / J. Connelly – Режим доступа: <http://xyzyzy.freeshell.org/trinary/CPE%20Report%20-%20Ternary%20Computing%20Testbed%20-%20RC6a.pdf>
70. Li, Ming. Theory of thermodynamics of computation // Ming Li, Paul M. B. Vitányi // Proc. IEEE Physics of Computation Workshop. – Dallas, 1992. – P. 42-46.
71. Marczyński, R. W. The First Seven Years of Polish Digital Computers [Электронный ресурс] / R. W. Marczyński // Annals of the History of Computing. – 1980. № 1 (Januar). – Vol. 2. – Режим доступа: <http://chc60.fgc.edu/images/articles/Marczyński.pdf>
72. Gutzmann, Michael M. Classification Approaches for Parallel Architecture [Электронный ресурс] / Michael M. Gutzmann, R. Weper // Berichte zur Rechnerarchitektur. – 1996. – № 19. – Режим доступа: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.4927&rep=rep1&type=pdf>
73. Vitányi, P. Quantum Kolmogorov complexity using classical descriptions [Электронный ресурс] / P. Vitányi // Transactions on Information theory. – 2001. – Vol. 47, № 6. – Режим доступа: <http://homepages.cwi.nl/~paulv/papers/quantumkolmcompl.pdf>
74. Treleaven, P.C. Data Driven and Demand Driven Computer Architecture / P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins. – Newcastle: Computing Laboratory, University of Newcastle upon Tyne, 1981. – 73 p.
75. Intel®. PENTIUM® PROCESSOR Specification Update [Электронный ресурс]. – Режим доступа: <http://download.intel.com/support/processors/pentium/sb/242480.pdf>
76. Pnueli, Amir. The temporal logic of programs / Amir Pnueli // Proceedings 19th Symposium on Foundations of Computer Science. – 1977. – P. 46-57.
77. Landauer, R. Irreversibility and Heat Generation in the Computational Process / R. Landauer // Journal of Research and Development. – 1961. – № 5. – P. 183-191.
78. Baase, Sara. Computer Algorithms: Introduction to Design & Analysis / Sara Baase, Allen Van Gelder. – 3-rd Edition. – [Б.м.]: Addison-Wesley, 2000 – 688 p.
79. Khmel'nik, Solomon I. Computer Arithmetic of Functions Algorithms and Hardware Design (in Russian) / Solomon I. Khmel'nik. – Israel, Bene-Ayish: MiC, 2004. – 164 p.
80. Komura, Taku. Systems Architecture for Visualisation – Lecture 4 [Электронный ресурс] / Taku Komura. – Режим доступа: http://www.inf.ed.ac.uk/teaching/courses/vis/lecture_notes/lecture4.pdf
81. Advances in computer systems architecture: 10th Asia-Pacific conference, ACSAC 2005, Singapore, October 24-26, 2005: proceedings / Thambipillai Srikanthan; Jingling Xue; Chip-Hong Chang. – Berlin; New York: Springer, 2005. – 657 p.
82. Thompson, S. Haskell: The Craft of Functional Programming / S. Thompson. – 2nd Edition. – [Б.м.]: Addison Wesley, 1999. – 528 p.
83. Hughes, J. Why Functional Programming Matters [Электронный ресурс] / J. Hughes. – Режим доступа: <http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf>
84. Kahan, W. An Interview with the Old Man of Floating-Point [Электронный ресурс] / W. Kahan, Ch. Severance // IEEE Computer. – 1998. – № 3. – Режим доступа: <http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html>

ПРИЛОЖЕНИЯ

Приложение 1

Краткая история ЭВМ

XV Век

1478 – Леонардо да Винчи в одном из своих дневников приводит эскиз 13-разрядного суммирующего устройства с десятизубцовыми кольцами.

XVII Век

1600 – Лейб-медиком королевы Елизаветы Тюдор Уильямом Гилбертом в его сочинении «О магните, магнитных телах и о большом магните – Земле» введен термин электричество, в нем же объясняется действие магнитного компаса и описываются некоторые опыты с наэлектризованными телами.

1617 – Джон Непер, шотландский барон, изобретатель логарифмов, первый публикатор логарифмических таблиц, описал метод умножения с помощью палочек. С их помощью можно было умножать и делить.

1642 – Блез Паскаль описал машину для суммирования чисел – «Паскалину». Основой ее стал счетчик-регистратор, или счетная шестерня (колесо Паскаля). Она имела десять выступов, на каждом из которых были нанесены цифры.

1658 – В русских летописях впервые упоминаются счеты.

1679 – Лейбниц публикует первое описание двоичной системы счисления.

1694 – Лейбниц строит 12-разрядный арифмометр, наиболее совершенная конструкция вплоть до середины XIX века.

XVIII Век

1794 – Первая надежная крупномасштабная сеть для передачи сообщений со стандартизированной системой кодирования – оптический телеграф Клода Шаппа появился во Франции в 1794 году и проработал 58 лет. Число семафорных станций Франции к середине XIX в. достигло 556.

1795 – Во Франции математик Гаспар Прони решал проблему перехода на метрическую систему мер и создал три класса исполнителей задачи, заложив основы теории управления и алгоритмов. Первый класс исполнителей разрабатывал методики расчетов, второй составлял последовательность действий — алгоритмы, третий – выполнял их.

XIX век

1804 – Инженер Жозеф Мари Жаккар построил полностью автоматизированный ткацкий станок, способный воспроизводить сложнейшие узоры. Работа станка программировалась при помощи колоды перфокарт, каждая из которых управляла одним ходом челнока.

1818 – Первый типовой серийный арифмометр, положивший начало счетному машиностроению, был построен руководителем парижского страхового общества Чарльзом-Ксавьером Томасом де Колмаром. Их выпуск был доведен в год до сотни, из которых шестьдесят экспортировались в другие страны. Чтобы умножить два восьмизначных числа с помощью первых арифмометров, нужно было 15 секунд, а деление шестнадцатизначного числа на восьмизначное занимало 25 секунд, машина была 16-разрядной, стоила 400 франков и выпускалась до начала XX века.

1820 – Ампер также строит прототип электромагнитного телеграфа.

1822 – Чарльз Бэббидж закончил строительство малой дифференциальной машины, бывшей первым воплощенным устройством такого класса.

1834 – Чарльз Бэббидж, продолжая постройку большой дифференциальной машины, одновременно начинает параллельный проект – аналитическую машину. Это устройство не имело равных по механической сложности среди всех, когда-либо спроектированных человеком, и представляло собой программируемый, полный по Тьюрингу цифровой механический компьютер, содержащий механический процессор с регистрами и устройством управления, механическую оперативную память, принтер и устройство ввода с помощью перфокарт.

1835 – Джозеф Генри в США изобретает электромеханическое реле.

1843 – Леди Ада Лавлейс Байрон публикует записи лекций об аналитической машине Бэббиджа. В одном из своих комментариев Ада приводит общие идеи программирования и описывает алгоритм вычисления Чисел Бернулли на аналитической машине. Было признано, что это первая программа, специально реализованная для воспроизведения на компьютере, а Ада Лавлейс считается первым программистом в истории.

1854 – Джордж Буль опубликовал наиболее фундаментальную и глубокую работу по логике – «Исследование законов мышления, на которых основываются математические теории логики и вероятностей».

1876 – После многих лет неудач и трагедий удалось проложить трансатлантический телеграфный кабель между США и Европой. Александр Белл создает телефон.

1877 – Эдисон изобретает фонограф, Девид Хьюз микрофон. Генрих Герц открыл фотоэффект.

1878 – На клавиатуре появляется клавиша Shift, QWERTY становится стандартной раскладкой, клерк Франк Эдвард Макгуррин изобретает слепой метод печати и, отточив технику, через 10 лет побеждает на соревнованиях стенографистов. Уильям Крукс создает прототип ЭЛТ.

1879 – Джеймс Ритти, владелец салуна, изобретает компактное арифметическое устройство, совмещенное с принтером – кассовый аппарат. Для него он придумал приспособление, фиксирующее каждую денежную операцию на специальном бумажном рулоне. Механизм записи покупки представлял собой перфоратор и специальную бумагу с колонками для записи количества долларов и центов.

Готлоб Фреге становится одним из основателей современного символического языка, а его работа – предшественником к исчислению предикатов, которое используется для представления понятий и операций в языках искусственного интеллекта.

1883 – Эдисон, пытаясь продлить срок службы лампы с угольной нитью, ввёл в её вакуумный баллон платиновый электрод и положительное напряжение, создав, таким образом, первую электронную лампу.

1884 – Основано старейшее общество специалистов по электронике – AIEE (American Institute of Electrical Engineers), в будущем преобразованное в международную ассоциацию стандартизации IEEE.

1885 – Основана компания American Telephone and Telegraph (AT&T Corp.). Много лет она являлась монополистом на рынке дальней и местной связи США. Дорр Фельт из Чикаго строит комптометр – первый серийный калькулятор, где числа вводятся нажатием клавиш.

1886 – Логик Чарльз Пирс предлагает соединить булеву алгебру и электрические цепи, основанные на переключателях, предсказав, таким образом, двоичный электромеханический компьютер, но никаких реальных изменений в этой области не происходит, пока Клод Шеннон не издаст статью Пирса в 1938 г.

1888 – Мысль о том, что намагничивание может быть использовано для записи звука, высказана Оверлингом Смитом. Описанное Смитом устройство имело все отличительные признаки магнитофона: магнитный носитель информации – проволоку, механизм для ее подачи и магнитную головку.

Генрих Герц в статье «Об электродинамических волнах в воздухе и их отражении» описал электромагнитные волны. Австрийский ботаник Фридрих Райнцер открывает жидкие кристаллы.

1889 – Никола Тесла демонстрирует первый опыт телеуправления с помощью радиоволн на выставке в Madison Square Garden в Нью-Йорке и изобретает ВЧ-генератор.

Бизнесмен из Канзас-Сити Элмон Струоджер заподозрил, что операторы местной телефонной станции перенаправляют выгодные заказы его конкурентам, и нанял несколько инженеров для решения этой проблемы. Результатом стала АТС – автоматическая телефонная станция.

1890 – Герман Холлерит блестяще применяет свои электромеханические табуляторы для статистической обработки таблиц переписи населения США. Результаты, которые обычно выдавались через 7-10 лет работы аналитиков, с помощью машин Холлерита были обчислены за 2 недели.

1893 – В Сент-Луисе Никола Тесла впервые продемонстрировал радиосвязь.

1895 – Оливер Лоджт провёл удачный эксперимент с радиотелеграфической связью, послав сигнал азбукой Морзе на расстояние 40 метров. Александр Степанович Попов усовершенствует прибор Лоджта и демонстрирует радиопередачу в Петербурге.

1896 – Холлерит основал компанию по производству счетно-аналитических машин – TMC – Tabulating Machine Company, позднее преобразованную в IBM.

XX Век

1900 – В городе Хартфорд штата Коннектикут был установлен первый уличный телефон-автомат, требующий монетки в качестве оплаты за пользование.

1902 – Радиотелефон изобретен Вальдемаром Поульсеном и Реджинальдом Фессенденом. Немецким физиком Артуром Корном запатентована технология фотоэлектрического сканирования, получившая впоследствии название телефакс.

1904 – Академик А.Н. Крылов предложил конструкцию машины для интегрирования дифференциальных уравнений, которая была построена в 1912 г.

Специалисты компании Bell создали телефон, объединяющий в своем функционале приемник и передатчик.

1905 – Флеминг изобретает диод – двухэлектродную электронную лампу. Альберт Эйнштейн опубликовал уравнение фотоэффекта.

1906 – Ли Д. Форест патентует вакуумный триод. В городе Рочестер, штат Нью-Йорк, основана «Haloid Company» (будущая «Xerox Company»), производящая фотобумагу.

1915 – Эдвин Портер и Уильям Уодделл представили в Нью-Йоркском театре «Астор» серию экспериментальных фильмов, снятых по анаглифическому методу (в красном и зелёном цвете). Рождение 3D-кинематографа.

1917 – Корпорация CTR переименована в International Business Machines Co., Limited или сокращенно IBM.

1918 – Александр Николсон изготавливает кварцевый генератор – одну из важнейших частей не только ЭВМ, но и вообще практически любой цифровой электронной техники. М. А. Бонч-Бруевич создал электронное реле, названное триггером.

1919 – Эдвард Хепберн разрабатывает Enigma – электромеханическую шифровальную машину.

1920 – Польский математик Ян Лукаевич разрабатывает бесконечный метод записи арифметических выражений, используемый во всех современных алгоритмах – прямую и обратную польскую нотацию.

1921 – Карел Чапек в пьесе «Универсальные Роботы Руссума» вводит термин робот – от чешского robota – подневольный труд.

1928 – Ванневар Буш усовершенствует интеграф и на его основе строит первый в мире аналоговый компьютер – дифференциальный анализатор. Основана корпорация Motorola.

1935 – В универсаме Питтсбурга установлена система из 250 терминалов, соединенных телефонными линиями с 20 табуляторами и 15 пишущими машинками. С терминалов передавались данные, отперфорированные на ярлыках, которыми снабжались продаваемые товары. Так был создан прообраз ЛВС.

1936 – Английский математик Алан Тьюринг (статья «О вычислительных числах») и независимо от него польский логик Э. Пост выдвинули и разработали концепцию абстрактной вычислительной машины – машины Тьюринга и машины Поста.

Конрад Цузе открывает, что машинные команды могут храниться в той же памяти, что и данные.

1937 – Клод Шеннон показал, что операции булевой логики и некоторые логические вентили эквивалентны.

1938 – Цузе создает Z1 – двоичный механический вычислитель с электрическим приводом и ограниченной возможностью программирования.

1939 – IBM разрабатывают Harvard Mark I. Mark I был перенесён в Гарвардский университет и начал работу в мае 1944 года.

Американский профессор физики Джон Атанасов вместе с аспирантом Клиффордом Берри разработали проект и незавершенную модель компьютера ABC – Atanasoff-Berry Computer. Проект стал первой системой, в которой были использованы цифровые коммутационные технологии и вакуумные лампы, а также концепции двоичной арифметики и логических схем.

1940 – В Англии осуществлен совершенно секретный военный проект – на электромеханических реле собран «Robinson» – первый операционный компьютер.

1941 – Открытие транзисторного эффекта Вадимом Евгеньевичем Лашкаревым. Это явление получило название p-n перехода. Основана Texas Instruments.

1942 – Создан электронно-механический аналоговый компьютер Ванневара Буша, содержащий 2000 электронных ламп и около 400 километров проводов, который через год начал применяться в Манхэттенском проекте.

1943 – Алан Тьюринг разрабатывает и строит ЭВМ Colossus.

1944 – Американский инженер Джон Преспер Эккерт впервые выдвинул концепцию хранимой в памяти компьютера программы.

IBM выпустила 5 одноплатных релейных машин («Я думаю, что в мире есть спрос, ну, положим, на пять компьютеров») – Томас Уотсон, IBM, 1943 г.)

Эккертом и Джоном Мокли начато строительство первой полностью электронной машины «ENIAC» (Electronic Numerical Integrator And Computer).

1945 – Цузе разрабатывает Plankalkul – первый алгоритмический язык программирования высокого уровня и пишет сложные программы для игры в шахматы.

Джон Мокли и А. Беркс написали статью «First Draft of a Report on the EDVAC».

1946 – Эккерт и Мокли вместе с сотрудниками университета Пенсильвания ввели в строй ENIAC, проработавший до 1955 года. Он работал с десятизначными числами и выполнял операции умножения со скоростью около 300 произведений в секунду, находя значения каждого произведения в таблице умножения, хранящейся в оперативной памяти. Эта система работала примерно в 1000 раз быстрее, чем электромеханические релейные вычислительные машины предыдущего поколения. В компьютере ENIAC было 18 тыс. вакуумных ламп, он занимал площадь 167 кв. м. и потреблял приблизительно 180 тыс. ватт. Команды, составляющие программу, создавались с помощью монтажной схемы и переключателей, которые управляли ходом вычислений.

1947 – 23 декабря 1947 Джоном Бардином, Уолтером Браттейном и Уильямом Шокли из Bell Labs успешно протестирован первый транзистор.

Созданы EDVAC и UNIVAC (Universal Automatic Computer). Двадцатью пятью национальными организациями по стандартизации создана ISO – Международная Организация по Стандартизации. Опубликован знаменитый труд математика Клода Шеннона «Математическая теория коммуникаций», в котором впервые была описана структура информации.

Голография предложена Дэннисом Габорм, он же ввёл термин голограмма.

1949 – В Кембриджском университете Морис Уилкс создал программируемый компьютер EDSAC (Electronic Delay Storage Automatic Computer), первый действующий компьютер, программы которого хранились в его внутренней памяти, и разработал для него первый ассемблер.

Джон Мокли придумывает Short Order Code — первый интерпретатор языка программирования.

Джон фон Нейман читает серию лекций в Иллинойском университете «Теория и организация сложных автоматов».

1950 – На UNIVAC спрогнозирована победа Эйзенхауэра на выборах, что приносит огромный коммерческий успех и известность всей индустрии ЭВМ.

Первый универсальный программируемый компьютер в континентальной Европе создан под руководством С. А. Лебедева из Киевского института электротехники. МЭСМ (Малая электронная счётная машина) содержала около 6000 электровакуумных ламп и потребляла 15 кВт. Машина могла выполнять около 3000 операций в секунду.

Алан Тьюринг в своей работе «Computing Machinery and Intelligence» описывает способ определения, является ли машина мыслящей. Метод стал известен как тест Тьюринга.

Эдмунд Беркли издает первый компьютерный журнал «Компьютеры и автоматизация».

Йосуито Накамато изобретает технологию записи на гибкие магнитные диски.

Ричард Хемминг из Bell Labs опубликовал алгоритм обнаружения и исправления ошибок передачи данных – код Хемминга.

1951 – Грейс Хоппер (математик, а также первая женщина – адмирал ВМФ США) создает первый, со времен Цузе, язык высокого уровня A-0.

Магнитная лента была впервые использована для хранения данных – эффективная скорость передачи данных составила около 7200 символов в секунду для ленточного устройства UNISERVO.

1952 – В Бюро переписи населения США установлен компьютер UNIVAC-1103 фирмы Remington-Rand, в котором впервые были применены программные прерывания.

В СССР построен компьютер БЭСМ (Большая Электронная Счетная Машина) — трехадресная машина параллельного действия, оперировавшая с 39-разрядными словами со скоростью 10 тыс. операций в секунду. На 1953 год она была самой мощной в Европе, но уступала по быстродействию и объёму памяти коммерческой американской IBM 701, поставки которой начались в декабре 1952 г.

IBM выпускает свой первый промышленный электронный компьютер IBM 701 – синхронную ЭВМ параллельного действия, содержащую 4000 электронных ламп и 12000 германиевых диодов. В качестве внешнего носителя информации, в вычислительной машине IBM 701 впервые использовали магнитную ленту Model 726, содержащую 1,4 Мбайт данных.

Первая в мире компьютерная игра была запрограммирована на EDSAC Дугласом Энгельбартом. Волоконная оптика изобретена Нариндером Капани.

1954 – IBM продемонстрировала первый полностью транзисторный калькулятор Model 650; в течение этого же года было продано 450 экземпляров.

1955 – Компания Bell Laboratories анонсировала первый полностью транзисторный компьютер TRADIC.

Морис Уилкс изобретает микропрограммирование контроллеров, позволяющее определять или расширять базовый набор команд с помощью встроенных микрокодов (firmware).

В Линкольновской лаборатории Массачусетского Технологического Института создан компьютер TX-0. Помимо большой памяти (65536 байт), он имел самые современные устройства ввода-вывода: дисплей на ЭЛТ и фоточувствительное устройство, называемое световым пером, которое позволяло изменять изображение на экране.

1956 – Известный ученый и общественный деятель-анархист Ноам Хомский публикует «Syntactic Structures», в которой создает структурную лингвистику – теорию контекстно-свободной грамматики, ставшую основой всех современных языков программирования.

Первый жесткий диск 24” IBM 305 RAMAC. Устройство имело размер, сравнимый со столом, весило 900 кг, имело емкость 5 Мб и содержало 50 алюминиевых намагничиваемых пластин. Диск стоил около 50.000\$.

1957 – Л. И. Куприянович публично продемонстрировал первый опытный носимый мобильный телефон ЛК-1 весом 3 кг, радиусом действия 20-30 км и временем работы без смены батарей 20-30 часов и базовую станцию к нему.

Группой программистов под руководством Джона Бэкуса в корпорации IBM завершена разработка языка Fortran (FORmula TRANslation) – первого языка программирования высокого уровня, имеющего транслятор.

Casio выпустила первый в мире полностью электронный калькулятор Model 14-A.

Первый искусственный спутник Земли, первый космический аппарат был запущен на орбиту в СССР 4 октября 1957 года.

1958 – Книга А. П. Ершова «Программирующая программа для электронной вычислительной машины БЭСМ» — первая работа по автоматизации программирования.

Джек Килби из Texas Instruments создает первую интегральную схему, состоящую из шести транзисторов и нескольких конденсаторов на одной полупроводниковой пластине.

Независимо от него идея интегральной схемы предложена также Робертом Нойсом из Fairchild Semiconductor, впервые расположившим соединения непосредственно на пластине.

Под руководством Н.П. Бруснецова в МГУ начата разработка уникальной по архитектуре и быстродействию ЭВМ Сетунь – первого компьютера на основе троичной логики.

Джон Маккарти создает язык LISP (List Procassing Language – язык обработки списков) функций, первый в мире декларативный язык программирования.

В СССР выпущена БЭСМ-2, в США – IBM 7070 – первый мейнфрейм с архитектурой клиент/сервер.

Bell создает первый модем — скорость 12.5-37.5 КБ/сек.

Изобретатель слова «бит» Джон Таки в журнале American Mathematical Mountly впервые употребляет термины hardware и software.

Фирма Computer Science Co продает первую программу отдельно от ЭВМ и становится первым в мире специализированным производителем ПО.

1960 – Тринадцать европейских и американских специалистов по программированию в Париже утвердили стандарт языка программирования ALGOL-60.

1961 – По данным журнала Datamation, продукция IBM занимала 81,2% компьютерного рынка.

1962 – Ник Холоньяк изобрел светодиод.

Совместными усилиями Bell Labs и NASA в космос был запущен первый спутник связи Telstar.

Фирма Teletype выпускает первую клавиатуру.

Математик Бенуа Мандельброт открывает фрактальную геометрию и применяет компьютер для построения фрактала — так появляется компьютерная графика.

Стив Расселл создает первую популярную игру Space War.

Дональд Кнут начинает писать «Искусство программирования», самый фундаментальный труд по информатике, не законченный до сих пор.

1963 – Дуглас Энгельбарт изобрёл компьютерную мышь.

Профессорами Дартмутского колледжа Томасом Куртом и Джоном Кемени разработан Beginners All-purpose Symbolic Instruction Code – BASIC.

Профессор Джек Деннис начинает разработку первой многозадачной ОС Multics (Multiplexed Information and Computing Service; из этого проекта впоследствии вырос UNIX).

1964 – Seiko выпускает первый коммерческий матричный принтер.

IBM выпускает массовый монитор IBM 2250 – монохромный дисплей 12”, разрешение 1024x1024, 40 Hz.

Сеймур Крей основывает корпорацию Cray.

В то же время IBM анонсировала семейство компьютеров System/360 – шесть совместимых модификаций и 40 периферийных устройств (в т.ч. жесткие диски IBM 2311 и IBM 2314, накопители на магнитных лентах IBM 2401 и 2405, оборудование для работы с перфокартами, устройства распознавания текстов, а также различные коммуникационные интерфейсы). Классическая архитектура IBM System/360 поддерживала одновременное подключение до 248 терминалов и, хотя и не была лишена серьезных недостатков (например, процессор не предусматривал реализацию даже стека, не говоря о виртуальной памяти), однако легла в основу всех современных мейнфреймов IBM.

1965 – Морис Уилкс вводит понятие кэша.

Гордон Мур обнаружил интересную особенность: емкость каждой новой микросхемы памяти удваивается, а сама новая микросхема появляется каждые 18-24 месяца. Эту зависимость стали называть законом Мура.

В СССР начата разработка БЭСМ-6 – вершины отечественных ЭВМ. Она имела 128 Кб оперативной памяти на ферритовых сердечниках и внешнюю память на магнитных барабанах и ленте. БЭСМ-6 работала с тактовой частотой 10 МГц. Архитектура БЭСМ-6 в значительной степени воспроизводила архитектуру системы CDC-1604, серийно выпускавшейся с 1960 года. Последняя из машин БЭСМ-6 проработала 30 лет и была списана и разобрана на металлолом в 1995 году.

1966 – Волоконно-оптические сети впервые были предложены Чарльзом Као и Джорджем Хокхамом.

В конце 1966 г. на заседании Академии наук СССР принимается историческое решение о закупке лицензии и полном копировании устаревшей на тот момент серии IBM-360 и сворачивании отечественных разработок ЭВМ. Против этого решительно выступил академик С. А. Лебедев и другие, но они остались в меньшинстве.

1967 – Начата разработка первого объектно-ориентированного языка программирования Simula.

Дэвид Нобль в лаборатории фирмы IBM предложил гибкий диск (прообраз дискеты диаметром 8”) и защитный кожух с тканевой прокладкой.

1968 – 18 июля 1968 года Робертом Нойсом, Гордоном Муром и Эндрю Грувом основана корпорация Intel – INTEgrated Electrínics.

Они разработали прототип первого в мире модуля ОЗУ – микросхему, известную как «динамическое оперативное запоминающее устройство 1103» – DRAM объемом 1 кбит.

Сотрудники AT&T Bell Laboratories Кеннет Томпсон и Деннис Ритчи создают операционную систему UNIX.

Хегох создает технологию лазерного копирования изображений.

Осуществлена первая связь между двумя компьютерами, установленными в Калифорнийском университете (Лос-Анджелес). На расстояние 5 метров было передано слово LOGIN.

Разработан стандартный разъем D-Subminiature, используемый в ПК до сих пор.

По распоряжению Министерства обороны США были созданы четыре узла сети ARPAnet, ставшей прообразом Internet. Два узла были установлены на территории Калифорнийского университета (один в Санта-Барбаре, а другой в Лос-Анджелесе) и по одному в SRI International и в Университете штата Юта.

В СССР был пик продаж арифмометров. Продано 300.000 штук.

IBM начала продавать отдельно программное обеспечение (или услуги по его изготовлению) и аппаратную составляющую, и на компанию посыпались жалобы, критика прессы и судебные иски. В итоге IBM стала отдельно продавать только прикладные приложения, тогда как ПО для контроля за работой компьютера (System Control Programming), фактически операционная система, обходилось бесплатно.

Джимми Сандерс основал AMD (Advanced Micro Devices).

В СССР создана «Кошечка» — первый мультфильм, выполненный с помощью ЭВМ (БЭСМ-4). В фильме показано перемещение силуэта кошки, сформированного печатными символами БЭСМ-4 на бумаге, после небольшой художественной обработки. Движение кошки моделировалось системой дифференциальных уравнений второго порядка.

1969 — Уиллардом Бойлом и Джорджем Смитом в AT&T Bell Labs изобретена матрица ПЗС. Стало ясно, что прибор способен получить заряд благодаря фотоэлектрическому эффекту, то есть могут создаваться изображения при помощи электронов.

1970 — Выпущена машина «Сетунь-70» — вторая и последняя троичная ЭВМ в мире.

Intel пускает в продажу первые интегральные схемы памяти.

Рэй Томлинсон разрабатывает первую программу для работы с электронной почтой SENDMSG.

Разработана первая сетевая карта.

Японская компания Busicom обратилась к Intel с просьбой разработать набор микросхем для семейства высокоэффективных программируемых калькуляторов. Первоначальная конструкция предусматривала, по крайней мере, 12 микросхем различных типов. Тед Хофф отклонил данную концепцию, так как Intel была небольшой компанией, которая не смогла бы выпустить такое количество разных микросхем, но, не желая терять выгодный контракт, предложил взамен однокристальное логическое устройство, получающее команды приложения из полупроводниковой памяти. Так появился микропроцессор Intel 4004.

1971 — Intel выпускает первый 4-разрядный «микрокомпьютерный набор 4004». Микросхема площадью 1 кв. см содержала 2250 транзисторов, стоила 300 долларов и по вычислительной мощности была сопоставима с ENIAC. Микропроцессор 4004 выполнял 60 тыс. операций в секунду, что являлось на то время невероятным достижением (производительность — 0.06 MIPS), он адресовал 640 байт ОЗУ и был изготовлен по технологии 10 мкм.

В журнале Electronic News в том же году появилась его реклама, а журнале Scientific American — реклама одного из первых персональных компьютеров Kenback-1 стоимостью 750 долларов.

Никлаус Вирт разрабатывает язык Paskal.

Рэй Томлинсон отправляет первое электронное письмо — "QWERTYUI" самому себе.

1972 — Херох создает первый КПК Dynabook, размером чуть больше записной книжки.

Билл Гейтс и Пол Аллен основывают компанию Traf-O-Data и разрабатывают компьютерную систему, предназначенную для управления потоками автомобилей на скоростных шоссе.

Выпущен Intel 8008 — 45 команд, 300 тыс. команд в секунду, адресация 16KB ОЗУ.

Появляются три основополагающих языка программирования: C (Деннис Ритчи из Bell Lab's), SmallTalk (Алан Кей из компании Херох) и Prolog (Programmation en logique — логическое программирование, ученые университета Люммини).

1973 — Алан Кей в исследовательском центре Херох создает первый персональный компьютер, оснащенный монитором — Херох PARC. Управлявшая им операционная система Alto — первая ОС с графическим интерфейсом, кроме того, PARC (также впервые) был укомплектован мышью и картой Ethernet.

Scelbi Computer Consulting Company выпускает на рынок первый готовый персональный компьютер, укомплектованный процессором Intel-8008 с 1 кб оперативной памяти.

IBM представляет жесткий диск IBM 3340. Накопитель состоял из двух дисков по 30 Мб в каждом — по этой причине он получил имя Winchester (30/30 — как и винтовка).

Начата работа над концепцией новой всемирной сети, впервые в печати появляется термин Internet (Interconnected Networks, Боб Кан, Винтон Серф).

В ходе экспериментов в Naval Research Laboratory Дэвид Стрикер, Джон Шор и Стэнли Вилсон впервые используют трояна для взлома компьютера Univac 1108.

В том же году разработаны принципы TCP/IP-адресации и FTP.

1974 — Проводится первый чемпионат мира по шахматам среди компьютеров. Советская программа «Каисса» завоевывает титул первого компьютерного чемпиона мира. Первый компьютер сыграл в шахматы с человеком, но живой шахматист легко выиграл.

Разработан язык SQL.

1975 — IBM выпускает первый портфельный компьютер IBM 5100 (IBM Portable Computer), оснащенный дисплеем, встроенным накопителем на магнитной ленте и 16 килобайтами оперативной памяти. Стоимость компьютера 10.000\$, вес — 23 кг.

Компания Popular Electronics начинает продажу первого массового персонального компьютера — MITS Altair-8800. Стоимость блока (монитор и клавиатура отсутствовали) - 397 долларов, и покупатель должен был сам собрать компьютер.

Пол Аллен и Билл Гейтс разрабатывают по заказу фирмы MITS интерпретатор языка Basic для компьютера Altair и 5 сентября 1975 г. основывают собственную фирму — Microtechnical Software (через год переименованную в Microsoft).

Под маркой Tele-Games, принадлежащей Sears, начались продажи игровой приставки Atari Pong, она мгновенно становится сверхпопулярной, до Рождества распродано 150.000 консолей, покупатели ждали в очередях по несколько часов.

1976 – Фирма AMD получает лицензию на копирование инструкций и микрокода процессоров Intel.

IBM разрабатывает первый прототип струйного принтера.

Личный электронный адрес получает королева Великобритании Елизавета II.

В печати публикуется открытое письмо Билла Гейтса, который жалуется на незаконное использование программного обеспечения Microsoft.

Майкл Шрейер разрабатывает первый текстовый редактор Electric Pencil.

Стив Возняк создал одноплатовый компьютер Apple I стоимостью 666 долларов. Его системная плата была привинчена к куску фанеры, а корпуса и блока питания не было вообще.

1977 – Представители Национального Института Профессиональной Безопасности и Здоровья США впервые измеряют уровень излучения мониторов. Они сообщают, что излучение монитора слишком низкое, чтобы его корректно измерить.

Джордж Лукас выпускает «Звездные Войны» — первый фильм с компьютерными спецэффектами.

IBM совместно с Motorola начинает разрабатывать архитектуру PowerPC.

Лоуренс Эллисон основал корпорацию Oracle.

1978 – Компания StorageTek разработала первый твердотельный накопитель современного типа, основанный на RAM-памяти.

В СССР снят с производства механический арифмометр Феликс, выпускавшийся с 1929.

1979 – Первый скриптовый язык Rexx (IBM UK Laboratories).

Дональд Кнут разработал систему для верстки текстов TeX.

Кевин Маккензи вводит в обращение смайлики :-).

Дэн Бриклин и Боб Фрэнкстон разрабатывают первую электронную таблицу VisiCalc.

Ричард Бартел и Роб Трэшбоу разрабатывают MUD — первую сетевую компьютерную игру (пока еще в чисто текстовом режиме).

Sony разрабатывает первый портативный аудиоплеер Walkman.

1980 – Джон Шох из исследовательского центра Пало Альто компании Xerox обнаружил первого компьютерного червя.

Seagate Technologies выпустила первый 5.25” жесткий диск.

На рынке микрокомпьютеров доминируют две базовые модели - Apple II и Altair. Эти компьютеры были совместимы один с другим, имели одну операционную систему CP/M и стандартные разъемы расширения с шиной S-100 (по 100 контактов на разъем).

Два наиболее известных в конце 1970-х годов процессора, используемых в ПК, не принадлежали Intel - Zilog Z-80 и MOS Technologies 6502.

1981 – Адам Осборн выпустил первый портативный компьютер Osborne I стоимостью 1795 долларов.

IBM выпустила свой первый персональный компьютер - IBM PC.

Компания Sony анонсировала первую 3,5” дискету.

Компании Philips и Sony представили дисковод CD-DA (компакт-диск с цифровой звукозаписью).

Семейство процессоров Intel пополнилось новой 16-разрядной моделью 8086.

1982 – Основана Sun (от Stanford University Networks) Microsystems, Inc. В 2009 г. куплена компанией Oracle за 7,4 миллиарда долларов.

Intel представила микропроцессор 286, содержащий 134 тыс. транзисторов.

ZX Spectrum — самый продаваемый английский компьютер.

Commodore 64 продано более 20 миллионов машин.

Джон Уорнок и Чарльз Гешке основали Adobe Systems, Incorporated.

1983 – Компания Apple выпустила компьютер Lisa, потерпевший катастрофический провал из-за ошибок проектирования, допущенных Стивом Джобсом.

Compaq Computer Corp. выпустила первый клон IBM PC.

1984 – Amiga Corporation в лице Эрджи Майкла и Дэйва Морса устраивает демонстрацию первого в мире персонального мультимедийного компьютера Amiga 1000. Демонстрация «Боинг» показывала, как трёхмерный шар, разрисованный красными и белыми квадратами, летает в трёхмерной же комнате и с грохотом ударяется о стены.

Выпущена Macintosh OS.

1985 – Рождение Интернета: 24 июля зарегистрирован домен .us, 24 июля домен .uk, 24 октября .il.

Компанией Philips выпущен первый музыкальный компакт-диск группы ABBA.

Первый 32-разрядный процессор Intel 386. Он содержал 275 тыс. транзисторов и выполнял более 5 млн. операций в секунду.

Выпущена первая версия GUI Microsoft Windows 1.0.

1986 – С 3 апреля 1986 г. в продаже первый ноутбук в современном смысле слова: IBM PC Convertible.

1988 – После провала второго проекта подряд (Lisa, а затем Apple III) Стив Джобс покинул Apple и создал собственную компанию NeXT.

Роберт Моррис создает и запускает червя в ARPAnet. Безобидная шутка привела к заражению более чем 60 тыс. узлов, полному коллапсу сети на несколько дней на всей территории США и многомиллионным потерям.

1989 – Intel выпускает процессор 486, который содержит уже 1,2 млн. транзисторов и первый встроенный сопроцессор. Он работал в 50 раз быстрее процессора 4004.

На Тайване основана корпорация ASUSTeK Computer Inc.

1990 – Фудзи Масуока изобретает флеш-память.

19 сентября зарегистрирован домен .su, 28 ноября домен .сп.

Первая полноценная цифровая камера DCS 100 выпущена компанией Kodak.

Тим Бернерс Ли, сотрудник Женевской лаборатории физики высоких энергий (CERN), разрабатывает гипертекстовый язык разметки HTML.

Федеральный совет по информационным сетям США отменяет правило, запрещавшее подключаться к Сети без рекомендации одного из государственных органов. Интернет становится общедоступным.

Первая русская версия DOS — MS-DOS 4.1.

В мае выходит Windows 3.0.

Разработан язык программирования Microsoft Visual Basic.

1991 – Линус Торвалдс выкладывает на свой ftp ядро новой ОС – Linux 0.01.

Запущена первая в истории вебкамера, она показывала кофеварку в Троянской комнате Кембриджского университета. Была отключена в 2001 году.

Разработан язык JAVA.

Филип Циммерман создает систему шифрования PGP (Pretty Good Privacy).

Первая аудиокарта — 8-битный Creative Sound Blaster Pro.

NCR Corporation и AT&T в Ньивегейн, Нидерланды, создает беспроводную сеть систем кассового обслуживания – WaveLAN, она обеспечивает скорость передачи данных от 1 до 2 Мбит/с.

Евгений Касперский разрабатывает свой антивирус для MS-DOS и IBM DOS.

Демонстрация Windows NT.

Разработан графический формат JPEG (Joint Photographic Experts Group).

1992 – Apple Newton.

Первый смартфон – IBM Simon.

В Сети появляется миллионный сервер.

Марк Андрессен и Эрик Бина, сотрудники NCSA (National Center for Supercomputing Application), разрабатывают первый браузер — Mosaic X.

AT&T анонсирует алгоритм сжатия видеoinформации MPEG-I.

1993 – Intel выпустила первый процессор Pentium из семейства P5. Кроме выпуска процессора, Intel разработала для него набор микросхем системной логики.

Выпущена FreeBSD.

Первый общедоступный веб-сайт в России создан компанией Элвис+. В Сети действует уже 600 сайтов.

Id Software выпускает в продажу знаменитую игру Doom.

Первая версия языка Delphi.

Джен-Хсун Хуанг, Кертис Прэм и Крис Малаховски основали компанию NVIDIA.

В Москве начали сборку клонов IBM PC – PS/1.

1994 – Рождение спама. Почтовые ящики американских пользователей заполнили рекламные письма от фирмы Canter & Siegel, предлагающей помощь в оформлении грин-кард.

На страницах впервые появляются рекламные баннеры.

Компьютер IBM Deep Blue обыгрывает в шахматы чемпиона мира Гарри Каспарова.

Выход Sony PlayStation, разработанной под руководством Кэна Кутараги.

Компаниями Netscape Communications и Microsoft разработан сценарный язык JavaScript для встраивания скриптов в HTML-файлы.

MPEG-2.

1995 – Компания Microsoft представила Windows 95.

Утвержден стандарт Digital video recorder – DVD. В продаже диски появятся лишь через 3 года.

Луис Монье создает первую поисковую систему AltaVista.

Регистрация доменных имен становится платной.

В двухкомнатной квартире в Сиэтле начинает работу интернет-магазин Amazon.

Студия Уолта Диснея выпускает на экраны мультфильм Toy Story — первый анимационный фильм, целиком созданный при помощи компьютера.

Арестован и осужден хакер Кевин Митник.

M-Systems представила первый коммерческий твердотельный накопитель на flash-памяти.

Начало производства массовых жидкокристаллических мониторов для домашних компьютеров.

В Сан-Хосе программист Пьер Омидьяр создал онлайн-аукцион AuctionWeb, как часть своего сайта. Первым предметом, проданным на AuctionWeb, была неисправная лазерная указка Омидьяра, за которую заплатили 13.83 \$. Связавшись с покупателем, Пьер поинтересовался у него, «поняли ли Вы, что лазерная указка неисправна?». В ответном письме покупатель объяснил: «Я коллекционер неисправных лазерных указок». Сейчас это eBay Inc. с оборотом \$11,65 млрд и средней прибылью \$1,5 млрд в год.

1996 — Компания 3Dfx выпускает первый в мире графический ускоритель Voodoo, прародитель всех современных видеокарт.

Принят стандарт USB.

Началось соревнование между браузерами Netscape и Internet Explorer.

В мире подключено к сети 12.8 млн. компьютеров и поднято 500 тыс. сайтов.

CompTek представила поисковую машину Yandex.

Поисковая система Google создана в качестве учебного проекта студентов Стэнфордского университета Ларри Пейджа и Сергея Брина. Они работали над поисковой системой BackRub, а в 1998 году на её основе создали новую поисковую систему Google. Корпорация была основана 2 сентября 1998 года, а домен Google.com зарегистрирован 15 сентября 1997 года.

1997 — Intel выпустила процессор Pentium II, построенный на базе Pentium Pro с поддержкой инструкции MMX.

AMD выпускает AMD K6, первый, не являющийся клоном Intel, однако в аппаратном и программном отношении он был совместим с Pentium и устанавливался в гнездо Socket 7.

На смену Baby AT пришёл форм-фактор ATX.

1998 — Sony выпускает MP3-плеер.

Microsoft анонсировала новую версию своей операционной системы Windows 98.

Intel выпустила Celeron — первый бюджетный процессор.

Apple выпускает компьютер iMac.

Special Interest Group представляет спецификацию Bluetooth.

Эпидемия вируса Win95.CIH, также известного как Чернобыль, ставшая самой разрушительной за все предшествующие годы.

1999 — Компания AMD представляет процессор Athlon.

Разработана первая видеокарта Radeon.

XXI Век

2000 — Огромное количество неспециалистов было введено в заблуждение мнимой «проблемой 2000». ИТ-компании во главе с Microsoft, предлагающие решение этой проблемы, получают многомиллионные прибыли.

Microsoft выпускает Windows Me (Millennium Edition) и Windows 2000.

Intel представляет Pentium 4 и Itanium, первый 64-разрядный IA-64 процессор для ПК.

Появление памяти DDR SDRAM.

Запущена первая система обмена MP3-файлами в Сети — Napster.

Самая массовая целенаправленная вирусная атака через Интернет — вирус I Love You поразил около 30 миллионов компьютеров во всем мире, уничтожив данные на 10 млрд. \$.

В этом же году число компьютеров, подключенных к Интернету, превысило 100.000.000; число сайтов — 30.000.000; число пользователей — 500.000.000. Объем торговых операций, совершенных в Интернете, достиг 1 триллиона долларов.

Появление первой программы для пиратского копирования DVD-дисков DeCSS.

Билл Гейтс покидает пост исполнительного директора Microsoft — его заменяет Стив Балмер.

Доменное имя business.com продано за \$7.500.000.

2001 — Intel представила первый процессор с рабочей частотой 2 ГГц, которым стала одна из версий Pentium 4.

Microsoft выпускает Windows XP Home и Professional.

Кризис доткомов. Массовые увольнения в большинстве предприятий виртуальной торговли и сетевого бизнеса. Рухнула большая часть мирового интернет-рынка.

Napster закрыт правообладателями после громкого судебного процесса.

Взломанный хакерами вариант видекодека Mpeg4 от Microsoft лег в основу системы видеокompresсии DivX.

Microsoft представляет прототип планшетного компьютера — Tablet PC.

Расмус Лерддорф разработал скриптовый язык PHP.

Intel выпускает процессоры Xeon для рабочих станций.

Первый torrent-клиент BitTorrent создан программистом Брэмом Коэном на языке Python.
 Джимми Уэйлс и Ларри Сэнгер запускают проект Wikipedia.
 Apple начинает продажи первого плеера с HDD – iPod.
2002 – Появление на рынке первого процессора Pentium 4, преодолевшего рубеж 3 ГГц.
 Intel выпустила Pentium M, процессор, разработанный специально для мобильных систем, обеспечивающий малое энергопотребление и при этом относительно высокое быстродействие.
 В Microsoft разработан новый язык программирования — C#, предназначенный для новой платформы .NET.
 Sony, Matsushita, Samsung, LG, Philips, Thomson, Hitachi, Sharp и Pioneer представили новый формат Blue-Ray.
 Чемпион мира Владимир Крамник встречается с Deep Fritz - программой для игры в шахматы. Матч завершается вничью. В списке гроссмейстеров, проигравших программе, можно найти имена Каспарова, Ананда, Широа и Deep Blue, которому ранее принадлежал чемпионский титул в компьютерном мире.
2003 – Компания AMD представила Athlon 64 — первый 64-разрядный процессор, ориентированный на использование в домашних и офисных системах.
 Появление SATA.
 21 ноября 2003 года начало работы старейшего и крупнейшего знаменитого трекера The Pirate Bay.
 Компания Skype была основана двумя предпринимателями — Никласом Зеннстрёмом из Швеции и Янусом Фриисом из Дании.
 Первый полнометражный стереофильм в формате IMAX — «Призраки Бездны» Джеймса Кэмерона.
2004 – Вышла первая версия Ubuntu Linux.
 Sony выпускает PlayStation Portable.
 18 сентября 2004 начал работу самый крупный и известный трекер Рунета – torrents.ru, сейчас rutracker.org
2005 – Первое применение боевых роботов непосредственно против людей в войне США и Ирака.
 Первая электронная бумага E-Ink.
 Появление первых двухядерных процессоров Pentium D и Athlon 64x2.
2006 – В январе Intel выпущен новый двухядерный процессор Core Duo, так пришёл конец эпохи Pentium.
 Компания Optware выпустила первый и единственный голографический накопитель Holographic Versatile Card (HVC). Емкость карты памяти – 30Гб, цена – примерно \$9, но устройство для чтения HVC стоит около \$1820, а устройство записи/чтения \$9000.
2007 – Hitachi представляет первый коммерческий накопитель емкостью 1 ТБ.
 Выпущена Windows Vista, одна из самых ожидаемых и неудачных ОС от Microsoft.
 Apple выпускает свой первый смартфон – iPhone.
2008 – Анонсирована DDR III.
 Intel выпускает первый процессор новой линейки Core – Core i7. Впервые в мире достигнута такая степень интеграции – все ядра, кэш и контроллер памяти (а в Core i7 8xx ещё и контроллер PCI-E) находятся на одном кристалле.
 Выпущен первый релиз Android.
2009 – Финальная спецификация USB 3.0.
 Выпущена избавленная от большинства ошибок версия Vista – Windows 7.
 Спутник с 3D-вещанием был запущен в США.
2010 – Выпуск планшета iPad.
 Червь Stuxnet поражает аппаратную часть компьютеров. В Иране этот червь поразил более 60000 компьютеров, причём 60% всех зараженных машин работали на объектах атомной промышленности этой страны, что парализовало всю атомную программу Ирана. Дополнительно по всему миру пострадало ещё почти 100.000 компьютеров. Это была первая успешная военная кибератака.
2011 – Умер Стив Джобс.
 Обнаружено, что в iOS 4 имеется недокументированная функция, которая записывает в скрытый файл информацию о географическом положении смартфонов iPhone и iPad, в Android тоже обнаружилась аналогичная функция, причем гуглофоны еще и передавали собранные сведения, включая географические координаты и цифровую подпись телефона, на серверы Google.
 Компания D-Way Systems продала первый коммерческий квантовый компьютер.

Основы квантовой физики

Классическая механика Ньютона способствовала развитию термодинамики и молекулярной физики. Практически все тепловые процессы стало возможным наглядно представить и объяснить с точки зрения механической аналогии. Из энергии механического движения молекул вытекали принципы работы паровых двигателей и тепловых машин вообще, законы состояния идеального газа и т.д. Казалось, что, при таком багаже знаний, легко объяснить закон теплового излучения тел, в частности, почему нагретое тело светится разным цветом в зависимости от температуры. Процесс нагревания и рассеиваемая при нем тепловая энергия хорошо описывались классической термодинамикой, процесс излучения света можно было также легко описать, пользуясь представлением о том, что световое излучение — это волна (как показал Гюйгенс). Так как это волна электромагнитная, соответственно излучение должно описываться классической электродинамикой Максвелла⁶⁵.

Удалось построить сносную модель для излучения всех видов тел, за исключением абсолютно черного тела⁶⁶ (АЧТ). Изначально к решению проблемы были применены чисто классические методы, которые дали ряд важных и верных результатов, однако полностью решить проблему не позволили, приведя, в конечном итоге, не только к резкому расхождению с экспериментом, но и к внутреннему противоречию — так называемой ультрафиолетовой катастрофе.

Из теории Максвелла следовало, что длина волны излучения должна быть кратной расстоянию между стенками полости (там должно помещаться целое число волн). Из термодинамики следует, что в состоянии равновесия каждая волна независимо от частоты будет нести одно и то же количество энергии E , определяемой температурой полости. Но поскольку ряд допустимых частот бесконечен, то и общая энергия излучения получается бесконечной.

Правильную формулу излучения АЧТ пытались вывести, угадать или подобрать (Релей-Джинс, Вин), но результаты не радовали — опыт показывает, что вторая формула Вина справедлива лишь в пределе высоких частот (малых длин волн). Закон Рэлея — Джинса предполагает квадратичное возрастание спектральной плотности излучения в зависимости от его частоты. На практике такой закон означал бы невозможность термодинамического равновесия между веществом и излучением, поскольку, согласно ему, вся тепловая энергия должна была бы перейти в энергию излучения коротковолновой области спектра.

В 1900 Макс Планк подогнал условия задачи под ответ и предположил, что энергия электромагнитной волны может изменяться только целыми порциями, а минимальная энергия, которую может нести волна, пропорциональна ее частоте. Поэтому в замкнутой полости число частот электромагнитного излучения не бесконечно, а ограничено: не может существовать волн, минимальная энергия которых превышает E . В самом деле, полость чёрного тела замкнута, поэтому в стационарном состоянии там могут существовать лишь стационарные стоячие электромагнитные волны. Чтобы так получилось, они своими узлами должны лежать на границах черного тела, а, следовательно, состоять из целого числа полуволн. Смущал тот факт, что энергия основной полуволны не могла быть любой, а должна была быть равной кратной, по предположению Планка, некоей малой величине, а минимальная энергия, которую может нести волна, пропорциональна её частоте. Из экспериментальных измерений энергии замкнутой полости Планк рассчитал коэффициент пропорциональности между минимальной энергией волны E и ее частотой ν : $E = h\nu$, который получил название постоянной Планка. Никто не понимал, почему энергия волн должна быть дискретной, за исключением того, что это работает, и проникновение в физический смысл постоянной Планка длилось многие десятилетия.

Стоит заметить, что Планк сам не понял, почему так происходит, он до конца жизни так окончательно и не поверил в эту, на его взгляд, ересь, однако формула работала с потрясающей точностью (а законы Вина и Рэлея-Джинса, как было доказано, являлись ее частными и неточными вариантами), и за это открытие он получил Нобелевскую премию в 1918 г.

Параллельно русский учёный А.Г. Столетов изучал фотоэффект (открытый в 1887 немецким физиком Генрихом Герцем: когда свет падает на некоторые металлы, то они испускают электроны, т.е. если кратко — проводимость вакуума при облучении светом металлического электрода). Эффект объяснили довольно быстро — свет вырывает электроны из металла. Однако А.Г. Столетов обнаружил нечто странное. Как положено в классической физике, сила тока зависела от интенсивности облучения (амплитуды волны). Столетов догадался поменять в схеме плюс на минус и обнаружил, что если понемногу увеличивать напряжение, то

⁶⁵ Той самой, что позже вошла в противоречие и с механикой — опыт Майкельсона показал, что скорость света не зависит от скорости его источника и является постоянной, что привело к отрицанию галилеевого сложения скоростей и родило СТО.

⁶⁶ Абсолютно чёрное тело (АЧТ) — физическая идеализация, применяемая в термодинамике, тело, поглощающее всё падающее на него электромагнитное излучение во всех диапазонах и ничего не отражающее. Несмотря на название, абсолютно чёрное тело само может испускать электромагнитное излучение любой частоты и визуально иметь цвет. Спектр излучения абсолютно чёрного тела определяется только его температурой. Важность абсолютно чёрного тела в том, что вопрос о спектре равновесного теплового излучения тел любого цвета и коэффициента отражения сводится к вопросу об излучении абсолютно чёрного. Наиболее чёрные реальные вещества, например сажа, поглощают до 99 % падающего излучения (то есть имеют альбедо, равное 0,01) в видимом диапазоне длин волн, однако инфракрасное излучение поглощается ими значительно хуже. Среди тел Солнечной системы свойствами абсолютно чёрного тела в наибольшей степени обладает Солнце.

фототок прекратится не сразу, а на строго определенном напряжении. Причём это напряжение зависит от длины волны излучения и материала электрода (и совсем не зависит от интенсивности!). А второе — если электрод облучали сильно красным и инфракрасным светом — фотоэффект не возникал вообще, вне зависимости от интенсивности. Проведя это, Столетов был в шоке, потому как согласно классической физике все должно быть наоборот — частота не влияет, а интенсивность влияет — представим себе волны, которые размывают берег. Вопрос повис почти на 5 лет.

А. Эйнштейн решил распространить планковскую дискретную модель энергии волны на свет. Он предположил в 1905, что свет представляет собой поток микроскопических частиц (названных в 1929 г. химиком Гильбертом Льюисом фотонами), причем энергия каждого фотона пропорциональна частоте света. Тогда, чтобы вырваться с поверхности металла, электроны должны испытать соударение с фотонами, обладающими определенным минимумом энергии. Поэтому если частота света будет слишком мала, то фотоны не будут нести энергию, достаточную для выбивания электронов из металла. Таким образом, А. Эйнштейн показал, что гипотеза Планка о дискретности энергии отражает фундаментальное свойство электромагнитных волн: они состоят из частиц — фотонов, которые представляют собой маленькие порции или кванты света. Дискретность энергии волн определяется тем, что они состоят из дискретных объектов.

Вот знаменитое уравнение фотоэффекта А. Эйнштейна (за которое он получил Нобелевскую премию)

$$h \cdot \nu = A_{out} + \frac{m_e v_e^2}{2},$$

где A_{out} — т. н. работа выхода (минимальная энергия, необходимая для удаления электрона из вещества), $\frac{m_e v_e^2}{2}$ — кинетическая энергия вылетающего электрона (в зависимости от скорости может вычисляться как кинетическая энергия релятивистской частицы, так и нет), $h\nu$ — планковская энергия падающего фотона. Из этой формулы следует существование красной границы фотоэффекта, то есть существование наименьшей частоты, ниже которой энергии фотона уже недостаточно, чтобы «выбить» электрон из металла. Суть формулы заключается в том, что энергия фотона расходуется на ионизацию атома вещества и на работу, необходимую для «вырывания» электрона, а остаток переходит в его кинетическую энергию. В 1923 американский физик Артур Комптон подтвердил корпускулярную природу света в экспериментах по рассеянию рентгеновских фотонов на свободных электронах (эффект Комптона). Его расчеты показали, что коэффициент пропорциональности между энергией и длиной волны фотона тоже равен постоянной Планка, которую он вычислил с точностью до 1%.

Рассматривать свет состоящим из потока частиц предложил еще Ньютон. Ему оппонировал голландский физик Христиан Гюйгенс, утверждавший, что свет — это волна. В начале XIX в. эксперименты английского физика Томаса Юнга (по интерференции света, пропускаемого через две щели) показали, что Ньютон ошибался, и свет — все-таки волна. Фейнман любил говорить, что вся квантовая механика может быть выведена из одного этого эксперимента. Можно было бы привести аналогию с морем, состоящим из отдельных молекул воды, которые совместным движением могут создавать волны. Но микромир устроен более тонко. Даже если через щели будем пропускать одиночные фотоны с большими временными интервалами, интерференционная картина за щелями сохранится. Как будто фотоны могут друг с другом сговориться двигаться так, чтобы за экраном чаще попадать в светлые полосы, чем в темные. Фотоэффект и опыт с двумя щелями показали, что фотоны обладают и волновыми, и корпускулярными свойствами одновременно.

Эксперименты действительно показывали, что происходит нечто странное. Один набор инструментов показывает мир, сделанный из отдельных шариков-частиц, а те же инструменты, но расположенные по-другому, показывают мир, состоящий из энергетических волн. Некоторое время физики даже говорили, как бы в шутку, но в то же время с долей отчаяния, о «волночастицах».

В 1923 молодой французский аристократ, князь Луи де Бройль объединил две гипотезы, что принесло ему Нобелевскую премию 1929 года. Эйнштейновское $E = mc^2$ связывает массу с энергией, но Планк связал энергию с частотой волн — $E = h\nu$. Объединяя эти два предположения, можно прийти к выводу, что масса должна иметь и волновое воплощение. В середине 20-х Клинтон Девиссон и Лестер Джермер, физики-экспериментаторы из лаборатории телефонной компании Bell, подтвердили волновые интерференционные свойства электронов при их рассеянии на атомах никеля. Расчеты показали, что длина волны электрона обратно пропорциональна его энергии с коэффициентом пропорциональности, равным постоянной Планка. Это было уже третье непонятное совпадение, и отсюда постоянная Планка стала приобретать универсальный и фундаментальный характер. Таким образом, выяснилось, что масса должна иметь и волновое воплощение. Соответственно, окружающая нас материя — это и волны, и частицы одновременно⁶⁷.

⁶⁷ Волны и частицы суть лишь приближение и попытка математическими средствами понять окружающую нас реальность. На самом деле все гораздо сложнее, в современной физике принято, что материя состоит из полей (т. е. имеет преимущественно волновую природу), но волны эти при взаимодействии друг с другом ведут себя так, что это порождает иллюзию взаимодействующих частиц. В результате: пока свет летит в пространстве, он ведет себя как волна — подвергается интерференции, дифракции, имеет поляризацию и т.д., когда свет (и любое ЭМ-излучение) взаимодействует с веществом, излучается или поглощается — его приходится считать потоком частиц и рассчитывать согласно квантовым эффектам.

Другая проблема возникла в 1911 году, когда Резерфорд открыл планетарную модель атома. Модель была подтверждена экспериментально, но опять вошла в противоречие с теорией Максвелла. Дело в том, что электрон движется по орбите и является заряженной частицей, а согласно электродинамике, такие частицы обязаны излучать (что тоже доказано!) и, как следствие — терять энергию, постепенно падая на ядро. А это означает, что наш мир не имеет права на существование.

Для объяснения структуры атома Нильс Бор в 1913 году предположил существование стационарных состояний электрона, в которых электрон не излучает, а его энергия может принимать лишь дискретные значения.⁶⁸ Этот подход, развитый Арнольдом Зоммерфельдом и другими физиками, часто называют старой квантовой теорией (1915—1924).

В рамках этой теории были приняты следующие утверждения:

- Энергия любой электромагнитной волны может изменяться только целыми порциями, а минимальная энергия, которую может нести волна, пропорциональна ее частоте (Планк).
- Свет представляет собой поток частиц фотонов, причем энергия каждого фотона пропорциональна частоте света (Эйнштейн).
- Масса и все вещество вообще имеет волновое и корпускулярное воплощение (де Бройль).
- Атом имеет планетарную структуру, электрон в его составе находится в стационарных состояниях, а его энергия может принимать лишь дискретные значения (Бор).

После этих постулатов встал вопрос — волны чего образуют материю? Австрийский физик Эрвин Шредингер, в порядке попытки опровержения данных постулатов, попытался вывести реальное уравнение, описывающее колебание частицы, без каких-либо попыток предположить, что же это за частица на самом деле. Он описал ее колебания т.н. волновой функцией, что, несмотря на его первоначальный скепсис, полностью согласовывалось с имеющимися данными. Так родилось уравнение Шредингера — основное уравнение квантовой механики. Однако, т.к. сам он благоразумно не делал предположений, что представляет собой элемент, который описывается его уравнением, то вопрос интерпретации понятия волновой функции лег на плечи иных физиков и породил длительные споры. В результате, что именно колеблется — не совсем понятно до сих пор, хотя уравнение работает отлично.

Шредингера изначально интересовал базовый физический принцип «наименьшего энергетического пути» — если шарик запустить по кривой к наклонной поверхности, путь его будет проходить по траектории с наименьшей потенциальной энергией. То же верно в отношении волны света, распространяющейся в неравномерной оптической среде. То же верно для электрического тока. Отто отметил, что формула описывает именно этот принцип для распространения волн. Первоначально порождённое описание получило название «волновая механика» — в результате позднего скрещивания с идеями Бора и Планка родилось современное понятие «квантовой механики».

Отказавшись от описания движения частицы с помощью траекторий, получаемых из законов динамики, и определив вместо этого волновую функцию, необходимо ввести в рассмотрение уравнение, эквивалентное законам Ньютона⁶⁹ и дающее рецепт для нахождения положения частицы в частных физических задачах:

$$-\frac{\hbar^2}{2m}\Delta\Psi(\vec{r},t) + E_p(\vec{r})\Psi(\vec{r},t) = i\hbar\frac{\partial}{\partial t}\Psi(\vec{r},t)$$

В квантовой физике вводится комплекснозначная функция Ψ , описывающая чистое состояние объекта, которая называется волновой функцией. В разных школах принято считать по-разному, что же она описывает на самом деле — т.е. все сходятся на том, что «состояние волночастицы», но расходятся в том что понимать под состоянием и под частицей. Вне зависимости от этого, поведение системы в чистом состоянии полностью описывается с помощью волновой функции.

В уравнении Шредингера $\hbar = h/2\pi$ — приведенная постоянная Планка, $\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \dots + \frac{\partial^2}{\partial x_n^2}$ — дифференциальный оператор Лапласа, $E_p(\vec{r})$ — внешняя по отношению к частице потенциальная энергия в точке $\vec{r} = (x_1, \dots, x_n)$, $i = \sqrt{-1}$.

Получившееся уравнение является дифференциальным уравнением второго порядка в частных производных, его собственные решения можно было истолковать как систему стоячих электронных волн в окрестности ядра, определенным образом соответствующую системе стационарных состояний.

⁶⁸ По модели Бора-Резерфорда, находясь на естественных стационарных орбитах, электрон не излучает энергии вообще. Если он поглотит квант энергии, то перейдет в возбужденное состояние, поднявшись на уровень выше. Долго он там не пробудет и излучит квант обратно, вернувшись на стационарную орбиту (на этих принципах работают лазеры, а также транзисторы и все полупроводниковые приборы). Естественно все это описывается в рамках того же закона Планка.

⁶⁹ Уравнение Шредингера предназначено для частиц без спина, движущихся со скоростями, много меньшими скорости света. В случае быстрых частиц и частиц со спином используются его обобщения (уравнение Клейна — Гордона, уравнение Паули, уравнение Дирака и др.) — уравнения квантовой теории поля, объединяющей СТО и квантовую механику.

Далее квантовая механика разделилась на несколько направлений:

- В 1926 немецкий физик Макс Борн внес предположение (усиленное главой «копенгагенской школы» Нильсом Бором с коллегами), что волны, описываемые Ψ , должны интерпретироваться как волны вероятности: вероятность обнаружить электрон в заданной точке равна квадрату амплитуды его волны в этой точке:



- Вернер Гейзенберг с 1925 (совместно с работами Паскуаля Йордана) начал развивать другой формализм описания микромира, названный им матричной механикой, которая описывала квантовые явления с помощью таблиц наблюдаемых величин. Эти таблицы представляют собой определенным образом упорядоченные математические множества, называемые матрицами, над которыми по давно известным правилам можно производить различные математические операции. Тем самым микрореальность допускала и другие математические описания и находила прямое отражение в математике. Метод квантовой механики, оперирующий прямо с матрицами и не имеющий дела с промежуточными величинами — волновыми функциями, более компактен и часто быстрее приводит к желаемым результатам. Метод же волновой механики лучше удовлетворяет интуиции физиков и лучше согласуется с образом их мыслей. Однако интерпретация этой квантовой механики (с формальными операциями над матрицами), исходя из обычных кинематических и механических представлений, оказалась невозможной. Такие понятия как «положение», «скорость», «траектория» и связанные с ними представления о непрерывном течении событий и о причинности словно повисли в воздухе. Гейзенберг особенно настаивал на отказе от каких-либо простых наглядных представлений или моделей в пользу только таких свойств, которые могли быть определены из эксперимента, и эта концепция была развита копенгагенской школой. Шредингер же, наряду с Эйнштейном и де Бройлем, был среди противников копенгагенской интерпретации квантовой механики, поскольку его отталкивало отсутствие в ней детерминизма.

Проблема детерминизма заключается вот в чем: по первым выводам Юнга, получалось, что фотон имеет свойства и волны и частицы одновременно, но оказалось, что он существует, как частица, а взаимодействует как волна. В классическом опыте — дифракции света, свет проходит через две щели и падает на экран, где появляются темные и светлые интерференционные полосы. Это можно объяснить тем, что в некоторых местах световые волны взаимно усиливаются, а в других — гасятся. С другой стороны, эксперимент показывает, что свет обладает и свойствами потока частиц, а такие объекты, как электроны могут проявлять и волновые свойства и тоже могут давать интерференционную картину. А что будет, если выпускать по одному фотону или электрону? Электрон-частица должен был пролететь через одну из щелей, но в реальности единый и неделимый квант пролетает через обе щели и интерферирует на экране сам с собою. Что будет, если поставить детекторы частиц возле каждой из щелей и попытаться поймать части кванта при его прохождении через обе щели? В этом случае квант будет всегда пойман выходящим из одной из щелей, но никогда из обеих (что логично, если учесть, что квант неделим по определению). Интерференционная картина на экране при этом исчезает (заменяется нормальным распределением).

Но и это еще не все. Что будет, если установить только один детектор возле одной из щелей? А происходит то, что даже если квант не был пойман детектором (пролетел через другую щель), интерференция на экране все равно исчезает (то есть квант «узнал», что его состояние измеряют у другой щели, и отказался интерферировать, ведя себя, как частица, а не как волна). Чтобы зарегистрировать, через какую из щелей прошел тот или иной электрон, необходимо перед ними поставить измерительный прибор, который работает на чем? На тех же самых электромагнитных волнах. То есть, чтобы зарегистрировать электрон в пространстве, нам нужно воздействовать на него. Воздействуя на него, мы меняем его состояние. А раз меняется состояние, меняется и интерференционная картинка — она исчезает. Если мы понижаем воздействие до минимума, тут же начинаем наблюдать интерференционную картинку, но уже не можем сказать, через какую из щелей прошел электрон. В итоге никакого влияния наблюдения на эксперимент нет, и магии тоже нет, и все просто: мы не можем измерить положение электрона, не изменив его состояния. В результате мы не сможем одновременно узнать положение частицы и ее скорость — это и есть нарушение детерминизма, описанное Гейзенбергом как принцип неопределенности. Важнейший философский вывод из квантовой механики заключается в принципиальной неопределенности результатов измерения и, следовательно, невозможности точного предвидения будущего. Математически соотношение неопределенностей выглядит так: точность определения положения Δx (здесь Δ понимается, как разность состояний) обратно пропорциональна точности определения скорости (импульса Δp , а коэффициент пропорциональности равен постоянной Планка: $\Delta x \Delta p \sim h$). В этом и есть ее физический смысл.

Исследования 90-х показали (Джон Белл, Алан Аспект и др.), что это соотношение носит фундаментальный характер: если поместить электрон в коробку и начать сближать ее стенки, то он начнет метаться с все большей и менее определенной скоростью. Соотношение неопределенности лежит в основе еще одного потрясающего явления — квантового туннелирования: низкоэнергетические частицы могут «просачиваться» через непреодолимый высокоэнергетический барьер (если стрелять пулями по стальной стене, то небольшая часть пуль проскочит через стену).

Гейзенберг показал также обратную пропорциональную связь между точностью измерения энергии частиц ΔE и временем ее измерения Δt : $\Delta E \Delta t \sim \hbar$. В дальнейшем это привело к концепции виртуальных частиц: на очень короткое время можно «занять» (а затем «вернуть») очень большую энергию с порождением кратковременных частиц. Невероятная энергетическая активность в микромире все более возрастает по мере уменьшения масштабов расстояния и времени. На ультрамикроскопических масштабах всплески энергии столь велики, что вследствие общей теории относительности рушится привычная гладкая топология пространства-времени (флуктуации массы, а следовательно — и тяготения становятся огромными) — в ней возникают ручки, дырки и т.п. — хаос квантовой пены (по выражению Джона Уилера). Но поскольку заем и возврат в среднем компенсируют друг друга, пустое макроскопическое пространство продолжает выглядеть тихим и спокойным. Так из-за границы вакуума — областей отрицательной энергии, известных как море Дирака на ничтожнейшие доли секунды появляются виртуальные частицы, например, переноса взаимодействие — и растворяются без следа. В силу малости времени Вселенная не обращает внимания на такое нарушение законов сохранения, и вакуум на самом деле не пустота, а место, заполненное виртуальными частицами.

Итак, существовала совокупность экспериментов. Существовала логически непротиворечивая, математически изящная теория (даже две, Шрёдингер показал, что волновая механика и матричная механика математически эквивалентны⁷⁰), корректно описывающая эксперименты. Но отсутствовали понятия, с помощью которых можно было бы описать объект исследования. Материальная точка, траектория, волна — это не образы. В системе классической физики это полноценные понятия со своей логикой и онтологией, позволявшие говорить о том, что собственно мы наблюдаем и описываем. В новой механике все эти объекты превращались из вещей, поведение которых описывается в теории, в язык описания неведомых объектов. Было неясно, что же можно поставить на их место. Тем не менее, адекватность абстрактных математических конструкций и физической реальности сама по себе вызывает крайне удивление и заставляет глубоко задуматься.

Когда родилась квантовая механика, то даже её создатель Макс Планк так и не принял в глубине души всей причудливости этой науки. Эйнштейн же попросту считал ее абсурдной теорией, называл ее «безумием». Кризис восприятия был так велик, потому что с появлением квантовой механики под ногами физиков полностью пропала опора в виде наглядных схем и понятных интерпретаций. Физика все тем больше становилась математикой, то есть формульной абстракцией, которую иногда даже невозможно проверить опытным путем, причем формулы порой выдавали абсурдные решения. На стороне Эйнштейна сражались такие физики, как Планк и Шрёдингер, а их врагами выступали Нильс Бор, а также Гейзенберг, Борн и Йордан. Суть спора состояла в том, действительно ли миром правит принципиальная неопределенность, или же мы просто не знаем некоторых свойств микрочастиц, которые (если бы их можно было измерить) позволяли бы стопроцентно предсказать поведение квантов в каждой конкретной ситуации. В 1935 году Эйнштейн вместе с Борисом Подольским и Натаном Розеном написал статью «Можно ли считать квантово-механическое описание физической реальности полным?», в которой описал мысленный эксперимент, который впоследствии был назван парадоксом Эйнштейна—Подольского—Розена (ЭПР-парадокс).

Предполагая, что причиной неопределенности является то, что, производя измерение одной величины, вносится принципиально неустранимые возмущения в её движение и искажение значения другой величины, можно предложить гипотетический способ, которым соотношение неопределенностей можно обойти. Если взять две одинаковые частицы, образовавшиеся в результате распада третьей частицы, то в этом случае их импульсы должны быть связаны. Это даёт возможность измерить импульс одной частицы и по закону сохранения импульса рассчитать импульс второй, не внося в её движение никаких возмущений. Поэтому, измерив координату второй частицы, можно получить для этой частицы значения двух неизмеримых одновременно величин, что по законам квантовой механики невозможно. Исходя из этого, можно заключить, что соотношение неопределенностей не является абсолютным, а законы квантовой механики являются неполными и должны быть в будущем уточнены.

Аргумент оказался удачным, Бор не нашелся с ответом, а спор перешел в стадию, когда все остаются при своем мнении, но не имеют веских аргументов. Над обоими лагерями оппонентов нависла тень, и только

⁷⁰Шрёдингер рассказывал, что ему никак не удавалось получить наблюдаемые в эксперименте спектры атомов исходя из уже известной двойственности волна—частица, так как, хотя он уже и написал «уравнение Шрёдингера», спектр неизменно получался непрерывным (как в интеграле, а не в ряде Фурье), из-за того что область, где рассматривалось уравнение, простиралась, естественно, неограниченно далеко. Однако Герман Вейль, которому Шрёдингер рассказал о своих трудностях, подсказал ему, что он, Вейль, уже один раз преодолел подобную же трудность в теории упругости, где он рассматривал колебания и волны в неограниченных областях: для получения дискретного спектра нужно наложить граничные условия на бесконечности, например, потребовать, чтобы пси-функция была интегрируема с квадратом модуля. Шрёдингер немедленно последовал этому совету, получил требуемый спектр атома водорода, и волновая квантовая механика быстро сменила предшествующую ей матричную.

экспериментальная проверка могла показать, кто прав. Но к всеобщему счастью, в то время еще никто не знал, как провести такой эксперимент. В 1951 году физик Бом придумал план постановки требуемого эксперимента. В 1964 году другой физик Белл вывел т.н. неравенство Белла, которое позволило формализовать поставленную задачу. Благодаря этому стало возможным в эксперименте получить определённую величину, описывающую корреляции между удалёнными измерениями, и на её основе сказать, имеет ли смысл описывать квантовые явления вероятностно или детерминировано.

Но лишь в 1972 году в Калифорнийском университете, наконец, были проведены опыты, зафиксировавшие нарушение неравенств Белла и тем самым подтвердившие правоту квантовой механики, и, соответственно, неправоту лагеря Эйнштейна. Как обычно в таких ситуациях, все радостно признали результаты эксперимента, но опять не договорились, что они значат на самом деле.

Итак, эксперимент был проведён, неравенства Белла были нарушены, ЭПР-парадокс однозначно подтвердил правильность квантовой механики. Принцип неопределённости Гейзенберга нельзя нарушить, а это означало, что внесение возмущения в импульс и координату одной частицы непременно влияет на другую, на каком бы далёком расстоянии она ни находилась. Это, ни много ни мало, «кошмарное дальное действие», как это называл Эйнштейн, «телепатия», но это факт. И тут над постулатами ОТО и всеми классическими представлениями нависла уже нешуточная беда.

Квантовая механика окончательно стала нелокальной теорией, вновь стала полной противоположностью ОТО. Свойство нелокальности вызывает существование корреляций между состояниями подсистем (например, 2-х частиц, на которые распалась одна – измерив импульс одной, мы автоматически сделаем невозможным измерение координаты не только нашей частицы, но и ее близнеца, как бы далеко он не находился, такие частицы называют связанными или сцепленными⁷¹) исходной системы, на каких бы расстояниях друг от друга они не находились. Поэтому появляется возможность мгновенного определения квантового состояния в одном месте на любом расстоянии путём измерения запутанного с ним состояния в другом месте и, соответственно, его передача с бесконечной скоростью — квантовая телепортация.

Хоть связанные кванты и коррелируют мгновенно, но передать полезную информацию таким волшебным образом всё-таки невозможно. На то, чтобы измерить квантовое состояние и сопоставить с измерениями другого связанного кванта, нужно потратить время и воспользоваться вполне обычными средствами связи. Тем не менее, сам факт, что две частицы, разнесенные на сколь угодно большое расстояние способны, хоть и особым образом, но мгновенно влиять друг на друга, заставил некоторых ученых усомниться в истинности данной теории.

Сцепленность приводит к интересным взаимоотношениям с принципом относительности, который утверждает, что информация не может переноситься с места на место быстрее, чем со скоростью света. Хотя две системы могут быть разделены большим расстоянием и быть при этом сцепленными, передать через их связь полезную информацию невозможно, поэтому причинность из-за сцепленности не нарушается. Это происходит по двум причинам:

- результаты измерений в квантовой механике носят принципиально вероятностный характер;
- теорема о запрете клонирования⁷² запрещает статистическую проверку сцепленных состояний.

⁷¹ Состояние, при котором изменение одной подсистемы сказывается на другой. Например, можно приготовить две частицы, находящиеся в едином квантовом состоянии, так, что когда одна частица наблюдается в состоянии со спином, направленным вверх, то спин другой оказывается направленным вниз, и наоборот, и это несмотря на то, что, согласно квантовой механике, предсказать, какие фактически каждый раз получатся направления, невозможно. Однако то, что понимается под информацией в классическом смысле, не может быть передано через сцепленность из-за статистического характера передаваемой информации. Квантовая сцепленность является основой таких технологий, как квантовый компьютер и квантовая криптография, а также она была использована в опытах по квантовой телепортации. В философском плане данное явление представляет собой одно из наиболее революционных свойств квантовой теории, так как можно видеть, что корреляции, предсказываемые квантовой механикой, совершенно несовместимы с представлениями о, казалось бы, очевидной локальности реального мира, при которой информация о состоянии системы может передаваться только посредством её ближайшего окружения. Различные взгляды на то, что происходит во время процесса квантовомеханического сцепления, приводят к различным интерпретациям квантовой механики. Математически сцепленность представляет собой нефакторизуемое многочастичное квантовое состояние (то есть состояние, которое нельзя «разложить на множители» — представить в виде произведения нескольких состояний). Простая аналогия из школьной математики: $a^2 - b^2$ может быть представлена в виде произведения $(a + b)(a - b)$, в то время как $a^2 + b^2$ разложить на множители (над полем вещественных чисел) невозможно. Это означает, что составляющие квантовое сцепленную систему объекты не имеют собственных волновых функций и векторов квантовых состояний.

⁷² Утверждение квантовой теории о невозможности создания идеальной копии произвольного неизвестного квантового состояния. Теорема была сформулирована Вуттерсом, Зуреком и Диексом в 1982 году и имела огромное значение в области квантовых вычислений, квантовой теории информации и смежных областях. Состояние одной квантовой системы может быть сцепленным с состоянием другой системы. Например, создать сцепленное состояние двух кубитов (квантовых битов, которыми оперирует квантовый компьютер) можно с помощью однокубитного преобразования Адамара и двухкубитного квантового вентиля C-NOT (квантовый вентиль реализующий операцию сходную с классическим XOR - сложением по модулю 2 или, строгой дизъюнкцией — логическая операция, результат выполнения операции является истинным только при условии, если является истинным в точности один из аргументов). Результатом такой операции не будет клонирование, поскольку результирующее состояние нельзя описать на языке состояний подсистем (состояние является нефакторизуемым). Клонирование — это такая операция, в результате которой создается состояние, являющееся тензорным произведением идентичных состояний подсистем. Хотя создание точных копий неизвестного квантового состояния невозможно, можно тиражировать его неточные копии. Для этого нужно привести исходную систему во взаимодействие с большей вспомогательной системой и провести специальное унитарное преобразование комбинированной системы, в результате которого несколько компонентов большей системы станут приблизительными копиями исходной. Такой процесс может быть использован для атаки на квантовые криптографические системы, а также для других целей в квантовых вычислениях.

И вот этот самый принцип связности, наконец, можно применить к созданию квантового компьютера. Полноценный квантовый компьютер является пока гипотетическим устройством, сама возможность построения которого связана с серьёзным развитием квантовой теории в области многих частиц и сложных экспериментов; эта работа лежит на переднем крае современной физики.

Почему же все эти противоестественные вещи работают? В связи с невозможностью договориться, что же всё-таки происходит в микромире, физики выдвинули большое количество теорий — иными словами, если свести двух учёных и показать им формулы, то они будут согласно кивать, но если попросить их объяснить, что же это значит, они, скорее всего, подерутся.

Копенгагенская интерпретация — наиболее популярная интерпретация в современной квантовой механике. Утверждает, что в квантовой механике результат измерения принципиально недетерминирован, а вероятностный характер предсказаний квантовой механики принципиально неустраним. Копенгагенская интерпретация отбрасывает вопросы типа «где была частица до того, как я зарегистрировал её местоположение» посредством уточнения, что этот вопрос некорректен. Последователи интерпретации уверены, что именно процесс измерения случайно выбирает в точности одну из возможностей, допустимых волновой функцией данного состояния, а волновая функция мгновенно изменяется, чтобы отразить этот выбор.

Опыт Юнга ставит следующие вопросы:

1. Законы квантовой механики говорят о том, где частицы попадут в экран статистически и дают возможность рассчитать местоположение светлых полос, куда скорее всего попадёт много частиц, и местоположение тёмных полос, куда скорее всего попадёт мало частиц. Однако для отдельной частицы законы квантовой механики не способны предсказать, где она окажется фактически. Каковы в таком случае правила поведения отдельных частиц?

2. Что происходит с частицей между моментом испускания и моментом регистрации? Создаётся впечатление, что частица претерпевает взаимодействие с обеими щелями, и это кажется противоречащим тому, как может себя вести точечная частица, тем более, что когда частица регистрируется, она оказывается точечной.

3. Что заставляет частицу переключаться от статистического к нестатистическому поведению и обратно? Когда частица летит сквозь щели, её поведение описывается нелокализованной волновой функцией, которая одновременно проходит через обе щели. Когда же частица регистрируется, никогда не получается размытый волновой пакет, но всегда фиксируется точечная частица.

Копенгагенская интерпретация отвечает на эти вопросы так:

1. Вероятностный характер предсказаний квантовой механики принципиально неустраним, то есть, он вовсе не говорит о том, что наши знания ограничены, что мы не знаем значений каких-то скрытых переменных. В классической физике вероятность использовалась для описания результатов типа подбрасывания игральной кости, хотя фактически этот процесс считался детерминированным. То есть, вероятности использовались вместо неполного знания. Напротив, копенгагенская интерпретация утверждает, что в квантовой механике результат измерения принципиально недетерминирован.

2. Физика — это наука о результатах измерительных процессов. Измышления на тему, что происходит за ними, неправомерны. Копенгагенская интерпретация отбрасывает вопросы типа «где была частица до того, как я зарегистрировал её местоположение» как бессмысленные.

3. Акт измерения вызывает мгновенное схлопывание, «коллапс волновой функции». Это означает, что процесс измерения случайно выбирает в точности одну из возможностей, допустимых волновой функцией данного состояния, а волновая функция мгновенно изменяется, чтобы отразить этот выбор.

Оригинальная формулировка копенгагенской интерпретации породила ряд вариаций; наиболее уважаемая основана на подходе непротиворечивых событий («Копенгаген прав?») и понятии квантовой декогеренции, которая позволяет рассчитывать нечёткую границу между «микро-» и «макро-» мирами. Другие вариации различаются степенью «реалистичности» волнового мира.

Полнота квантовой механики (тезис 1) была подвергнута сомнению в мысленном эксперименте Эйнштейна, Подольского и Розена (ЭПР-парадокс), который был предназначен для того, чтобы доказать, что скрытые параметры должны существовать, чтобы теория не приводила к нелокальному и мгновенному «дальнодействию». Однако проверка ЭПР-парадокса на опыте при помощи неравенств Белла показала, что квантовая механика верна, и что различные теории локальных скрытых параметров не согласуются с экспериментом. Из всех трёх тезисов с физической точки зрения наиболее проблематичен последний, поскольку он ставит процессы измерения в особое положение, но не определяет ясно, что они такое, и не указывает на их отличительные черты.

Многие физики и философы не соглашались с копенгагенской интерпретацией, потому что она не детерминистична и потому, что вводит неопределённое понятие измерения, которое превращает вероятностные функции в достоверные результаты измерений. Иллюстрируя это, Эйнштейн писал Бору «я убеждён, что Бог не бросает кости», а также восклицал в беседе с Абрахамом Пайсом: «Вы и вправду думаете, что Луна существует лишь когда Вы на неё смотрите?». Бор отвечал ему «Эйнштейн, не указывайте Богу, что делать». Эрвин Шрёдингер придумал знаменитый мысленный эксперимент про кота Шрёдингера, которым он хотел показать неполноту квантовой механики при переходе от субатомных систем к макроскопическим.

Парадокс состоит вот в чем. Возьмем, к примеру, одиночный радиоактивный атом. По законам квантовой механики он спонтанно распадается в непредсказуемый заранее момент времени. Поэтому его волновая функция представляет сумму двух компонент: одна описывает целый атом, а другая — распавшийся. Вероятность, соответствующая первой, убывает, а второй — растет. Физики в такой ситуации говорят о суперпозиции двух несовместимых между собой состояний. Если проверить состояние атома, произойдет коллапс его волновой функции, и атом с определенной вероятностью окажется либо целым, либо распавшимся. Но в какой момент происходит этот коллапс — когда измерительный прибор взаимодействует с атомом или когда о результатах узнает наблюдатель-человек? Оба варианта выглядят непривлекательно. Из первого следует неприемлемый вывод о том, что атомы измерительного прибора чем-то отличаются от остальных, раз под их влиянием происходит коллапс волновой функции вместо образования запутанного состояния, как должно быть при взаимодействии квантовых частиц. Второй вариант вносит в теорию так нелюбимый физиками субъективизм. Приходится согласиться, что сознание наблюдателя (тело его с точки зрения квантовой механики — все тот же прибор) непосредственно влияет на волновую функцию, то есть на состояние квантового объекта. Эта проблема была заострена Эрвином Шрёдингером в форме знаменитого мысленного эксперимента. Поместим в ящик кота и устройство с ядом, которое срабатывает при распаде радиоактивного атома. Закроем ящик и подождем, пока вероятность распада достигнет, скажем, 50%. Поскольку никакой информации из ящика к нам не поступает, находящийся в нем атом описывается как суперпозиция целого и распавшегося. Но теперь состояние атома неразрывно связано с судьбой кота, который, до тех пор пока ящик остается закрытым, пребывает в странном состоянии суперпозиции живого и мертвого. Но стоит только вскрыть ящик, мы увидим либо голодное животное, либо бездыханный труп, причем, скорее всего, окажется, что в таком состоянии кот пребывает уже некоторое время. Выходит, пока ящик был закрыт, в нем параллельно развивались как минимум две версии истории, но достаточно одного осмысленного взгляда внутрь ящика, чтобы реальной осталась лишь одна из них.

Аналогично вызывает проблемы необходимый «мгновенный» коллапс волновой функции во всем пространстве. Теория относительности Эйнштейна говорит, что мгновенность, одновременность имеет смысл только для наблюдателей, находящихся в одной системе отсчета. Не существует единого для всех времени, поэтому мгновенный коллапс тоже остается неопределенным.

Многомировая интерпретация — это интерпретация квантовой механики, которая предполагает существование «параллельных вселенных», в каждой из которых действуют одни и те же законы природы и которым свойственны одни и те же мировые постоянные, но которые находятся в различных состояниях. При проведении любого квантового эксперимента вселенная расщепляется на столько вселенных, сколько есть возможных исходов эксперимента, и каждый из результатов стопроцентно реализуется в одной из них, а наблюдатель, оказавшись опять же в одной из них, видит свой один конкретный результат. В случае с «двухщелевым опытом» (опыт Юнга) происходит следующее: при подлете к щели вселенная раздваивается, и фотон вылетит из той щели, во вселенной которой окажется наблюдатель (на самом деле фотон вылетит и из второй щели, но об этом никто не узнает, поскольку наблюдателя там, увы, нет). Соответственно наблюдатель констатирует, что опыт удался. И наоборот: если наблюдатель окажется в той вселенной, где фотон не вылетит (то есть на самом деле он вылетит, только в той вселенной, где наблюдателя нет), то наблюдатель констатирует, что опыт не удался. Согласно мнению противников, данная гипотеза считается самой научно-фантастической. Тем не менее, множество авторитетных ученых признают данную гипотезу имеющей право на существование.

Эту идею предложил в 1957 году американский физик из Принстонского университета Хью Эверетт III. В своей диссертации он поставил на первое место принцип линейности, а значит, и непрерывность действия линейных законов квантовой механики. Это привело Эверетта к выводу, что наблюдателя нельзя рассматривать в отрыве от наблюдаемого объекта, как некую внешнюю сущность.

В момент измерения наблюдатель вступает во взаимодействие с квантовым объектом, и после этого ни состояние наблюдателя, ни состояние объекта не могут быть описаны отдельными волновыми функциями: их состояния спутываются, и волновую функцию можно написать только для единого целого — системы «наблюдатель + наблюдаемое». Чтобы завершить измерение, наблюдатель должен сопоставить свое новое состояние с прежним, зафиксированным в его памяти. Для этого возникшую в момент взаимодействия запутанную систему надо вновь разделить на наблюдателя и объект. Но сделать это можно по-разному. В результате получаются разные значения измеряемой величины, но, что еще более интересно, разные наблюдатели. Выходит, что в каждом акте квантового измерения наблюдатель как бы расщепляется на несколько (возможно бесконечно много) версий. Каждая из этих версий видит свой результат измерения и, действуя в соответствии с ним, формирует собственную историю и свою версию Вселенной. С учетом этого интерпретацию Эверетта часто называют многомировой, а саму многовариантную Вселенную — Мультиверсом (чтобы не путать ее с космологическим Мультиверсом — множеством независимых миров, образующихся в некоторых моделях Вселенной, — некоторые физики предлагают называть ее Альтерверсом).

Идея Эверетта не проста и нередко трактуется ошибочно. Чаще всего можно услышать, будто при каждом столкновении частиц вся Вселенная разветвляется, порождая множество копий по числу возможных исходов столкновения. На самом деле квантовый мир, по Эверетту, ровно один. Поскольку все его частицы прямо или косвенно взаимодействовали друг с другом и находятся поэтому в запутанном состоянии, его

фундаментальным описанием является единая мировая волновая функция, которая плавно эволюционирует по линейным законам квантовой механики. Этот мир столь же детерминирован, как лапласовский мир классической механики, в котором, зная положения и скорости всех частиц в определенный момент времени, можно рассчитать все прошлое и будущее. В мире Эверетта бесчисленное множество частиц заменено сложнейшей волновой функцией. Это не приводит к неопределенностям, поскольку Вселенную никто не может наблюдать извне. Однако внутри можно бесчисленным множеством способов разделить ее на наблюдателя и окружающий мир.

Понять смысл интерпретации Эверетта помогает такая аналогия. Представьте себе страну с многомиллионным населением. Каждый ее житель по-своему оценивает происходящие события. В некоторых он прямо или косвенно принимает участие, что меняет как страну, так и его взгляды. Формируются миллионы разных картин мира, которые своими носителями воспринимаются как самая настоящая реальность. Но при этом есть еще и сама страна, которая существует независимо от чьих-то представлений, обеспечивая возможность для их существования. Точно так же единая квантовая Вселенная Эверетта дает место для огромного числа независимо существующих классических картин мира, возникающих у разных наблюдателей. И все эти картины, согласно Эверетту, совершенно реальны, хотя каждая существует лишь для своего наблюдателя.

На диссертацию Эверетта мало кто обратил внимание. Сам Эверетт еще до защиты принял приглашение от военного ведомства, где возглавил одно из подразделений, занимающихся численным моделированием последствий ядерных конфликтов, и сделал там блестящую карьеру. Его научный руководитель Джон Уилер поначалу не разделял взглядов своего воспитанника, но они нашли компромиссный вариант теории, и Эверетт представил ее для публикации в научный журнал *Reviews of Modern Physics*. Редактор Брайс Девитт отнесся к ней весьма негативно и намеревался отклонить статью, но потом неожиданно стал горячим сторонником теории, и статья вышла в июньском номере журнала за 1957 год. Однако с послесловием Уилера: я, мол, не думаю, что все это правильно, но это как минимум любопытно и не бессмысленно. Уилер настаивал, что теорию необходимо обсудить с Нильсом Бором, но тот фактически отказался ее рассматривать, когда в 1959 году Эверетт полтора месяца провел в Копенгагене.

В известном смысле Эверетту не повезло. Его работа терялась в потоке первоклассных публикаций, выполненных в то же самое время, к тому же была слишком «философской». Сын Эверетта, Марк, как-то сказал: «Отец никогда, никогда не говорил со мной о своих теориях. Он был для меня незнакомцем, существующим в каком-то параллельном мире. Я думаю, что он был глубоко разочарован тем, что он знал про себя, что он гений, но никто в мире больше об этом и не подозревал». В 1982 году Эверетт умер от сердечного приступа.

Сейчас даже трудно сказать, благодаря кому она была извлечена из небытия. Вероятнее всего, это произошло, когда все те же Брайс Девитт и Джон Уилер пытались построить одну из первых «теорий всего» — теорию поля, в которой квантование уживалось бы с общим принципом относительности. Потом на необычную теорию положили глаз фантасты. Но только после смерти Эверетта начался настоящий триумф его идеи (правда, уже в формулировке Девитта, от которой десятилетие спустя категорически откrestился Уилер). Стало казаться, что многомировая интерпретация обладает колоссальным объяснительным потенциалом, позволяя дать внятную трактовку не только понятию волновой функции, но и наблюдателю с его загадочным «сознанием». В 1995 году американский социолог Дэвид Роб провел опрос среди ведущих американских физиков, и результат был ошеломляющим: 58% назвали теорию Эверетта верной.

Наблюдателем может быть любая система, например, вычислительная машина, помнящая свои прежние состояния и сравнивающая их с новыми. «Как хорошо известно людям, работающим со сложными автоматами, фактически весь общепринятый язык субъективного опыта полностью применим к таким машинам», — пишет Эверетт в своей диссертации. Тем самым он уходит от вопроса о природе сознания. Но его последователи уже не были склонны к такой осторожности. Наблюдателя стали все чаще рассматривать как мыслящее и наделенное волей сознание, а не просто как сенсор с памятью. Это открывает простор для столь же интересных, сколь и спорных попыток объединить в одной концепции традиционную объективистскую физику и различные эзотерические представления о природе человеческого сознания.

Например, доктор физико-математических наук М. Менский из Физического института им. П.Н. Лебедева РАН активно развивает свою расширенную концепцию Эверетта, в которой отождествляет сознание с самим процессом разделения альтернатив. Физическая реальность имеет чисто квантовую природу и представлена единой мировой волновой функцией. Однако рационально мыслящее сознание, по Менскому, не способно непосредственно воспринимать ее и нуждается в «упрощенной» классической картине мира, частью которой оно себя воспринимает и которую само создает (в этом заключается его природа). При определенной подготовке, проявляя свободу воли, сознание способно более или менее произвольно выбирать, какую из бесконечного числа классических проекций квантовой Вселенной оно будет «проживать». Со стороны такой выбор может восприниматься как «вероятностное чудо», при котором «маг» способен оказаться именно в той классической реальности, которая ему желательна, даже если ее реализация маловероятна. В этом Менский усматривает связь своих идей с эзотерическими учениями. Он также вводит понятие «сверхсознания», которое в те периоды, когда сознание отключается (например, во сне, в трансе или медитации),

способно проникать в альтернативные эвереттовские миры и черпать там информацию, принципиально недоступную рациональному сознанию.

Другой подход уже не первое десятилетие развивает профессор Гейдельбергского университета Хайнц-Дитер Це. Он предложил многоразумную интерпретацию квантовой механики, в которой наряду с материей, описываемой волновой функцией, имеются сущности иной природы — «разумы» (minds). С каждым наблюдателем ассоциировано бесконечное семейство таких «разумов». При каждом эвереттовском расщеплении наблюдателя это семейство тоже делится на части, следуя вдоль каждой ветви. Пропорция, в которой они делятся, отражает вероятность каждой из ветвей. Именно «разумы», по мнению Це, обеспечивают самотождественность сознания человека, например, проснувшись поутру, вы узнаете себя той же личностью, что и ложась спать вчера.

Идеи Це пока не нашли широкого признания у физиков. Один из критиков, Питер Льюис, отметил, что из этой концепции следуют довольно странные выводы в отношении участия в авантюрах с риском для жизни. Например, если бы вам предложили посидеть в одном ящике вместе с котом Шрёдингера, вы бы, скорее всего, отказались. Однако из многоразумной модели следует, что вы ничем не рискуете: в те варианты реальности, где радиоактивный атом распался, и вы с котом были отравлены, сопутствующие вам «разумы» не попадут. Все они благополучно проследуют по той ветви, где вам суждено выжить. А значит, и риска для вас никакого нет.

Это рассуждение, кстати, тесно связано с идеей, так называемого квантового бессмертия. Когда вы умираете, это, естественно, происходит лишь в некоторых эвереттовских мирах. Всегда можно найти такую классическую проекцию, в которой на этот раз вы остаетесь в живых. Продолжая это рассуждение бесконечно, можно прийти к выводу, что такого момента, когда умрут все ваши «клоны» во всех мирах Мультиверса, никогда не наступит, а значит, хоть где-то, но вы будете жить вечно. Рассуждение логичное, но результат непредставим.

Теория скрытых параметров пытается объяснить результаты квантовых экспериментов неполнотой наших знаний о микромире. Вполне логичные в своей основе эйнштейновские идеи этой теории, тем не менее, не подтверждаются экспериментально, а проведенные проверки неравенства Белла напрямую опровергают существование скрытых параметров практически полностью.

Транзактная интерпретация квантов утверждает, что частица посылает запрос в будущее (волна предложение) и получает ответ из будущего (волна подтверждение). Степень совпадения фаз определяет амплитуду. Квадрат амплитуды определяет вероятность события. Такой подход решает проблему наблюдателя (парадокс кота Шрёдингера, и мыши Эйнштейна) и вылечивает солипсизм. Свобода воли при этом тоже исчезает.

Кроме того, многие физики склоняются к так называемой «никакой» интерпретации квантовой механики, ёмко выраженной в афоризме Дэвида Мермина: «Заткнись и считай!»

На данный момент квантовая механика является наиболее проверенной (и в то же время наиболее парадоксальной) теорией в истории науки. Каким же образом она работает, так никто до сих пор и не понял, несмотря на многолетний мозговой штурм.

Информатика против кибернетики. Взлет и крах советских ЭВМ

Последний вопрос, который мы затронем в этой книге, касается достаточно интересной и весьма деликатной темы. Что такое кибернетика, зачем она нужна, как она соотносится с информатикой, действительно ли вопиющее отставание Советского Союза в разработках ЭВМ связано со сталинскими гонениями на «реакционную буржуазную лженауку», как была кибернетика поименована в Философском словаре начала 50-х? Что такого великого сделал Норберт Винер, ее основатель? Правда ли что она положила начало исследованиям искусственного интеллекта, и если да, то где результат? Занимается ли сейчас хоть кто-нибудь этой наукой, и если да, то где они все? Почему компьютерный факультет МГУ называется «Факультет вычислительной математики и кибернетики», но учат там обыкновенному программированию и прочим ИТ-премудростям, а науки, давшей название всему факультету, даже нет в учебных планах, как, впрочем, нет ее и в планах всех университетов России и США? Куда и почему она пропала, если была так полезна, а если не была, то почему ее так превозносили, что называли факультеты в ее честь? И вообще, чем она занимается?

Это достаточно сложные вопросы, но поскольку они напрямую касаются многих вещей, связанных с разработкой компьютеров, надеюсь, читатель извлечет пользу из их освещения. Итак, для начала версия официальная, плавно перетекающая в мифологическую. Термин «кибернетика» происходит из греческого языка и дословно означает «искусство управления». Это наука, изучающая общие закономерности процессов управления и передачи информации в различных системах, будь то машины, живые организмы или общество.

Термин «кибернетика» изначально ввел в научный оборот Ампер, который в своем фундаментальном труде «Опыт о философии наук» (1834—1843) определил кибернетику как науку об управлении государством, которая должна обеспечить гражданам разнообразные блага. А в современном понимании — как наука об общих закономерностях процессов управления и передачи информации в машинах, живых организмах и обществе впервые был предложен американским математиком Норбертом Винером в 1948 г.

Базовыми концепциями кибернетики являются понятия системы, обратной связи, черного ящика, информации, самоорганизации. Теоретически в сферу ее интересов попадает процесс обработки информации любой сложной системой (электрической, биологической, социальной и т.п.), реакции на нее и выработки в ответ какого-либо поведения, а так же, как можно вмешаться в этот процесс с целью его улучшения и/или изучения и создать обучаемую и управляемую сложную систему искусственно (опять же, неважно какую — от искусственного интеллекта, до новой политико-экономической модели социума).

Методы кибернетики заключаются в выявлении системы, определении ее т.н. петли обратной связи⁷³, исследовании ее с целью понять принцип работы системы, не вникая в ее конструкцию⁷⁴ и модифицировать систему в желаемом направлении.

Кибернетика является междисциплинарным исследованием, объединяя области систем управления, теории электрических цепей, машиностроения, математического моделирования, математической логики, эволюционной биологии, неврологии, антропологии, социологии и экономики.

Современную форму кибернетике придал Норберт Винер — инженер, математик и философ. Среди его технических достижений основное место занимают исследования ПВО и конструирование усовершенствованных систем управления огнем зенитной артиллерии, выполненные по заказу Пентагона. В ходе исследований им была развита теория случайных процессов (т.н. винеровских процессов), стохастических дифференциальных уравнений, интегральных уравнений и рядов Фурье. Занимаясь конструированием систем наведения и разработкой математического аппарата, способного корректно описывать случайные процессы, Винер пришел к идее, что введенные им понятия подходят не только для теории автоматического управления, но также годятся для описания любых сложных систем и являются универсальными философскими законами. Итогом его изысканий в этом направлении стала программная книга «Кибернетика, или управление и связь в животном и машине» (1948 г. США, 1958 г. СССР), где он изложил основания соответствующей дисциплины.

Его философия заключалась в том, что принципы черного ящика, петли обратной связи и самоорганизации характерны для любых сложных систем — от компьютера до человека и социума, и общая модель организации и самоорганизации этих систем одинакова. Соответственно, занимаясь исследованиями в одной из областей, например, обучением компьютеров, можно экстраполировать результаты на все остальные, скажем, разработать оптимальные педагогические программы для людей и, наоборот, например, изучая мозг человека как черный ящик, научить думать машину. Такие взгляды тесно связаны с философией механицизма — представлениями, что человек, не более чем очень сложная машина, а также когнитивной психологией.

⁷³ поступление внешней информации от окружающей среды → обработка → принятие решения → реакция → изменение окружающей среды → поступление новой информации и т.д.

⁷⁴ идея «черного ящика» — понимать работу системы, не разбирая, и изучая, из чего она сделана, а подавая внешние сигналы и изучая реакцию на них.

Далее начинается миф, и вот его основные элементы:

- Профессор Винер был гениальным визионером, кибернетика является цельной наукой, сравнимой по важности с квантовой механикой и теорией относительности. Классический труд «Кибернетика, или управление и связь в животном и машине» является сравнимым по силе и глубине идей с работами Шеннона и Тьюринга.

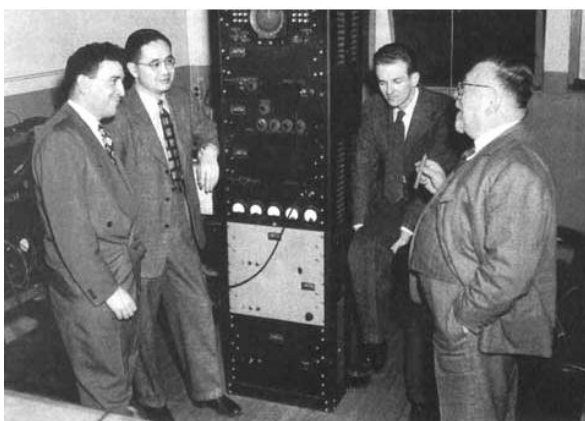
- Кибернетика является основоположницей всех Computer Sciences, которые без нее никогда бы не развились до нынешнего состояния. Она чрезвычайно важна для понимания работы и разработки ЭВМ.

- Всем, или многим современным успехам в области информатики США обязано работам Винера, в СССР же кибернетика много лет угнеталась по приказу Сталина и считалась буржуазной лженаукой, из-за этого наше неизбежное отставание в ИТ было предопределено глупым руководством страны. Поэтому сейчас в России и не делают компьютеры.

Рассмотрим эти мифы подробнее и начнем с примечательной личности основателя этой дисциплины. Вышедшая в 2005 году исследовательская книга знаменитого Ф. Конвея в соавторстве с Дж. Зигельманом «Темный герой информационной эпохи. В поисках Норберта Винера, отца кибернетики» («Dark Hero of the Information Age: in search of Norbert Wiener, the father of cybernetics» by Conway, F. and Siegelman, J., 2005), а также еще несколько менее крупных историко-научных работ последнего времени по крупицам восстанавливают детали престранной истории о ярком послевоенном рождении новой науки и о ее закате, последовавшем вскоре после неожиданной смерти отца-основателя в 1964 году. Норберт Винер (Norbert Wiener) – фигура в каком-то смысле типическая для второй половины XX века. Он – сын богатого белостокского еврея Лео Винера, приехавшего в Россию до революции из Австро-Венгрии. Сначала Лео учился в Варшаве на медика, но работа ему не понравилась, и он отправился в Берлин, где стал инженером, потом отправился в Англию, а затем окончательно переехал в США. После некоторых приключений Лео стал школьным учителем в Канзас-Сити, где женился на Берте Кан. Норберт – первый ребенок, родившийся через год после заключения брака. Впоследствии Лео Винер имел весьма интересную карьеру – в частности, добился положения профессора славянских языков в Гарварде.

Маленький Норберт был типичным ребёнком интеллектуалов, из которого хотели сделать гения. Родители были убеждены в его гениальности с самого начала и грузили науками, как могли, настраивая на успех, одновременно и травмируя – отправив его сразу в старший класс, и при этом, нагружая его домашним образованием, например, его домашним чтением в 7(!) лет были Дарвин и Данте. Винер в 11 лет окончил среднюю школу, а в 14 – колледж и получил первую в своей жизни ученую степень бакалавра искусств. Это не далось ему даром – он вырос фрустрированным невротиком, что отразилось на всей его жизни, включая колоссальные проблемы с памятью и вниманием. Кроме этого, у него были и чисто физические трудности, заметные даже постороннему глазу – он был феноменально неуклюж, страдал чрезвычайной полнотой и сильнейшей близорукостью. Впоследствии Винер неоднократно обращался к услугам психоаналитиков, но большого эффекта это не дало, зато в четырнадцать лет поступил в Гарвард, где защитил докторскую диссертацию в 18 лет. С детства он был подготовлен к блестящей академической карьере и одновременно, по его собственному признанию, владел «коллекцией клинических неврозов и душевных терзаний».

Дальше талантливого юношу родители и друзья стали продвигать в науку. Когда у него появился интерес к философии математики, он поехал в Кембридж к великим Бертрану Расселу и Харди, а потом брал курсы математики у лучших немецких профессоров, включая Гильберта. В 1914 г. грянула Первая мировая война, и он спешно вернулся домой. В Америке Норберт добровольно отправился на призывной пункт, но не прошел медкомиссию из-за сильной близорукости. Винер пытался преподавать в Университете штата Мэн, писал статьи для энциклопедии, работал помощником инженера, занимался журналистикой. Однако все его начинания неизменно кончались полным крахом. Так прошло пять лет, пока в 1919 г. отец, тогда уже профессор Гарварда, смог продвинуть его на должность преподавателя кафедры математики Массачусетского технологического института, где он и работал всю жизнь.



Норберт Винер

Вскоре стало ясно, почему Винер-младший до этого нигде долго не задерживался – об уровне преподавания профессора Винера лучше всего скажут его ученики. Китайский физик К. Джен (С. К. Jen), обучавшийся в Массачусетском технологическом, пишет: «Вспоминая жизнь в МТИ, невозможно не рассказать о замечательном человеке, Норберте Винере, свидетелем эксцентричности которого мне довелось быть. Я помню, что профессор Винер всегда приходил в аудиторию без конспекта лекции. Сначала он доставал большой носовой платок и прочищал нос очень энергично и шумно. Он почти не обращал внимания на аудиторию и редко объявлял тему лекции. Он поворачивался лицом к доске, стоя очень близко к ней из-за своей очень сильной близорукости. Хотя я обычно сидел на первом ряду, мне было трудно разобрать, что он пишет. Большинство других студентов не могли видеть вообще ничего. Но наибольшее удовольствие для аудитории было слышать, как профессор Винер говорит сам себе: «Ну, это, определенно, совершенно неверно». При этом он быстро стирал все, что было написано. Затем он начинал все сначала, бормоча про себя: «Пока это, похоже, правильно». И через минуту: «Однако это не может быть правильно», – и стирал все опять. Этот процесс повторялся вновь и вновь, пока не звенел звонок с лекции. Профессор Винер уходил из аудитории, даже не взглянув на своих слушателей».

Роберт К. Везерол (Robert K. Weatherall), директор службы по трудоустройству выпускников, передает рассказ одного из студентов, который «по дороге в Нью-Хемпшир остановился, чтоб помочь человеку, беспомощно стоящему около машины с проколотой шиной», в котором он узнал Норберта Винера. Винер проверил у него зачетку и сказал, что может принять от него помощь, так как зачет уже получен.

Кроме «таланта» лектора и педагога, Винер Норберт был наделен своеобразной памятью. Другой сотрудник МТИ, администратор факультета математики Филлис Блок (Phyllis L. Block), вспоминает: «Он часто навещал меня в офисе и разговаривал со мной. Когда спустя несколько лет мой офис переехал в другое помещение, Винер пришел ко мне представиться и познакомиться. Он не помнил, что я тот же самый человек, с которым он часто общался. Я был в другом помещении, и он принимал меня за кого-то другого, – смеялся Блок – Меня он помнил только по комнате, в которой я сидел...»

Однажды семья этого чудака, а он был женат и имел двух дочек, переехала. Жена Норберта Маргарет дала ему листок с новым адресом, чтобы муж нашел дорогу домой. Однако Винер в первый же день исписал его чистую сторону формулами и вычислениями. Затем, увидев в своих расчетах ошибку, он смял и выкинул бумажку с новым адресом. Вечером он, как ни в чем не бывало, пришел по старому адресу. Увидев, что его дом пуст, Норберт Винер минут 15 стоял с несчастным видом. Затем, с большим трудом преодолев робость и запинаясь, он обратился к стоящей рядом девочке: «Извините, возможно, вы помните меня, я профессор Винер. Моя семья недавно переехала отсюда. Вы не могли бы мне сказать, куда именно?» Девочка вздохнула и ответила: «Да, папа, мама так и знала, что ты это забудешь».

Причем сам Норберт Винер о своей памяти был высокого мнения: «То, что я действительно унаследовал от отца, – великолепная память. Под этим я имею в виду, что если нас осенил ряд идей или нам открылся какой-то новый взгляд на вещи, то это становится неотъемлемой частью нас самих, и, несмотря ни на какие превратности судьбы, мы не способны утратить это».

Непосредственной причиной запуска проекта «кибернетика» послужила важная военная задача: создание системы ведения зенитного огня на основании информации, получаемой от радиолокационных станций. Поставленная цель была простой – полностью заменить наводчика, что требовало предсказания поведения самолетов противника и создание автоматизированной системы управления огнем. Винер, имевший неплохое медицинское и физиологическое образование, заметил сходство между рефлексами живых организмов и движениями стволов батареи, впрочем, на эту мысль навёл его друг, мексиканский физиолог Артур Розенблют, один из основателей нейрокомпьютинга. Впоследствии Винер посвятит ему «Кибернетику».

События, вызвавшие к жизни кибернетику, были не столь забавны, как личность ее основателя, и реши бы какой-нибудь любопытный сунуть нос в суть работ доктора Винера в те годы – свести бы ему тесное знакомство с Федеральным Бюро Расследований, выполняющим в США функции контрразведки. Объяснялось это тем, что кибернетика начала свое развитие с события, которое американцы называют Day of Infamy – День бесчестия 07.12.41, атака японского ударного соединения на Пирл-Харбор, военно-морскую базу США.

Торпедоносцы и пикирующие бомбардировщики, поднятые со скрытно подошедших к Пирл-Харбору авианосцев, лишили американский Тихоокеанский флот его ядра: четыре линкора было потоплено, а четыре – тяжело повреждено. Это позволило Японии взять инициативу в свои руки и развернуть массированное наступление по всему Тихоокеанскому театру военных действий. Атакам японской авиации подвергался даже Датч-Харбор, расположенный на далеких Алеутских островах.

От американских военных инженеров потребовалось резко повысить эффективность зенитного вооружения линейных кораблей. Сделать это классическим путем – увеличить количество орудий противовоздушной обороны и их калибры – было невозможно, поскольку, как ни велик линкор или тяжелый крейсер, свободного места на нем немного. Выход был найден в повышении эффективности систем управления зенитным огнем. По современным понятиям эти системы были очень просты, но тогда они казались верхом совершенства, и их создание потребовало разработки нового теоретического аппарата.

Радиолокаторы поста ПВО предупреждали о появлении воздушных целей. Радиолокаторные прицелы наводили на самолеты орудия. Сложнейшие релейные коммутаторы аналоговых вычислителей распределяли целеуказания между полуторайдомовыми зенитными автоматами. Синхромеханизмы устанавливали на

нужную дальность дистанционные взрыватели снарядов автоматических пушек, и все это без участия человека! По тем временам это было магией.

Определение точек, куда нужно навести ствол орудия для того, чтобы выпущенный из него снаряд встретился с атакующим самолетом, производился с помощью центрального бортового компьютера корабля. Группа доктора Винера занималась как раз разработкой алгоритмов работы таких вычислителей (алгоритмы тогда реализовывались путем коммутации операционных усилителей, емкостей, сопротивлений и кое-каких нелинейных элементов – машины были аналоговые). Почему привлекли именно его? Суть в том, что Норберт Винер был действительно выдающимся в своей области математиком – специалистом в теории случайных процессов и зарождающейся теории автоматического управления.

С 1930 по 1940 Винер издает ряд книг математике: «Обобщенный гармонический анализ», «Тауберовы теоремы», «Интеграл Фурье и некоторые его применения» и др. Совместное исследование с немецким математиком Э. Хопфом о радиационном равновесии звезд вводит в науку «уравнение Винера–Хопфа. Другая совместная работа, монография «Преобразование Фурье в комплексной области» написана в сотрудничестве с английским математиком Р. Пэли. В компании с китайским ученым Ю.В. Ли и знаменитым Ванневаром Бушем, известным конструктором аналоговых вычислительных машин, Винер занимается исследованиями АВМ. В 1935–1936 гг. Винер был вице-президентом Американского математического общества. Но вернемся к кораблям.

Традиционная схема управления зенитками тогда выглядела следующим образом. На основе наблюдения за параметрами полета самолета производился расчет его прогнозируемой траектории (экстраполяция кривой по данным замеров) и выполнялся выстрел в точку, где должен находиться объект в данный момент (будем пока считать, что проблему попадания в нужную точку и установки дистанционного взрывателя артиллеристы умеют решать отлично). Решение такой задачи требует некоторого аппарата расчетов и управления. На кораблях в начале XX века применяли специальные приборы для замеров дальности и скорости движения и соответствующий математический аппарат с использованием артиллерийских таблиц.

Проблема с самолетами заключалась в том, что они летали в трехмерном пространстве и существенно быстрее, чем раньше: потребовались новые методы наблюдения за ними (радиолокаторы), более сложные алгоритмы расчетов, с одновременным повышением скорости их выполнения, а также автоматизация процедуры наводки орудий, не говоря уже о проблеме управления дистанционными взрывателями для поражения самолета осколками. Для повышения эффективности всей этой процедуры необходим переход от детерминированных математических моделей к стохастическим, с чем Винер блестяще справился, разработав новую теоретическую модель ПВО.

Суть проблемы заключается в том, что на самом деле предсказать траекторию полета самолета можно лишь с определенной степенью вероятности. На это влияют самые разнообразные погрешности, например, данные наблюдений и математического метода экстраполяции, не говоря уже о том, что летчик может выполнить противозенитный маневр. Таким образом, поведение самолета описывается функцией $P(x, y, z, t)$ — вероятности того, что он будет находиться в данной точке, в данный момент времени. Раньше все стреляли в точку, где эта вероятность была максимальной, а теперь ставилась другая задача — разные орудия должны стрелять в разные точки пространства таким образом, чтобы зоны их поражения не пересекались, но при этом суммарная вероятность накрытия всех целей была максимальной.

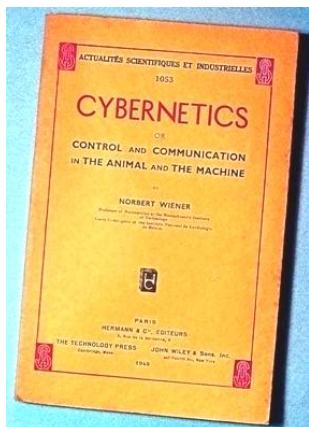
Такая постановка задачи сразу позволяет оптимизировать и использование ресурсов (орудий) — расчеты смогут сразу показать, что для практически стопроцентного поражения цели достаточно и пяти зениток. Соответственно менялась и задача при защите от атаки нескольких самолетов. Если раньше каждое орудие (или их группа) отрабатывали отдельную цель, то теперь методом суперпозиции определялась общая функция всех самолетов $P(x, y, z, t)$, и стрельба велась по этой виртуальной цели. Сложность решения такой математической задачи является совершенно очевидной, тем более, если учесть, что такими же стохастическими функциями характеризуются все элементы системы ПВО. Кроме того, нужно учитывать параметры действенности (поражения) каждого конкретного орудия, степень опасности разных целей и пр.

Винер применил для решения задачи открытый им класс стохастических дифференциальных уравнений, введя понятие случайного процесса и разработав методы оптимизации, позволяющие с максимальной вероятностью навестись на максимальное количество целей, используя механизм обратной связи «орудие-радиолокатор», без участия оператора. Отныне вычислители орудий могли сами анализировать и предсказывать поведение самолетов врага.

Задача повышения эффективности корабельной ПВО с помощью новейших теоретических методов была решена блестяще. Свидетельством этого стал тот факт, что из 33 японских самолетов, решившихся в 1943 году атаковать американский линкор South Dakota, 32 были уничтожены в воздухе еще на подлете шквалом огня, выпущенного зенитными орудиями корабля. Результаты Винера были обобщены в его секретном отчете 1942 г. «Интерполяция, экстраполяция и сглаживание стационарных временных рядов» (рассекречен и издан в 1949 г.).

Пока отчет был не рассекречен, Винер не мог легально поделиться своими математическими методами с мировым сообществом. Тем не менее, его жажда славы превозмогла опасность последствий раскрытия тонкостей механизма обратной связи и управления. В 1946 г., находясь в Париже на конференции по математи-

ке, Винер встретился с издателем, который предложил ему написать книгу об обратной связи между автоматами и нервной системой человека. Винер не смог избежать искушения. Он уезжает в Мексику, где ему никто не мешает, в гости к уже упоминавшемуся физиологу Розенблюту и в течение года пишет первое издание «Кибернетики». Параллельно из области чистой математики его уносит в философию и физиологию, первая часть книги посвящена непосредственно теории управления, но уже во второй он начинает активно продвигать концепции универсальности и всеприменимости открытого им знания, и дает новой науке название – кибернетика.



Книга вышла из печати с огромными погрешностями, в основном потому, что Винер в то время страдал катарактой и не мог держать корректуру, а на ассистентов полагаться было нельзя. На специалистов в области вычислительной техники она произвела, скорее, отрицательное впечатление. В ней они увидели, в основном, беспочвенные фантазии и оторванное от реалий тогдашней науки и техники философствование, необоснованные претензии на всеобщность методов частных наук. Но неожиданно эта книга заинтересовала ученых, к вычислительной технике отношения не имеющих и поэтому склонных весьма преувеличивать ее возможности. Именно они и сгруппировались впоследствии в «Американское Кибернетическое Общество» и выступили самыми горячими защитниками кибернетики.

Справедливо опасаясь последствий разглашения военных тайн, Винер делает интересный шаг – выступает в качестве яростного антимилитариста. Еще в январе 1947 года американский журнал *Atlantic Monthly* опубликовал его открытое письмо под заголовком «Ученый восстает!». В этом письме математик публично и в довольно резких выражениях отвечает инженеру корпорации «Боинг», занимающемуся созданием управляемых ракет. Винер отказывается предоставить ему информацию об одной из своих работ военного времени и попутно делает весьма жесткие заявления, призывая присоединиться к нему других ученых. В частности, Винер клянется не публиковать в будущем никаких работ, которые могли бы нанести людям вред, оказавшись «в руках безответственных милитаристов».

Это письмо стало лишь первым актом в целой серии энергичных политических воззваний и демаршей, предпринятых Винером в конце 1940-х и в 1950-е годы. На уровне идеологической деятельности Винер не устанно критиковал все те новые организации, что создавались государством и индустрией для научной работы в области — как он ее называл — «науки мегабаксов» (*Megabuck Science*). Конец был предсказуем — сразу же после публикации письма на него было заведено дело в ФБР, которое велось и регулярно пополнялось все новыми свидетельствами «симпатий к коммунистам» и «подрывной деятельности» математика на протяжении последующих 17 лет — вплоть до его смерти в марте 1964-го. В ЦРУ параллельно предпринимались специальные шаги, чтобы оценить масштабы потенциальных угроз со стороны теории автоматического управления, в том случае, если она получит развитие и широкое распространение в СССР.

Служба американских спецслужб за Винером была настолько плотной, что, когда он отправлялся в Европу для лекций и встреч с учеными, его по эстафете передавали коллегам из британской контрразведки MI5. В итоге всей этой напряженной активности психическое здоровье Винера, и так не слишком крепкое, пошатнулось окончательно. Он стал еще больше страдать рассеянностью, мысли его занимала уже не математика, а отношения с властями. Большая часть его знаменитых «лекций», когда он мог выйти к доске, никого не замечая простоять так все занятие, иногда что-то бормоча под нос, а затем удалиться, относится именно к этому периоду. Он посетил СССР, общался со знаменитым А.Н. Колмогоровым, активно продолжал продвигать кибернетику, проводя опыты с нейрофизиологией мышей и кошек, правда, особого успеха не добился. Начал писать фантастические рассказы и повести в кибернетическом духе, в одном из них, «Голова», нейрохирург, у которого мафия убила жену, во время операции на мозге главарию оной мафии, делает ему лоботомию, чтобы тот больше не смог лелеять преступные планы. Издает «Новые главы кибернетики», где окончательно отходит от науки, в частности, ссылаясь как на достоверный источник информации о поведении на мультипликацию Уолта Диснея.

Кибернетика не пользовалась в США особым успехом — поначалу привлеченные необычными идеями, ученые один за другим покидали Винера, например, как уже упоминалось, со скандалом покинул Американское Кибернетическое Общество знаменитый Мак-Каллох, теоретик нейросетей и его последователи. По поводу книги «Кибернетика» лучше всего высказался Кендалл: «Кибернетика» — отвратительно организованный текст. Это собрание опечаток, ошибочных математических формул, продуктивных, но бессвязных идей и логических нелепостей. Очень жаль, что именно эта работа сделала Винеру большую часть его общественной славы ... в то время читатели были в большей степени очарованы богатством идей этой книги, нежели её недостатками».

На самом деле никакого особенного «богатства идей» в книге не наблюдается. Идея, которая захватила публику и сделала Винера великим — там была ровно одна: законы, повелевающие интеллектом, так же просты, как законы природы, и вскоре мы создадим автоматы, которые будут мыслить не хуже людей. Всё это якобы доказывалось неудобопонятными формулами, к тому же относящимися на самом деле к совсем другим вещам — теории автоматического управления. Публика это охотно проглотила — в те времена все жда-

ли от науки чудес, в том числе и создания «искусственных мыслящих существ». Впоследствии престарелый философ и футуролог Станислав Лем с сожалением признавался: «меня убедили, что в самом скором времени будет создан искусственный интеллект, и это исказило все мои прогнозы».

Помимо формул, книга содержала многостраничные размышления, например, о том, какое лечение шизофрении предпочтительнее: лоботомия или электрошок? Оказывается, всё-таки, электрошок. Описание опытов в институте сердца над бедными кошками, из которых делали «спинномозговой препарат», разрезая сухожилия. Ладно бы, если бы из этих опытов был получен какой-то познавательный результат. Но всё, что из своих опытов над бесчисленными невинными созданиями Винер смог получить, это то, что частота нервных импульсов не зависит от психологического состояния подопытного «спинномозгового препарата» и определяется, по-видимому, (он даже в этом не уверен), только собственными постоянными времени нервной системы. Много ума, чтобы воткнуть в нерв щуп осцилографа, не надо, тем более, когда до тебя уже это делали.

Отдельная тема – Винер и компьютеры. Вопреки распространённому мнению, что Винер имеет какое-то отношение к этим вопросам, он не опубликовал ни одной значительной работы, связанной с вычислительной математикой и тем более вычислительной техникой. Не написавший за всю свою жизнь ни единой программы, Норберт Винер обожал раздавать многозначительные интервью на тему перспектив развития вычислительной техники – разумеется, туманные.

В 1964 г. ФБР, наконец, отстало от измученного Винера, поняв, что никаких военных секретов он больше не разгласит. Вся страна была озабочена другими вещами – кубинский кризис, затем убийство Кеннеди и неожиданное возвышение его вице-президента Линдона Джонсона. Новый президент, отвлекая общественность от странной гибели своего предшественника, развернул бурную деятельность и вскоре после инаугурации пожаловал Винеру «Золотую медаль ученого», высшую награду для человека науки в Америке. На торжественной церемонии награждения президент восхвалял заслуги Винера в самых лестных выражениях: «Ваш вклад в науку на удивление универсален, ваш взгляд всегда был абсолютно оригинальным, вы потрясающее воплощение симбиоза чистого математика и прикладного ученого». Но Винер и тут не изменил себе. Во время речи президента ученый достал платок и по-традиции мощно прочистил нос, а после спрашивал у соседей, что сказал этот джентльмен. Через два месяца, в марте 1964-го в Стокгольме, когда Винер поднимался по лестнице Шведской академии наук, чтобы сделать доклад для местной научной элиты, он неожиданно упал и больше не поднялся. Предположительная причина смерти по результатам вскрытия — легочная эмболия. Ему было 69 лет. Еще через несколько месяцев, летом 1964-го, США начали войну во Вьетнаме, и всем стало уже не до кибернетики.

Винер был близко знаком практически со всеми великими основателями информатики – от Тьюринга до фон Неймана, тем не менее, его вклад в эту науку минимален. Т.н. кибернетика второго уровня испытала определенный подъем в 70-е годы, в основном за счет философских рассуждений об идеальных механизмах управления и человеческой природе, родственной машинам. Затем она и уже окончательно снова развалилась на десятки специализированных дисциплин, какими, в сущности, и являлась – нейрофизиологию, социологию, когнитивную психологию, менеджмент, теорию автоматического управления и т.п. Иногда их по традиции называют биологической, социальной, математической и прочими кибернетиками, но ясно, что цельной науки из этой дисциплины не вышло – науку не построишь на одном основном постулате и двух-трех понятиях, известных задолго до нее.

Никаких факультетов или гигантских институтов кибернетики в США так и не возникло. Есть люди, занимающиеся перечисленными областями, но их совсем не так много и они тонким слоем распределены по инженерным, компьютерным или математическим факультетам. Если вы, например, захотите изучать теории управления или обработки сигналов в каком-нибудь приличном американском университете, то вам нужно будет взять на факультете Electrical Engineering курсы «control engineering» и «signal processing», при этом никаких работ Винера там в программе не будет. Слово «cybernetics» есть в названиях нескольких научных журналов, но журналов маргинальных и малопрестижных. Книга Винера и сам термин кибернетика куда менее популярен в США, чем был у нас. Есть сейчас слова киберпространство, кибертерроризм, киберкафе, даже киберпанки, и если спросить среднего американца про кибернетику, то он ответит, что это что-то про Интернет и виртуальную реальность. Но ведь все это никакого отношения к идеям Винера не имеет.

Посмотрим, какие же плоды дала эта наука. Самые важные достижения научно-технического прогресса в последние несколько десятилетий XX века случились в области компьютеризации, автоматизации, телекоммуникаций, информационных технологий и т.п. Именно здесь произошли самые важные прорывы. Дешевые персональные компьютеры, встроенные микропроцессоры на каждом шагу, Интернет, мобильные телефоны и т.п. – появление всего этого имело огромные экономические, социальные, военные, пропагандистские последствия. И именно в этом, опять же, СССР проиграл Западу вчистую. При этом на предыдущем этапе СССР вполне смог выдержать конкуренцию в области принципиально важных технологий 50-х 60-х годов – ядерное оружие, атомная энергетика, космос, ракеты. Замечательные достижения в этих областях на 99% были обусловлены развитием электроники, совершенствованием элементной базы, вещами сугубо технологическими, вклад же книги Винера в научно-технический прогресс был нулевым.



Первое издание «Кибернетики»
в СССР, 1948 г.

Несмотря на то, что он мог бы продолжить свою математическую карьеру, как ученый Винер закончился после разработки систем ПВО. Он больше не написал ни одной статьи ни по математике, ни по вычислительной технике и все, чем он занимался, носило исключительно философский⁷⁵ и, временами, даже мистический характер. Возможно, его поразила болезнь, которую некоторые называют синдромом Льва Толстого (амбициозный и честолюбивый классик русской литературы под старость захотел еще и славы великого философа и основателя новой религии). Эта болезнь часто поражает стареющих маститых профессоров, когда они, не довольствуясь уже заработанными в своих областях почестями, начинают заниматься какими-то привлекающими куда большее общественное внимание областями. Обычно ничего дельного из этого не получается.

Что же случилось с кибернетикой в СССР? Ее развитие было прямо противоположным тому, что обычно представляют. Во время «гонений» на нее информатика развивалась отлично. По иронии судьбы, с прекращением оных «гонений» развитие ВТ стало хиреть, пока не прекратилось совсем, при этом кибернетических НИИ и факультетов было на порядок больше, чем в США, только они либо занимались совсем не кибернетикой, как ФВМК МГУ, либо были совершенно бесполезны.

Давайте рассмотрим хронологию «гонений» и, заодно, развития отечественной ВТ до 60-х годов:

Год	Кибернетика	Информатика
1948	В СССР издается книга Винера. Так как она касается в т.ч. военных проблем, то автоматически попадает в спецхран СКБ-245 оборонного Министерства радиопромышленности СССР.	Под руководством Л. И. Гутенмахера создана первая электронная АВМ для решения ОДУ Создан Институт точной механики и вычислительной техники АН СССР – основной центр разработки перспективных ИТ.
1950	В начале мая 1950 г. в «Литературной газете» появилась статья известного публициста Бориса Агапова «Марк III, калькулятор» содержащая резкую критику идеи использования электронных вычислительных машин для обработки экономической информации, а также нелицеприятную оценку личности Винера.	Пробный пуск первой в континентальной Европе вычислительной машины МЭСМ, третьей в мире, после английского Collosus и американского ENIAC.
1951	В 1951 году майор А.И. Китов, научный референт Академии артиллерийских наук находит в спецхране и читает книгу Винера. Он впечатлен и знакомит с ее идеями многих известных математиков и инженеров, в т.ч. Ляпунова.	Завершены испытания и начата регулярная эксплуатация МЭСМ
1952	Статья М. Ярошевского «Кибернетика — «наука» мракобесов» в «Литературной газете». Статья К. Гладкова «Кибернетика или тоска по механическим солдатам» в журнале «Техника — молодёжи» Статья Б. Э. Быховского «Кибернетика – американская лженаука» в журнале «Природа». Статья Ю. Клеманова «Кибернетика» мозга» в журнале « Медицинский работник». Пик «гонений» на кибернетику.	Начата опытная эксплуатация ЭВМ М-1, завершена разработка и начата опытная эксплуатация БЭСМ-1, на тот момент — самой быстродействующей ЭВМ в Европе. Начаты работы по проектированию машины М-2. В МГУ появляется тогда еще кафедра вычислительной математики, где на самом деле изучали ЭВМ.
1953	Сталин умирает. Статья Б. Э. Быховского «Наука современных рабовладельцев» в журнале «Наука и жизнь». Выходит 5-й номер журнала «Вопросы философии» со статьёй «Кому служит кибернетика», подписанной псевдонимом «Материалист». С середины 1953 г. в течение двух лет (!) А. И. Китов и А. А. Ляпунов выступают с лекциями о кибернетике в Центральном лектории Политехнического музея, в МГУ, в ЦК КПСС (!), во Всесоюзном Доме науки и техники, в ведущих НИИ Москвы, а А. И. Китов ещё и Ленинграда.	Начата эксплуатация ЭВМ М-2. Начат серийный выпуск ЭВМ «Стрела», продолжался до 1956 года. На кафедре ВМ МГУ Ляпунов читает первые лекции по алгоритмизации, вычислительной сложности и теории программирования.

⁷⁵ В рамках книги по архитектуре ЭВМ вполне простительно не считать религиозно-философские взгляды наукой.

1954	Выходит 4-е дополненное и исправленное издание «Краткого философского словаря» со знаменитой статьёй «Кибернетика – реакционная буржуазная лженаука...» Во второй половине 1954 г. на семинаре по общеметодологическим проблемам естествознания в НИИ-5 Главного артиллерийского управления МО СССР, по инициативе участников семинара в программу был включён ряд сообщений о рациональных зёрнах кибернетики. Сообщения были поручены трём кандидатам наук, включая известного А. И. Китова. Выступления встретили большой интерес.	Начата разработка ЭВМ «Урал-1» Китов организовал и возглавил ВЦ-1 Министерства обороны – первый в СССР вычислительный центр
1955	Подписан в печать дополнительный тираж 4-го издания «Краткого философского словаря», в котором отсутствует критическая статья о кибернетике. В журнале «Вопросы философии» выходит статья в защиту кибернетики С. Л. Соболева, А. И. Китова и А. А. Ляпунова «Основные черты кибернетики». На Мехмате МГУ проходит выступление, посвященное кибернетике. Докладывают те же Китов и Ляпунов, зал аплодирует, противники кибернетики от выступления отказались. Статья «Техническая кибернетика» печатается в журнале «Радио». «Гонения» закончились.	
1958	Переиздание Винера большим тиражом, книга помещена в общий читальный зал Ленинской библиотеки.	
1959		Создается троичная ЭВМ «Сетунь»
1962	В.М. Глушков становится директором организованного им же при активной поддержке знаменитого инженера-адмирала-академика Берга Института кибернетики АН УССР. Лично президент академии М. В. Келдыш был категорическим противником создания нового института в то время, когда уже активно работали Институт математики и точной механики, Математический институт им. Стеклова и др.	
1964	Хрущёв снят со всех постов и принудительно отправлен на пенсию.	Создается ЭВМ МИР-1
1970	По всей стране работают сотни НИИ кибернетики, открыты факультеты, печатаются журналы, защищаются диссертации, правда, в основном по философии. Кафедра ВМ МГУ преобразована в факультет вычислительной математики и кибернетики.	Принято решение копировать серию IBM s/360 История отечественной ВТ завершилась

В результате мы можем сделать несколько важных выводов. Во-первых, «гонения» на кибернетику заключались всего лишь, в нескольких статьях, написанных журналистами и философами и опубликованными в изданиях соответствующего качества – литературных и философских.

Однако, в последнее время обнаружили достаточно интересные факты, касательно двух анонимов – автора разгромной статьи в Словаре и того самого «Материалиста». Это как раз были техники, осознающие бесполезность этой науки. Анонимная статья в словаре была написана Е. А. Шкабарой, заместительницей главного конструктора первой советской вычислительной машины самого С. А. Лебедева. Она предложила подписать ее своему шефу и дала ей ход. Что касается знаменитой разгромной статьи «Кому служит кибернетика», то и там не все просто. Статья содержит:

- описание огромного значения, которое имеет применение вычислительных машин для науки, техники и промышленности;
- сетования (со ссылками на американских ученых) на то, что в США ЭВМ используются в основном, военным ведомством;
- критику претензий кибернетиков на то, что машина представляет собой аналог человеческого мозга и что разница между мозгом и ЭВМ исключительно количественная.

Эта часть была самой крупной и занимала полстатьи. Самое интересное, что критика эта производилась не с позиций философии, а исключительно с позиций учения Павлова о нервной системе. На Павлова в статье имеется 13 ссылок, в то время как из философов там упоминается только Лейбниц, которому почему-то приписали изобретение первого арифмометра, да Ламетри, как представитель механицизма. Один раз упоминается Маркс, но не в связи с разбираемым вопросом.

Характерно, что автор статьи не подписался собственным именем, а скрылся под псевдонимом «Материалист». Один из исследователей, Д. А. Поспелов объясняет это тем, что автор статьи «по-видимому, чувствовал некоторый страх». Но это объяснение лишено всякой логики. Если это был один из, как выразился сам Д.А. Поспелов, «придворных философов», то кого же он боялся? Неужели сторонники кибернетики тогда были страшнее, чем власть? Есть предположение, что причина того, что автор выступил под псевдонимом, была другая. Она состояла в том, что автор (или авторы) не хотел портить отношения с коллегами. Другими словами, что это был не философ. Даже самый поверхностный сравнительный анализ статей «Кому служит кибернетика» из «Вопросов философии» и статьи «Кибернетика» из философского словаря показывает, что это очень родственные тексты, статья из словаря фактически просто сжато пересказывает журнальную. Так что вполне возможно, что они принадлежат перу одного и того же автора.

Далее еще интересней. Была организована серия выступлений в академических институтах и вузах с целью пропаганды кибернетики и сохранилась стенограмма обсуждения одного из таких докладов, который был зачитан 24 июня 1954 года на методологическом семинаре в Энергетическом институте. В это время именно ЭИ был ведущей организацией в деле конструирования ЭВМ. Очень характерно, что оппонировали Ляпунову или отнесли к его докладу настороженно именно специалисты этого института. Что касается приглашенного из института философии Н. Ф. Овчинникова, то он оказался фанатиком и едва ли не большим сторонником кибернетики, чем сам Ляпунов. Надо сказать, что Ляпунов в то время еще был весьма осторожен и делал весьма разумные оговорки, когда речь заходила о весьма специфических философских вопросах, таких как вопрос о природе общества и мышления. Это чуть позже, когда победа кибернетиков станет очевидной, сторонники кибернетики разойдутся окончательно.

Неправы были в споре и те, и другие. Они лишь метались между двумя борющимися партиями технарей, ни одна из которых не понимала и в принципе не могла ничего понимать в предмете спора. Ведь сущность его была вовсе не техническая. Никто из спорящих не отрицал, что вычислительную технику нужно развивать. Спорили о природе мышления - о том, можно ли средствами математики смоделировать человеческое сознание. Для того чтобы решить этот вопрос, нужно было как минимум понимать, что такое человек и что такое мышление, а для этого недостаточно быть философом, для этого нужно быть еще и нейрофизиологом, притом, что эта тема до конца не развита и сейчас.

Во-вторых, половина из них вышла уже после смерти Сталина, так что он к ним никакого отношения не имел. В-третьих, книга Винера была убрана в спецхран вовсе не из-за мифических «гонений», а из-за того, что потенциально содержала информацию военного характера, которую у нас и в лучшие годы прятали под замок. При этом все, кто имел допуск, ее могли найти и прочитать, а после этого – обсудить с теми, кто оно допуска не имел, как Китов с Ляпуновым. Как только разобрались, что книга ничего важного собой не представляет – ее отдали в Ленинскую библиотеку.

Далее, литературные «гонения» отнюдь не помешали в самый их разгар выступать с публичными лекциями о кибернетике от НИИ до ЦК КПСС, причем лекции пользовались успехом, и никто их не разгонял. Ну и наконец, проследив развитие информатики, легко заметить, что с кибернетикой там корреляция нулевая – достижения и ее провалы от достижений и провалов кибернетики не зависели.

В 1958 г. книга Винера была переиздана у нас и сразу стала необыкновенно популярна. В конце 50-х - начале 60-х годов в СССР был настоящий кибернетический бум. Словно грибы после дождя, появлялись научно-исследовательские институты, факультеты ВУЗов, лаборатории, кафедры, отделы, научные журналы, ученые советы, в названиях которых стояло супермодное тогда слово «кибернетика» в самых разных сочетаниях. Была кибернетика техническая, математическая, теоретическая, технологическая, экономическая, химическая и даже юридическая! Сотни тысяч научных сотрудников и преподавателей были вовлечены в эту сферу. Чем же занималась эта гигантская армия?

Границы новой науки были определены весьма нечетко. Если читать советскую литературу тех лет, то ставилась задача создания новой фундаментальной науки, столь же важной и универсальной, как физика. Предполагалось, что эта наука сформулирует универсальные законы анализа и синтеза сложных систем самой разной физической природы и будет играть самую важную, определяющую роль в научно-техническом прогрессе. Часто звучала идея, что в новой науке концепция информации будет столь же важна, как концепция энергии в физике. На деле же, в основном, изучались несколько специфических областей, стоящих на границе технических наук и математики: математическая теория управления и обработки сигналов, теория информации. Часто туда включались элементы теории автоматов и алгоритмов (на Западе обычно называется «theoretical computer science»), распознавания образов, иногда даже разделы вычислительных методов или исследования операций.

Иными словами, кибернетика пышно расцвела именно в хрущевском СССР, в США же к этому времени, кибернетический бум давно прекратился.

Народному хозяйству требовалось много программистов, и не теоретиков, а прикладников, решающих типовые задачи. Бесчисленные же «кибернетические» факультеты очень мало учили студентов собственно программированию, а учили всему, чему угодно – от математики, до философии кибернетики. Вообще, советский кибернетический бум создал громадную армию прикладных математиков, полезность которых для реальных приложений была весьма сомнительной, чистые же математики, считали их не прикладными, а просто плохими.

Короче, проблема с кибернетикой в СССР была не в том, что она изучала какие-то лженаучные вещи. Нет, все это вполне разумные и легитимные области для научных исследований. Проблема в том, что они не столь важны и перспективны, как это представляли лидеры этой области в СССР. Как фундаментальное, так и прикладное значение этих областей довольно скромно, во всяком случае, совершенно недостаточно для создания громадных институтов и факультетов. Советскую кибернетику можно рассматривать, как непредумышленно порожденную, а затем самоподдерживающуюся гигантскую аферу советской научно-технической интеллигенции: вещам третьестепенным был придан статус первостепенных, и термин «кибернетика» оказался поразительно удачным для получения государственных средств.

Спустя 50 лет, можно уверенно сказать, что предложенная Винером и подхваченная в СССР программа построения новой фундаментальной науки полностью провалилась. Кибернетика оказалась еще одной шестидесятилетней иллюзией. Кибернетический миф очень здорово подошел атмосфере хрущевской оттепели («Девять дней одного года», «И на Марсе будут яблони цвести», фантастические повести о коммунизме на Венере братьев Стругацких – «Страна багровых туч», «Путь на Амальтею», «Стажеры»), нарождающейся субкультуре советского НИИ. Кибернетика действительно оказалась лженаукой, но отнюдь не буржуазной, а сугубо советской.

Финальным шагом, когда советская микроэлектроника уже начала разваливаться, стал XXIV съезде КПСС (1971).

Виктор Михайлович Глушков — ключевая фигура отечественной кибернетики. Герой Социалистического Труда, лауреат Ленинской и двух Государственных премий СССР. Главным материальным результатом его интеллекта и воли было создание АСУ всех видов и направлений. Отличный математик, конструктор великолепной ЭВМ МИР-1, Глушков был также сторонником полной автоматизации. Работы по созданию АСУ на базе отечественных универсальных цифровых вычислительных машин были начаты по его инициативе в 1963-1964 годах. Первой в СССР системой для предприятия с крупносерийным характером производства стала АСУ «Львов», внедрённая на Львовском телевизионном заводе «Электрон». В конце 1960-х — начале 1970-х под руководством академика была создана типовая система «Кунцево», внедрённая на Кунцевском радиозаводе. Благодаря его инициативе были приняты решения о разработке 600 систем для машиностроительных и приборостроительных предприятий девяти оборонных министерств СССР на основе «Кунцево».

Новый этап в развитии АСУ пришёлся на вторую половину 1970-х и 1980-е годы с созданием ОГАС (общегосударственной автоматической сети связи). Это был грандиозный по масштабам социально-научный эксперимент, длившийся три пятилетки, главным мотором, идеологом и руководителем которого и стал В.М. Глушков. К 1980 году были введены в строй 5097 АСУ организационного типа, в том числе и для высших органов управления. В Госплане был АСПР, в ЦСУ — АСГС, в ГКНТ — АСУНТ и т. д. Практически все предприятия с численностью более 5000 работающих имели свои автоматизированные системы. Около 500 НИИ и КБ специализировались на разработках в области автоматизации; поскольку уровень унификации был невысок, каждая АСУ была, по сути, уникальной системой. Сформировалась также система подготовки кадров всех уровней — от высшего образования до подготовки специалистов высшей категории.

Конечно, были получены кое-какие новые результаты, но реального эффекта не было. По своим прямым результатам АСУнизация управления промышленностью оказалась неэффективной. Были затрачены колоссальные средства, а эффект от совершенствования управления отсутствовал полностью.

Однако появление вычислительной техники объективно потребовало формализации и экономических знаний. В НИИ и на кафедрах стали осваиваться математические методы в экономик. Кибернетика стала брэндом, под которым в СССР внедрялась западная экономическая теория.

В 1963-м создаётся целый институт — Центральный экономико-математический (ЦЭМИ). Именно здесь под флагом экономической кибернетики разрабатывается СОФЭ — система оптимального функционирования экономики, то есть определённая теория с развитым математическим аппаратом, позволяющим оптимизировать экономические решения. Неслучайно в числе ведущих экономистов перестройки оказались академики С. Шаталин и Н. Петраков, работавшие именно в этом институте. За свой труд «Экономико-математические методы и модели» В. С. Немчинов в 1965 году вместе с Л. В. Канторовичем и В. В. Новожиловым были награждены Ленинской премией. Издаются журналы, книги, защищаются диссертации. Экономическая кибернетика преподаётся в десятках вузов, экономико-математические методы отрабатываются в ВЦ АН СССР, НИЭИ и ГВЦ Госплана, в отраслевых экономических институтах, Сибирском отделении и т.д.

Уже в 1964 г. появляется на русском языке популярнейший американский учебник по основам математической экономики Пола Самуэльсона, своего рода «Капитал» для экономистов-математиков. В отличие от философии и других гуманитарных наук, где идеологическая цензура ограничивала зарубежные публикации, под вывеской «экономико-математические методы» было опубликовано большинство ведущих западных работ XX века.

Другим направлением, родственным кибернетике, стало широчайшее распространение теорий, связанных с управлением: сетевое планирование, теория управления, прогнозирование, системные исследования, исследование операций и т. п. Особую роль сыграли научные разработки в военном секторе экономики Запада. Ими зачитывались и их активно цитировали как образец передовой науки. В СССР практически на всех руководящих постах трудились инженеры по образованию, и им был чрезвычайно близок технократи-

ческий проектный подход. Казалось, что если заменить «план» «программой», если заимствовать у Запада систему постановки целей с последующим развёртыванием их в систему задач, где возможны поиски оптимума, то что-то изменится, станет обоснованней и эффективней. Последние двадцать лет советского времени широко использовались такие инструменты управления, как «программы» — научно-технические, комплексные, а то и просто общенародные (типа продовольственной и жилищной). И всё это, подчёркиваю, было напрямую связано с кибернетикой. Те же авторитеты, те же технологии.

Правда, возникали трудности. У Мольера Журден обнаруживает, что он говорит прозой. После появления кибернетики те, кто изучал экономику, а также продвинутые начальники узнали, что они занимаются управлением. Это слово сегодня кажется обычным, но в советских условиях главным всё-таки было планирование, и вовсе не случайно главным хозяйственным органом был Госплан: руководство народным хозяйством означало составление плана, организацию его выполнения, контроль и учёт. Появление западной методологии создало проблему — надо было как-то примирить ведущую роль Госплана с кибернетическим управлением, как главной теоретической функцией. Те, кто работал на Госплан и разрабатывал автоматизированную систему для этого комитета, не могли признать никакой общей функции под названием «Управление» — потому-то здесь и разрабатывали Автоматизированную систему плановых расчётов (АСПР). Потому и вышеупомянутая ОГАС создавалась для «управления и планирования». Только так удалось совместить привычные термины и практику.

Теория управления стала ещё одним фронтом кибернетического наступления. Важную роль в пропаганде западных достижений в этой сфере, особенно на первых порах, сыграл Джермен Гвишиани, основатель Всесоюзного научно-исследовательского института системных исследований (ВНИИСИ). Именно его книги стали основополагающими при исследованиях и разработках проблем управления в 1970-е годы.

В экономике среди тех, кто первым взялся за пропаганду «буржуазных теорий управления», были теоретики С. Е. Каменицер, будущий известный политик Г. Х. Попов, возглавивший соответствующую кафедру МГУ ещё в 1971 году, основатель первого ВУЗа, где сразу претендовали на науку управления — МИУ. Кроме вышеперечисленных, стоит упомянуть В. Г. Афанасьева, будущего академика, а также главного редактора газеты «Правда», который также претендовал на роль ведущего управленца и «системщика» СССР.

Таким образом, кибернетический бум, плавно перешедший в «управленческий», «экономико-математический», продолжался практически до краха перестройки. Именно те, кто выдвинулся на основе кибернетики (за исключением Леонида Абалкина, пропагандировавшего диалектику), — соратник Горбачёва Вадим Медведев, помощник первого президента СССР Петраков, один из соавторов программы «500 дней» Шаталин — активно «реформировали» советскую экономику вплоть до её смерти.

Итак, либерализация экономических методов началась не при Егоре Гайдаре, а задолго до него, ещё в 1960-е, а потом она шла во всё больших масштабах. Но то, что давало результаты за кордоном, оказалось совершенно бесполезным в СССР. И, к сожалению, в сфере информатики нас сегодня обгоняет даже Индия. Мощнейший «кибернетический бум», сотни тысяч, а может, и миллионы людей, участвовавших в создании АСУ, внедрении математических моделей, в попытках совершенствования управления социалистической экономикой, — всё это обернулось крахом. Мы потерпели историческое поражение.

Отставание от ведущих стран в области технологии началось ещё в 1960-е. Транзисторы в массовом порядке заменялись интегральными, а потом и сверхбольшими интегральными схемами. К концу 1960-х технологический разрыв в области вычислительных машин достигал уже 6-7 лет. Это привело сначала к отставанию, а затем к тупиковому для отечественной информатики копированию разработок фирмы IBM. Работки в математике и собственно в информатике без отечественной технической базы оказывались всё менее оригинальными и интересными.

Символично, что Брежнев и Глушков умерли в одном и том же году — 1982-м. К этому времени звезда кибернетики действительно закатилась. В 1980-е получает широкое распространение термин «информатика», а понятие «кибернетика» постепенно исчезает из обращения, сохранившись лишь в названиях институтов, которые возникли в эпоху «кибернетического бума». Эффективные АСУ построить не удалось, отечественные ЭВМ на фоне только что появившихся персональных компьютеров демонстрировали свою полную историческую безнадёжность.

Среди причин краха компьютеризированных систем управления основной считают ту, что экономика вообще не может управляться сверхпланированием, кроме того:

- понятия «достать» и «выбить», на которых покоилась советская экономика, которыми преодолевался её тотально-плановый характер и которыми отсрочивался её крах, во-первых не являются формализуемыми, а во-вторых не могли быть введены в официальный советский оборот (и тем более — в качестве плановых индикаторов); учёт цехового рынка не мог быть введен в официальный советский оборот; успешная инновационная деятельность невозможна в системе, в которой распределение гаек и болтов расписано на 5 лет вперёд;
- советская система отторгала инновации и подавляла их, так как компьютеры и вообще инновации в советской системе реально никому были не нужны. То есть, именно системно не нужны. Кучки энтузиастов могли биться сколько угодно, и иногда удавалось что-то пробить, но в целом это всё было бесполезно. Единственное, что система ещё как-то воспринимала (и то всё хуже и хуже), это спецсистемы для оборонных задач. Но технология этих спецсистем не могла существовать в вакууме, она могла лишь настолько-то отгры-

ваться от общего технологического уровня. Что и привело к известным последствиям: чтобы обеспечить поставку приемлемого качества комплектующих, в оборонную сферу приходилось переводить все большее и большее количество предприятий;

- наиболее непосредственная из практических причин: СССР был просто не в состоянии произвести качественной элементной базы для производства какой угодно категории компьютеров из-за технологической отсталости техпроцессов.

- общая отсталость отечественных разработчиков.

График отставания советских разработок ЭВМ от США:

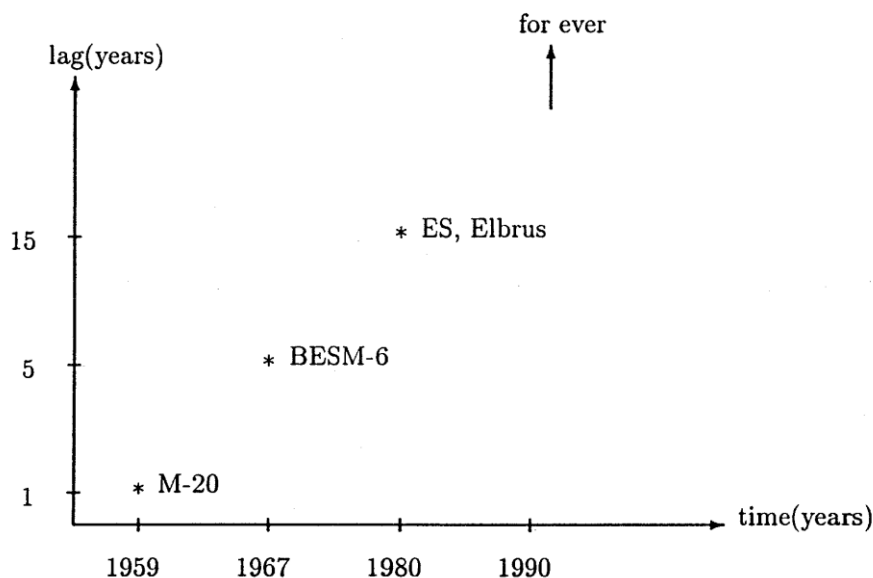


График отставания советских разработок ЭВМ от США

Историю кибернетики уместно закончить цитатами:

«Говорили, что кибернетика – реакционная лженаука. Это не так. Во-первых – не реакционная. Во-вторых – не лже, а в-третьих – не наука», – математик и биолог А. М. Молчанов.

«Кибернетика – это самый большой кусок, который человечество откусило от плода Древа Познания за последние 2000 лет. Однако большинство таких кусков оказалось совершенно несъедобными – как правило, по кибернетическим причинам», – Грегори Бэйтесон (Gregory Bateson).

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
I. МАТЕМАТИКА.....	5
1.1. Системы счисления	5
Двоичная система	6
Троичная система	7
Восьми и шестнадцатеричная система	10
Смешанные системы счисления	14
Машина Фибоначчи	15
Двоично-десятичное кодирование.....	17
1.2. Теория информации.....	19
Информация и энтропия.....	19
Кодирование и теоремы Шеннона	22
Обратимые вычисления.....	27
1.3. Математическая логика.....	30
Булева алгебра	30
Секвенциальная логика	33
Нечеткая логика и нейровычисления.....	34
NULL Convention Logic и асинхронные машины	48
Формальные системы	51
1.4. Машинная арифметика в кольце вычетов	54
Введение в алгебру	54
Арифметика в кольце вычетов.....	57
Вычисления в полях Галуа.....	58
Вещественная арифметика.....	60
К чему приводят ошибки.....	67
1.5. Алгоритмизация и машина Тьюринга.....	69
Теория графов.....	69
Алгоритмы и вычислимость.....	72
Конечные автоматы	76
Машины Тьюринга, Поста, фон Неймана	79
Сети Петри и клеточные автоматы	83
Вычисления на графах и потоковые ЭВМ.....	88
λ-исчисление и редукционные ЭВМ.....	94
Процессные алгебры и параллельные вычисления	98
II. ФИЗИКА	106
2.1. Аналог и цифра	106
Сигналы и их представление.....	106
Аналоговые компьютеры	108
2.2. Физика ЭВМ - настоящее и будущее	116
Явление полупроводимости.....	116
Транзистор	117
ЭВМ будущего – от мемристоров до оптических машин.....	119
Квантовые вычисления.....	126
Молекулярные машины.....	136
2.3. Основы схемотехники	138
Типы цифровых схем и их разработка.....	138
Типы логики.....	144

III. ИНФОРМАТИКА.....	147
3.1. Типы данных и их представление в машине.....	147
Типизация.....	147
Биты, байты, слова.....	153
3.2. Польская запись и стековые машины.....	158
Принципы бесскобочной записи.....	158
Стековая машина	160
3.3. Ассемблер	168
Основы FASM.....	175
Теговые машины.....	207
Ассоциативные ЭВМ.....	213
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	215
ПРИЛОЖЕНИЯ	218
Приложение 1. Краткая история вычислительной техники.....	218
Приложение 2. Основы квантовой физики.....	228
Приложение 3. Информатика против кибернетики. Взлет и крах советских ЭВМ	238

Учебное издание

Алексей Павлович Еременко

Аппаратное обеспечение ПК

Часть I

Введение в базовые принципы организации ЭВМ

Учебное пособие

Редактор – И.Т. Куликова

Подписано в печать 5.09.2013. Формат 60 × 90/8

Бумага писчая. Печать офсетная.

Усл.-п.л. 31,5. Тираж экз. Заказ №

Отпечатано: РИО, ВоГТУ 160000, г. Вологда, ул. Ленина, 15