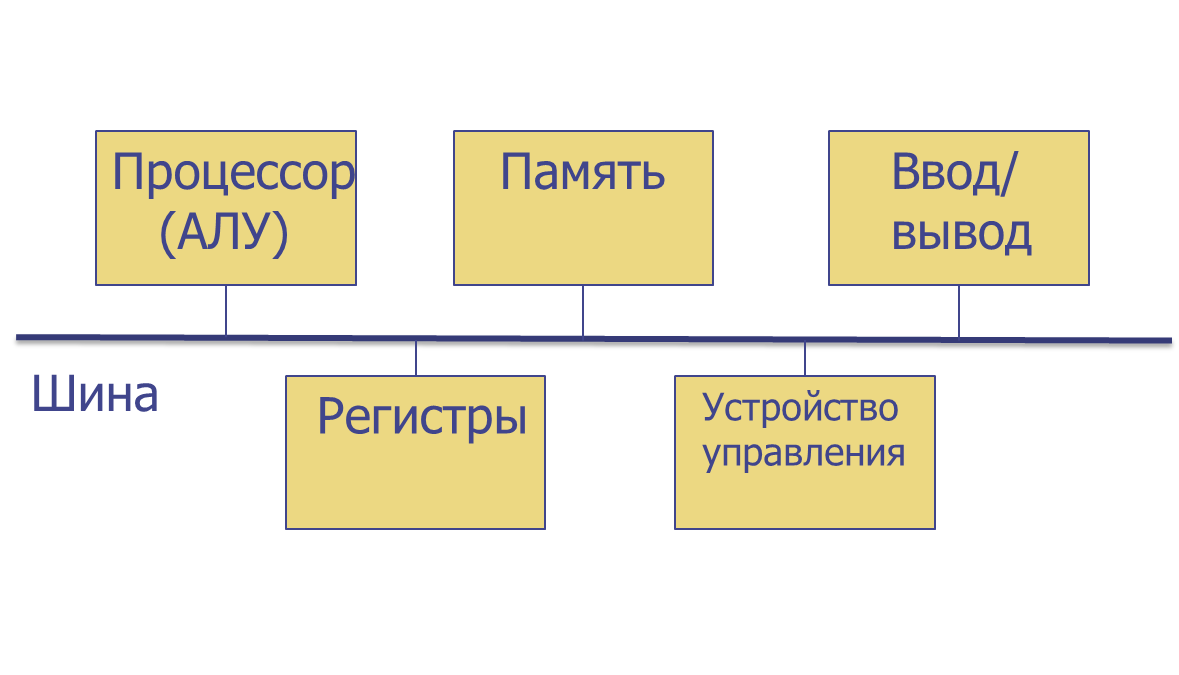
**Глава 1**

**Что такое архитектура ЭВМ**

**Часть 1. Что такое архитектура ЭВМ**



В принципе все современные ЭВМ (электронно-вычислительные машины) устроены примерно одинаково: процессор, память, устройство ввода/вывода, регистры, устройство усправления, таймеры и т.д. Когда я читаю этот курс в Университете я всегда спрашиваю студентов: “Как вы думаете, когда структурная схема, приведенная выше, была впервые создана?” Обычно говорят - 50-60-е годы 20-го века. На самом деле эта схема была создана Чарлзом Бэббиджем в Лондоне в 1834-37 гг, т.е. в начале 19-го века.

В то время он делал машину из шестеренок в соответствии с технологией того времени, но тем не менее все основные структурные элементы были представлены и до сих все это сохранилось, конечно, элементная база сейчас совершенно другая. В 1943 году Марк 1 был сделан из шестеренок и электромагнитных реле. Сегодня все это уже ушло в прошлое. Современные устройства создаются из электронных элементов под названием Триггер. Эту схему изобрел русский ученый М.А. Бонч-Бруевич в 1918 году в Петрограде.

**Организация ЭВМ на уровне ассемблера**

Допустим, надо выполнить формулу: C = A + B, тогда в кодах некоторой ЭВМ программа выполнения этой формулы может выглядеть так:

0001 0101

0011 0110

0010 0111

Разумеется, эта запись плохо читается, автору надо помнить, что адрес 0101 – это адрес переменной A, по адресу 0110 расположена переменная B, а С имеет адрес 0111, не говоря уж о запоминании кодов операций. Особенно досаждал первым программистам тот факт, что вставка любой команды или элемента данных менял все последующие адреса, поэтому приходилось заново набивать программу на перфокартах или перфолентах.

Буквально через несколько лет были придуманы языки ассемблера, в которых каждая переменная или адрес в коде обозначались своим идентификатором, тогда на языке ассемблера некоторой ЭВМ программа выполнения этой же формулы может выглядеть так:

L R0, AddrA

A R0, AddrB

ST R0, AddrC, где

L (load) - загрузка из памяти в регистр;

A (add) – добавить;

ST (store) – запомнить значение регистра в памяти.

Тогда же появились и первые системные программы – ассемблеры, которые переводили текст на языке ассемблера в двоичные коды ЭВМ. Алгоритм работы ассемблера до смешного прост: исходный текст программы просматривается дважды, на первом просмотре составляется таблица идентификаторов, в которой каждому идентификатору присваивается его адрес, а на втором просмотре эти адреса подставляются в программу. За один просмотр ассемблер реализовать сложнее (хотя при наличии большой памяти можно), так как описания некоторых идентификаторов могут встретиться после их применения.

Программы на языке ассемблера выглядят, конечно, лучше, чем программы в двоичных кодах, но все равно, это слишком мелкозернистое программирование, позже появились алгоритмические языки высокого уровня, которые позволили писать программы в существенно более крупных легко понимаемых конструкциях, таких как последовательные и условные предложения, массивы и циклы, процедуры и процессы. А еще позже появились предметно ориентированные языки (для каждой предметной области свои), которые еще более подняли производительность программистов, но это уже предмет другого курса.

Есть понятие исполнительный адрес, он может быть сложным: складывается из базового регистра, который указывает на начало участка памяти, индексного регистра и смещения. “Много копий сломано” на тему, какое должно быть смещение, какой длины, но практически все машины выглядят примерно именно так, поэтому можно спокойно изучать структуру ЭВМ на примере одной машины и будьте уверены, что на других машинах это будет выглядеть очень похожим образом.

Итак, исполнительный адрес складывается из базового регистра, возможного индексного регистра и смещения.

**Команды и данные**

Гарвардская структура ЭВМ - Марк 1 был сделан в Гарвардском университете, правда, за деньги IBM. Все исполнялось на отдельных устройствах, т.е. память команд была физически отделена от памяти данных. До сих пор Гарвардская архитектура означает, что команды и данные хранятся в разных устройствах памяти. С одной стороны, это повышает надежность, потому что никаким образом по ошибке программист не может испортить чужую или свою программу, но у данной структуры есть масса неудобств: приходится делать два устройства для чтения с диска, два средства выборки данных и команд разных устройств. В современной военной практике все машины устроены по гарвардской архитектуре именно из соображений надежности.

История создания другой архитектуры - архитектуры фон Неймана также интересная. Джон фон Нейман - математик венгерского присхождения, еврей из Будапешта. Когда он закончил школу в 1917 г., в Будапеште был страшный антисемитизм, поэтому он поехал поступать в университет в Берлин (в то время в Берлине антисеметизма не было). Он проучился в университете 3 года, потом переехал в Цюрих и завершил обучение там. К моменту выпуска из университета ETH в Венгрии произошли революция, гражданская война, и антисемиты дома утихомирились, поэтому он вернулся домой. Там он жил и работал, получил великолепные математические результаты, к программированию отношения не имеющие. Регулярно ездил в Америку читать лекции, в итоге решил там остаться. По этой причине американцы считают его великим американским ученым, хотя он туда приехал будучи уже доктором наук. Он имел очень широкое образование, в частности, поэтому во время войны участвовал в расчетах зенитных пушек. Дело в том, что самолет имеет довольно большую массу, поэтому не может резко повернуть, а летает по плавным кривым. Если через локатор за ним последить несколько секунд, можно определить параметры его траектории в этом коротком промежутке времени и предсказать, где он будет находиться через те доли секунды, которые нужны снаряду, чтобы долететь от зенитной пушки до самолета. Именно такими расчетами и созданием соответствующих приборов и занимался фон Нейман во время войны, причем добился замечательных результатов.

В этой работе он сталкивался с людьми, которые делали расчеты, поэтому он слышал про о создателях Марк 1 и Эниак (первая действительно электронная вычислительная машина, 1946 г.), там была тройка авторов: Моучли, Эккерт и Гольдстайн. Гольдстайн в это время служил в армии США, но до войны был профессором в университете, поэтому он понимал в технике. Фон Нейману были интересны семинары, на которых Моучли, Эккерт и Гольдстайн обсуждали возможность развития архитектуры.

Стоит отметить, что инженеры никогда никогда не торопятся публиковать свои результаты, так как они предпочитают патентовать свои изобретения, чтобы получать доход, а у математиков патентов нет и принято публиковать каждый результат. Так вот, фон Нейман сделал записи этого семинара и выпустил First Draft (отчет по первому вычислителю), где собрал воедино информацию, которая была придумана Гольдстайном и др. Он сделал 21 экземпляр, и они быстро разошлись по всему миру. Один даже попал в Англию. Там было описано, что можно хранить программу и данные в одной памяти и это удобней, эффективней, традиционно такую архитектуру стали называть архитектурой фон Неймана, хотя придумал не он.

Анекдот: через много лет после смерти всех фигурантов, родственники Гольдстайна, Эккерта и Моучли судились с родственниками фон Неймана, чтобы название архитектуры сменили. Но судьи посмеялись и название оставили.

Итак, В гарвардской архитектуре команды и данные хранятся в разных устройствах памяти.

В архитектуре фон Нейманакоманды и данные хранятся в одной и той же памяти.

**Каманды и данные**

Память всегда была медленнее, чем процессор. Когда я был молодой, коэффициент был 4, то есть память была в 4 раза медленнее, чем процессор. Сейчас электроника развилась таким образом, что сегодня память медленнее, чем процессор в 15-20 раз, поэтому еще острее сказывается проблема: вы взяли команду из памяти, исполнили ее и теперь вам надо исполнять следующую команду, но процессор будет стоять долго-долго, пока читается следующая команда. В 1955 году советский ученый Сергей Алексеевич Лебедев придумал схему, которую он назвал по-русски “водопровод”, то есть делается отдельное устройство, которое читает несколько команд вперед, пока одна команда исполняется, несколько команд читаются впрок. Но поскольку есть жёсткая статистика, что примерно каждая 6-я команда - это передача управления, то делать водопровод больше чем на 6-8 позиций смысла нет, поэтому возникла такая константа: *водопровод нужно делать не больше, чем на 6-8 позиций.* Американцы, конечно, это первенство не признают, утверждая, что они всё это знали, но первая публикация, причём на английском языке, про водопровод - это публикация Лебедева в 1955 году, а других публикаций на эту тему до этого времени не было, поэтому мы искренне уверены, что водопровод придумал именно Лебедев, первый в мире. В его статье, которую американцы перевели на английский язык, был термин “Pipeline”, который перевели с русского на английский, а не наоборот. Сейчас американцы называют его fetching.

Итак, чтобы остальные устройства не простаивали, пока очередная команда читается из памяти, устройство управления организует предварительное чтение из памяти и дешифрацию команд на фоне работы АЛУ и других устройств – водопровод (pipeline)

**Часть 2. Цифровая логика**

Если посмотреть на осциллограф, то такой прямоугольный полочки идеального сигнала вы никогда не увидите. Если есть “горочка”, то - это единичка, если ближе к нулю, то это нолик.

Идеальный сигнал:



Реальные сигналы:



С реальным сигналом у меня связана ужасная история. В 1987 году мы сделали вычислительную машину Самсон. Машина была оригинальная, не копия какой-то американской машины, как тогда было принято. Заказчиком было Управление правительственной связи КГБ, поэтому работу поддерживал оборонный отдел ЦК КПСС, но даже с такой поддержкой ни один советский завод по производству ЭВМ не взялся нашу машину выпускать, а мы объехали практически все такие заводы, кроме армянского. В конце концов, первые 100 экземпляров Самсона сделали на заводе в городе Пловдиве совместно с болгарскими коллегами, причем оформили это как подарок к 70-летию Советской власти.

На 14 часов 5 ноября 1987 года было назначено первое предъявление Самсона, приехали несколько членов Политбюро болгарской коммунистической партии и КПСС, но в 13 часов Самсон еще не работал. Мы несколько суток практически не спали, но понять в чем дело не могли. В последний момент болгары привезли хороший западный осцилограф, с помощью которого мы увидели на тактовом сигнале маленький пичок-помеху. Инженеры быстро припаяли какой-то конденсатор, помеху сняли, и Самсон заработал. На советском осцилографе мы этой помехи не видели.

Следующее понятие - это clock. Clock - это тактовая частота. Эти импульсы управляют работой машины.

****

Сигналы участвуют в работе цифровой логики. Придумал эту логику Джорж Буль. (В советское время ее называли алгеброй Жегалкина.) Вентиль *И*выдает 1, когда оба операнда 1, *Или* выдаёт единицу, когда хотя бы один операнд единица, *Исключающее или* - это когда 0,1 и 1, 0 дают 1, а 1,1 и 0,0 дают 0.

На самом деле ЭВМ уже давно не делаются из вентилей. Базовые элементы современных ЭВМ - это триггеры которые, в свою очередь, состоят из нескольких вентилей. Несколько сотен тысяч или даже миллионов триггеров комплексируются в один кристалл – чип, вот из этих чипов и делаются ЭВМ.

Триггер– схема с 2 устойчивыми состояниями, т.е. память на 1 разряд. В 1916 году петроградский (так тогда назывался наш город) ученый М.А.Бонч-Бруевич создал первый в нашей стране триод – электронную лампу из трех элементов (анод, катод и сетка), в первых публикациях он называл триод катодным реле (какие-то лампы уже существовали, здесь речь идет о специальной вакуумной конструкции), а в начале 1918 года он придумал схему триггера на основе двух ламп-триодов. Видимо, независимо от М.А.Бонч-Бруевича практическая схема триггера была опубликована [5 августа](https://ru.wikipedia.org/wiki/5_%D0%B0%D0%B2%D0%B3%D1%83%D1%81%D1%82%D0%B0) [1920 года](https://ru.wikipedia.org/wiki/1920_%D0%B3%D0%BE%D0%B4_%D0%B2_%D0%BD%D0%B0%D1%83%D0%BA%D0%B5) американцами [У.Г. Икклзом](https://en.wikipedia.org/wiki/William_Eccles) и [Ф. У. Джорданом](https://en.wikipedia.org/wiki/F._W._Jordan) в [патенте](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%82%D0%B5%D0%BD%D1%82) [Великобритании](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BB%D0%B8%D0%BA%D0%BE%D0%B1%D1%80%D0%B8%D1%82%D0%B0%D0%BD%D0%B8%D1%8F) № 148582, заявленном 21 июня 1918 г и в статье «[Переключающее реле, использующее трёхэлектродные вакуумные лампы](https://en.wikipedia.org/wiki/Flip-flop_(electronics)#History)»от [19 сентября](https://ru.wikipedia.org/wiki/19_%D1%81%D0%B5%D0%BD%D1%82%D1%8F%D0%B1%D1%80%D1%8F) [1919 года](https://ru.wikipedia.org/wiki/1919_%D0%B3%D0%BE%D0%B4).

Регистр– несколько триггеров, рассматриваемых как 1 устройство (8,16,32 бита) - это просто много битов, соединенных в одно устройство.

Есть отдельный термин **“**Счетчик**” –** регистр, который может выполнить одну операцию +1. Счетчики из триггеров применялись в нашей стране еще до Отечественной войны как элементы сложных приборов, но, разумеется, до программируемых ЭВМ этим приборам было еще далеко.

**Жесткие кристаллы**

Первые кристаллы - чипы были сделаны в фирме Intel.

Что такое чип? *Это множество (миллионы) вентилей, соединенных в сложную схему и упакованных в единый «камень» (ЧИП).* Маленькая штучка размером с фалангу мизинца, и она умеет делать большое количество операций в секунду, поскольку внутри нее все провода очень тонкие, это дает высокие скорости и малое энергопотребление, но разработать жесткий кристалл стоит очень много денег и несколько лет работы.

**Гибкие кристаллы**

Позже придумали гибкие кристаллы FPGA (Field-Programmable Gate Array или ПЛИС - Программируемая Логическая Интегральная Схема) – кристалл, имеющий управляющую (иногда говорят конфигурационную) оперативную память (bitmap), биты которой определяют логику работы отдельных элементов ПЛИС. Сегодня элементами ПЛИС являются не просто вентили, а более сложные элементы, прежде всего, LUT (Look-Up Table), микросхема ПЛИС может содержать также внутреннюю память (BRAM), умножители, DSP (подпроцессоры цифровой обработки). LUT – это просто логическая функция, скажем, 6 входов и 1 выход, задаваемая таблицей-памятью из 64 бит, в каком-то смысле 6 входов LUT являются адресом в этой памяти, которая заполняется из bitmap при включении питания. Таким образом, разные LUT могут исполнять разные логические функции.

В управляющую оперативную память кристалла можно много раз записывать последовательность битов. Одна последовательность соответствует одной электронной схеме, другая последовательность заставляет кристалл работать по другой схеме. Представьте, если у вас есть 1000 000 вентилей (а это не так много), то понятно, что соединительных линий может быть миллион в квадрате, которые ни один человек даже представить себе не может. Но есть специальные программные средства, которые умеют это делать. LUT-ами занимается всего несколько фирм в мире, главной из которых является Xilinx. Компании, которые производят гибкие кристаллы, содержат в строжайшей тайне структуру LUTа, как заполняется их память, как проецируются операции языков проектирования кроисталлов в логические функции LUTов, поэтому скопировать производство ПЛИС очень тяжело, но можно разработать самим!

**Программирование гибких кристаллов**

**VHDL**

**bitmap**

**Verilog Mentor Graphics**

**Cadence**

**Synopsys**

Для программирования гибких кристаллов применяются разные языки программирования, VHDL и Verilog - основные. Есть дорогущие программы: Mentor Graphics, Cadence, Synopsys, которые по тексту программ на VHDL и Verilog могут получить bitmap. Эти языки были созданы более 25 лет назад, и они были созданы для совсем других целей: документирование кристаллов, но не проектирование. Поскольку программирование кристаллов на этих языках - очень тяжёлая работа, много лет назад мы придумали свой язык HaSCoL (Hardware and Software Codesign Language). Программа на HaSCoL во много раз короче, чем на других языках, легче понимается и исправляется. Первое название этого языка было очень красивое и понятное - PADLA (Processor Architecture Definition Language). В 2004 году мой аспирант Дмитрий Булычев защитил диссертацию по реализации этого языка, но потом на нас стали нападать, что название некрасивое, и даже мои доводы в пользу того, что это только аббревиатура, не помогли, и нам пришлось сменить название на HaSCoL. Сегодня на этом языке мы уже разработали довольно много кристаллов. У нас есть свои средства генерации с отладкой, трессировкой, то есть полноценная технология, применяемая в самых разных областях. Были проведены промышленные эксперименты и доказано, что то, что обычный инженер делает на VHDL за полгода, специалист, работающий на нашем языке, делает то же самое за неделю/две, не больше.

**Часть 3. Представление данных**

Как данные представляются в ЭВМ.

Бит– 1 разряд (триггер),

Байт– 8 бит (с 1964 г.) 1 литера,

Слово– 16 и 32 бит (целое число),

Двойное слово – 64 и 128 бит (вещественное число).

**Целые числа**

Целые числа представляются очень просто – в старшем разряде знак числа, затем двоичное представление числа. В основном, применяется дополнительный код, в котором положительные числа представляются естественным образом, а отрицательные числа получаются инвертированием всех разрядов числа с последующим добавлением 1 к младшему разряду, в результате получается дополнение числа до максимального числа «единица и 16 или 32 нуля». Легко сообразить, что тогда операцию вычитания можно заменить сложением с дополнительным кодом вычитаемого.

**Вещественные числа**

Как представляются вещественные числа? Тут тоже нужно немного углубиться в историю: еще 20-30 лет назад каждая машина имела собственное представление вещественных чисел, это страшно мешало работе, потому что не было возможности перенести одну программу с одной машины на другую. Американский профессор математики Вильям Каган из университета Беркли придумал новый стандарт, который сначала стал стандартом «де факто», а затем стал и формальным стандартом. Вещественное число представляется в виде мантиссы – дробного числа в интервале (1, 2), умноженного на 2 в степени порядок. Понятно, что, увеличивая или уменьшая порядок, можно любое число преобразовать так, чтобы мантисса попадала в интервал (1, 2), но при этом старшим битом всегда будет 1, поэтому ее можно в памяти не хранить, а подразумевать. По техническим причинам порядок удобнее хранить не в виде целого числа со знаком, а в виде положительного числа, сдвинутого на 127 или на 1023 в зависимости от точности.

Одинарная точность:

|  |  |  |
| --- | --- | --- |
| **знак** | **порядок** | **мантисса** |
| 31 | 30 23 | 22 0 |

Порядок 8 бит со сдвигом 127, мантисса не 23, а 24 бита, старшая 1 подразумевается.

Двойная точность:

|  |  |  |
| --- | --- | --- |
| **знак** | **порядок** | **мантисса** |
| 63 | 62 52 | 51 0 |

Порядок 11 бит со сдвигом 1023.

**Особенные вещественные числа**

Есть несколько вещественных чисел, которые имеют специальное значение:

NaN (not a number)

Порядок- все единицы

Мантисса – не нулевая

Бесконечность

Порядок – все единицы

Мантисса – 0.

В современных ЭВМ бесконечность вполне может быть аргументом арифметических операций, разумеется, результат тоже будет бесконечностью.

Нуль

Порядок – 0

Мантисса – 0

**Денормализованные числа**

Порядок – 1

Мантисса – не равна 0, подразумеваемый старший разряд не 1, а 0. Денормализованные числа дают возможность работать с очень маленькими значениями, но дают большое замедление счета.

**Часть 4. Представление нечисловых данных**

Надо начать с истории - первые лет десять ЭВМ использовались исключительно для военных нужд. Машины работали только с числами, но уже в 1951 году американцы выпустили машину Юнивак и стали применять ее в коммерческих нуждах: печатали ведомости, счета и т.д. И вот был придуман первый стандарт представления литер:

ASCII (American Standard Code for Information Interchange), 1963 г., первый вариант был семиразрядным, этого хватало для представления латинских букв и специальных знаков, позже появились восьмиразрядные стандарты, в частности, в СССР был выпущен стандарт КОИ 8, в котором были и русские буквы.

Как обычно, фирма IBM не захотела следовать стандартам, созданным не в ней, и выпустила свой:

EBCDIC (Extended Binary Coded Decimal Interchange Code), 1964 г., поскольку в СССР массово выпускались ЕС ЭВМ – копии IBM/360, а затем и IBM/370, пришлось сделать аналог EBCDIC - ДКОИ 8.

Постепенно международное сообщество осознало, что 8 битов для представления литер слишком мало, был придуман двухбайтовый формат:

Unicode, 1991 г., 2 байта, UTF 16, но буквально через год был придуман более гибкий формат, в котором для представления литеры можно использовать до 6 байтов (для русских букв нужны всего 2 байта), но для латинских букв и основных знаков оставалось однобайтовое представление:

UTF 8 (Unicode Transformation Format), 1992 г.

**Как кодируется UTF 8**

В первом байте часть битов указывает количество байтов в коде литеры:

0ххххххх –1 байт, совпадает с ASCII ,

110 ххххх –2 байта,

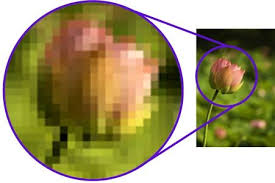
1110хххх –3 байта, и т.д. (всего до 6 байтов)

В остальных байтах кода 10хххххх

**Представление графических данных**

**Растровое изображение**

**Растовое изображение** - совокупность точек, используемых для отображения на экране монитора. Каждая точка имеет свою кодировку и называется пиксель.

Рис. 1

**Черно-белое изображение**

Черно-белое изображение, пиксель – 1 байт,

0000000 – не подсвечивается

11111111 – черный

Между ними оттенок серого.

**Цветное RGB изображение**

Цветное RGB, пиксель – 3 байта, яркость красного, зеленого и синего.

Например, в HTML:

# FF0000 интенсивный красный

# 00FF00 интенсивный зеленый

# 0000FF интенсивный синий

# 000000 черный

# FFFFFF белый

FF– наибольшая яркость

**Векторное изображение**

Векторное изображение - набор примитивов из элементарных отрезков кривых с параметрами (координаты узловых точек, радиусы кривизны и пр.), которые описываются математическими формулами. Для представления диаграмм, чертежей и т.п. векторные изображения очень удобны и компактны.

**Часть 5. Представление записей и массивов**

**Представление записей**

Запись(иногда говорят «структура») – последовательность полей произвольных типов, например:

struct {float a; int b; char c;} s;

Если float занимает 8 байтов, int 4 байта, а char 1 байт, то s займет в памяти 13 байтов, но если написать

struct {char c; int b; float a;} s1;

то s1 может занять и 20, и 24 байтов, так как современные устройства памяти требуют определенного выравнивания (например, если хочешь прочитать или записать 8-ми байтовое значение, то адрес должен быть кратен 8).

**Доступ к полю записей**

struct {float a; int b; char c;} s;

s.a, s.b, s.c – выборка поля записи.

Выборка поля – статический объект, нельзя, например, организовать цикл по полям записи, зато выборка ничего не стоит, так как смещение каждого поля записи можно определить во время трансляции.

Тип поля может быть любой, например, записью или массивом.

**Представление массивов**

Массив **–** это последовательность элементов одинакового типа, например:

[L : U] int A;

Во многих языках программирования границы массива L и U могут вычисляться во время счета, в Паскале и Си – только константы, причем в Cи L всегда равен 0. Адрес элемента массива А[i] вычисляется по формуле:

*+ i\*d,*

где - индекс элемента с индексом 0, а *d* – шаг, то есть размер элемента.

Бывают многомерные массивы, например:

[L1:U1, L2:U2] int M;

Тогда адрес элемента M[i,j] (иногда говорят «вырезка элемента») вычисляется по формуле:

+ i\*d1 + j\*d2,

где d1 – шаг, по первому измерению (в данном случае – длина строки), d2 – шаг по второму измерению. Эта формула очевидным образом обобщается на массивы произвольной размерности.

**Векторы Айлифа**

В некоторых языках (например, в Си) многомерные массивы представляются специальным образом, например, массив

int M [U1] [U2];

представляется как одномерный массив из

U1 элементов, каждый из которых является адресом одномерного массива из U2 элементов, тогда вырезка элемента записывается M[i] [j].

Представление массивов в виде векторов Айлифа позволяет уменьшить количество дорогостоящих операций умножения, кроме того, можно сформировать, например, двумерный массив из строк разной длины.

**Часть 6. Системы и типы команд**

Современные ЭВМ имеют достаточно много регистров, доступ к которым во много раз эффективнее, чем доступ к памяти. Команды ЭВМ делятся по формату (регистр – регистр, регистр – память, память - память) и по типу:

* Работа с памятью (чтение в регистр, запись из регистра в память, семафорное чтение, пересылка из памяти в память);
* Арифметические команды (+; -; \*; /);
* Логические команды (И; ИЛИ; НЕ);
* Команды передачи управления;
* Команда ввода/вывода SIO.

Эта команда запускает работу канала ввода/вывода, номер которого является параметром команды, после чего центральный процессор продолжает свою работу до прерывания, свидетельствующего об окончании операции.

**Команды передачи управления**

Команды передачи управления бывают безусловные, когда управление передается на заданный адрес, и условные, когда управление передается только при выполнении определенного условия, если же условие не выполнено, то управление «проваливается» на следующую команду. Если управление произошло, то водопровод необходимо разгонять заново, в противном случае водопровод уже содержит все необходимые команды.

* Безусловная передача управления

Branch Label

Label **–** метка в программном коде, куда нужно передать управление

* Условная передача управления

Branch Condition Label

Condition **–** условие, при выполнении которого необходимо совершить передачу управления, в противном случае управление передается на следующую команду

**Механизм вызова подпрограмм**

Подпрограммы – важный элемент структурирования больших программ, если какой-то фрагмент программы повторяется в нескольких местах, то его стоит вынести в подпрограмму. Во всех ЭВМ существует команда типа BAL R, Label (Branch And Link), которая запоминает в регистре R адрес следующей после нее команды, а затем передает управление на метку Label, указывающую на первую команду подпрограммы. В конце подпрограммы ставится команда

BR R (передача управления по содержимому регистра R), осуществляющая возврат из подпрограммы в основную программу.

В больших компаниях создают и строго следуют соглашению о связях, в частности, для возврата из подпрограммы используют всегда один и тот же регистр (например, в IBM – это 14-й регистр). Поскольку внутри подпрограммы могут быть вызовы других подпрограмм, в начале каждой подпрограммы все регистры запоминаются в памяти, а перед возвратом восстанавливаются. В стековых машинах, например, в архитектуре ARM в начале подпрограммы запоминают адрес возврата в стеке, а перед возвратом выбирают его из стека. Это позволяет безопасно использовать вызовы других подпрограмм из текущей подпрограммы, но о других регистрах общего назначения должен заботиться автор программы.

**BAL R, Label**

**Label:**

**BR R** Рис. 2

**Прерывания**

В некоторых случаях нужно прервать последовательное исполнение текущего процесса и запустить какую-то служебную программу, например,

* Аварийная ситуация (деление на 0, переполнение, выход индекса за границу массива и т.д.)
* Завершение операции ввода/вывода
* Завершение кванта времени, выделенного текущему процессу.

**Обработка прерываний**

* Каждое прерывание имеет свой номер;
* В конце памяти размещается последовательность команд безусловной передачи управления;
* При возникновении прерывания с номером N, происходит аппаратная передача управления по адресу K-N, где K – адрес конца памяти, а там стоит команда передачи управления на соответствующий обработчик.

Такая косвенная передача управления при возникновении прерывания нужна для возможности подмены обработчиков.

Когда появились первые «наладонники» и персональные ЭВМ, оперативной памяти было очень мало. Тогда хаккеры (в те времена это слово еще не носило современного отрицательного оттенка – просто люди, знающие детальную организацию ЭВМ) использовали прерывание вместо вызова подпрограмм, экономя 1-2 байта на каждом вызове. С обработкой прерываний связана одна из самых дорогостоящих ошибок мира – авария ракеты «Ариан-5» (4 июня 1996 года).

**Часть 6. Функциональная организация устройств**

Первые ЭВМ имели прямую аппаратную реализацию для всех своих команд, если какое-то действие, например, сложение использовалось в нескольких командах, для каждой такой команды формировался свой сумматор, то есть реализация была слишком избыточной с точки зрения объемов аппаратуры. И сегодня суперкомпьютеры реализуются именно так.

В 1951 г. Морис Винсент Уилкс (Кембридж, Англия) предложил принципиально другое решение – каждая команда ЭВМ реализуется в виде последовательности мелких аппаратных действий (микрокоманд), аппаратную реализацию которых можно переиспользовать в разных командах ЭВМ. Это позволяет существенно уменьшить избыточность аппаратуры, но уменьшает скорость работы ЭВМ из-за времени выборки микрокоманды из микропамяти и ее интерпретации.

Морис Уилкс написал книгу “Микропрограммирование”, которую я сам лично держал в руках. Представляете, в 51 году он придумал, написал труд по-английски, и наши в те годы так быстро сработали, что в этом же году перевели эту книгу и даже издали. Кстати, он был одним из тех, кто получил на одну ночь First Draft фон Неймана. За одну ночь он во всём разобрался, несмотря на ограниченность по времени. В то время в отношении труда фон Неймана действовал принцип: прочитай и передай товарищу.

Так вот Морис придумал следующее: любая команда ЭВМ распадается на маленькие действия под названием микрокоманды. Эти микрокоманды хранятся в микропамяти, память маленькая, но довольно быстрая по сравнению с обычной памятью. ЭВМ является аппаратным интерпретатором микрокоманд, то есть она после исполнения одной микрокоманды переходит к другой, которую она читает из микропамяти. Там также есть водопровод. Таким образом, получается гибкость, которая заключается в том, что одно и то же устройство может использовать в разных командах. Это безусловно экономит всё, но в какой-то степени замедляет процесс, потому что появляется интерпретация, поэтому все современные машины сегодня работают на микропамяти, только суперЭВМ делают это напрямую через аппаратную реализацию.

**Микропрограммирование**

На протяжении каждого такта всей ЭВМ управляет регистр микрокоманд – двоичная шкала размером 60-100 разрядов, в которой каждое поле управляет каким-то элементом ЭВМ. Эта шкала и называется микрокомандой.

Группа микрокоманд, реализующая одну команду ЭВМ, называется микропрограммой.

Микропрограммы хранятся в специальной быстрой микропамяти

Устройство управления выбирает из водопровода очередную команду, по ее коду определяет адрес начала соответствующей микропрограммы и загружает первую микрокоманду в регистр микрокоманд.

Одно из полей микрокоманды управляет выборкой следующей микрокоманды, которая осуществляется по концу текущего такта.

**Реализация микропрограмм**

Первые микропрограммы первых ЭВМ так и готовились в виде длинных последовательностей нулей и единиц

Позже появились микроассемблеры (самый известный фирмы AMD), но резкого повышения производительности труда микропрограммистов не произошло, для составления микропрограмм одной ЭВМ требовалось десятки человеко-лет. Когда мы начали делать машину Самсон, мы объехали несколько заводов в СССР, но никто не хотел ставить на производство ЭВМ, которая не является копией американской машины. Мне запомнился завод имени Орджоникидзе в Минске. Там был отдел микропрограммирования, под руководством сотрудника по фамилии Романовская. У неё в подчинении было 40 человек, и они делали микропрограммы для ЕС ЭВМ 1035 два года. То есть это и правда очень тяжелая работа**.**

В нашем коллективе Николай Фоминых придумал и реализовал способ задания микрокоманд на алгоритмическом языке высокого уровня (на Алголе 68), использовав возможности этого языка по созданию специализированных языков на его основе с помощью описания новых типов и операций над данными этих типов.

**Пример микропрограммы**

Пусть надо реализовать команду «Ч Смещение» – загрузить на стек слово из памяти по адресу регистр L + Смещение (1 байт)

(ВД1 = > АЛУ ( D + L ) = > БА, КМК) ;

(АЛУ (вверх), КМК) ;

(БД = > АЛУ ( D -> S), КМПУ (Г)) ;

(БД = > АЛУ ( D- > S), КМП);

**Необходимые пояснения**

В методе Николая Фоминых микропрограммист по-прежнему пишет каждую микрокоманду отдельно, но система не даст ему написать неправильную операцию или операцию с неправильными типами данных, например,

= > - операция пересылки данных по шине компьютера

- > - операция пересылки внутри АЛУ

+ - это не сложение целых чисел, а обозначение соответствующей операции внутри АЛУ

Таких операций Николай Фоминых описал более сотни, основной объем этих описаний занимают проверки соответствия запрошенных действий аппаратуре кристаллов.

Итак, микропрограмма– это обычная программа на языке Алгол 68, запуск которой приводит к порождению битовых шкал микрокоманд.

В нашем коллективе так уж повелось, что первым пользователем всех создаваемых нами технологий являюсь я. Сначала это была проверка качества технологии (то, что теперь называется QA – обеспечение качества), потом это превратилось в традицию. За 1 месяц я, не бросая других своих многочисленных обязанностей, написал все микропрограммы для всех команд Самсона и пришел поздравить Колю с такой хорошей технологией, но он, только бросив взгляд на мои труды, сказал, что это совсем не похоже на микропрограмму. Увидев обиду на моем лице, он обяснил, что каждая микрокоманда должна быть широкой, обеспечивающей одновременную работу многих разных устройств, а я написал микропрограммы в виде «тонкой колбаски» в стиле обычного программирования. Еще две недели я потратил на переписывание с учетом Колиных замечаний, после чего Коля сисходительно сказал, что теперь это на что-то похоже.

С того момента труднейшая задача микропрограммирования превратилась в типовую студенческую задачу уровня курсовой, но уж никак не дипломной. У меня есть куча историй, как и с помощью каких приемов нам удалось ускорить самые частые команды Самсона, но, боюсь, они далеко выходят за рамки курса по архитектуре ЭВМ.

**Глава 2**

**Память**

**Часть 1. Иерархия памяти**

* Регистры, расположены прямо в процессоре
* Кэш память для хранения часто используемых фрагментов памяти
* Основная оперативная память
* Дисковая память
* Память на магнитных лентах (стримеры)
* В старых ЭВМ активно использовались перфоленты и перфокарты

**Кэш память**

Кэш-память содержит копии наиболее часто используемых участков основной памяти и строится на быстрой статической памяти. Кэшируются не отдельные слова, а линейки из последовательности байтов основной памяти (скажем, в архитектуре x86 длина линейки составляет 64 байта)

Кэш - память является своеобразной аппаратно реализованной hash- таблицей, где hash образуется из адреса основной памяти отбрасыванием нескольких младших битов

Линейка, на которую адрес попал, переносится в начало списка, а линейки, на которые долго не было попадания, выбрасываются из списка.

**Часть 2 Основная память**

**Принципы работы основной памяти**

Статическая память на триггерах, быстрая, но дорогая.

Динамическая память на конденсаторах, медленнее, чем статическая, но и значительно дешевле. Поскольку идеальных диэлектриков не бывает, заряд с конденсатора постепенно стекает, поэтому нужно регулярно делать чтение и перезапись (refresh), что еще больше ее замедляет.

**Характеристики основной памяти**

Latency –задержка. Многим электронным устройствам, в том числе и памяти, нужно некоторое время разгона, пржде чем начать полезную работу

Cycle time – длительность такта (см. Глава 1. раздел «Цифровая логика»)

Bandwidth- пропускная способность, то есть ширина шины доступа процессор-память.

Пример: первые IBM PС XT базировались на микропроцессоре i8086, процессор и шина в котором были16-ти разрядные. Когда перешли на i8088 с 16-ти разрядным процессором, но c 8-ми разрядной шиной, в несколько раз уменьшили цену персонального компьютера.

**Часть 3. Виртуальная память**

Virtualis (лат.) означает «возможный», объект, который не существует, но может возникнуть.

Страница– фрагмент оперативной памяти фиксированного размера (скажем, 2 Кб).

Сегмент– фрагмент оперативной памяти, размер которого не фиксирован.

Математический адрес– пара ( N, D ), где N – номер страницы или сегмента, D – смещение нужного адреса от начала страницы.

Таблица страниц или сегментов – таблица, входом которой является номер страницы, а содержимым – адрес страницы в оперативной памяти, если она находится в этой памяти, или информация о ее месте на диске, если этой страницы нет в памяти.

Главный принцип виртуальной памяти – использование прямых физических адресов запрещено, только относительные математические.

Физический адрес – настоящий адрес внутри оперативной памяти.

**Реализация виртуальной памяти**

Перед каждым использованием математического адреса для обращения к памяти его нужно преобразовать в физический адрес.

По номеру страницы обращаемся к содержимому таблицы страниц.

Если страница присутствует в памяти, то просто добавляем к ее адресу начала смещение, если же нет – то вызывается процедура операционной системы по подкачке нужной страницы. Если свободной памяти для этой страницы нет, то приходится откачивать одну из страниц на диск.

**Способы ускорения виртуальной памяти**

Если нужной страницы не оказалось в оперативной памяти, говорить о быстродействии не приходится. Но оказывается, при работе любой программы быстро формируется рабочее множество страниц, к которым осуществляется 90-95% обращений.

Но даже если страница в памяти находится, преобразование «математический адрес в физический адрес» сильно замедляет работу.

TLB(Translation lookaside buffer) – специализированная кэш-память, где находятся адреса часто используемых страниц.

**Обработка ошибок в памяти**

Ошибки можно компенсировать только избыточной информацией.

Самый простой способ - к каждому байту информации добавить бит четности, если сумма по модулю 2 всех битов, включая дополнительный, равна 0, все в порядке, этот способ помогает только фиксировать ошибку, но никак не помогает в ее исправлении.

Код Хэмминга, например, на 16 разрядов можно добавить 5 контрольных разрядов (контрольные суммы в 1, 2, 4, 8, 16 позициях), тогда можно не только обнаружить одиночную ошибку, но и исправить ее.

Существуют и другие, более сложные коды.

**Часть 3. Организация взаимодействия устройств**

**Основы ввода-вывода**

В самых первых ЭВМ уже были устройства ввода-вывода. Нельзя использовать машину, которая не умеет ни писать, ни читать. В старые времена были перфокарты, которые, кстати, были придуманы Жаккардом в 1801 году для управления ткацкими станками, чтобы плести красивую ткань с разными рисунками. Эти станки до сих пор называются жаккардовыми. Перфокарта - это плотная картонка с дырочками. Ее кладут в устройство считывания, в котором есть электроконтакты (это сегодня, а у Жаккарда все было механическое), если между контактами проходит дырка, контакты соединяются и проходит сигнал, если между контактами проходит фрагмент перфокарты без дырки, то ничего не происходит. Перфокарты уже использовал Чарльз Бэббидж. Современный вид перфокарт был изобретён Германом Холлеритом в конце XIX века для переписи населения США на табуляторах. Это простое электромеханическое устройство, которые умеет подсчитывать количество перфокарт, у которых дырка оказалась в нужной позиции. Это очень удобно, и перепись населения провели очень быстро, а Холлерит создал компанию, которой сегодня уже больше ста лет. Я помню как в 96 году ХХ века праздновали 100 лет со дня создания компании TMC ([Tabulating Machine Company](https://ru.wikipedia.org/w/index.php?title=Tabulating_Machine_Company&action=edit&redlink=1" \o "Tabulating Machine Company (страница отсутствует))), которая в [1911](https://ru.wikipedia.org/wiki/1911) вошла в конгломерат [C-T-R](https://en.wikipedia.org/wiki/Computing_Tabulating_Recording_Company), а в 1924 году была переименована в IBM. Сегодня IBM ведет свою историю именно с 1911 года (и, естественно, еще раз отпраздновали столетие).

Затем появились перфоленты, после барабаны, магнитные диски, стримеры и так далее. Если машина хотела прочитать какие-то данные, процессор останавливался и спокойно ждал пока механическое устройство считывало данные. Это было крайне неэффективно, поэтому в середине шестидесятых годов появились специальные устройства, которые назвали каналами (каналы, как и многое другое, изобрели в IBM). Канал - это просто маленький процессор, со своей памятью, со своим управлением, который умеет управлять своим устройством ввода-вывода. Центральный процессор знает только одну команду ввода вывода SIO (Start INPUT/OUTPUT). Эта команда запускает работу канала, номер канала является параметром этой команды, и вот запускается какой-то канал, канал что-то делает, допустим, на диске устанавливает головки на нужный цилиндр на одной вертикали, ждет появления начала трека при обороте диска и считывает данные или пишет. Всё это делается совершенно независимо от центрального процессора, и канал сам всем управляет. Когда он заканчивает свою работу, выдается прерывание. Центральный процессор понимает, что канал работу закончил. Это было великое открытие. Пока канал работает, процессор может делать другие вычисления. Потери резко уменьшились, то есть каналы - это действительно большое достижение в нашей области, а именно в параллельной работе устройств ввода/вывода и центрального процессора. Конечно, никто не читает одно слово с диска или с перфоленты и не пишет одно слово - это было бы слишком не эффективно, совершенно не соразмерно со скоростью работы процессора. Сегодня соотношения могут быть 1000 000 и больше скорости работы процессора и устройства ввода/вывода. В общем сильно большая разница, поэтому стараются сделать работу поблочно. Данные группируются в блоки примерно по 3-4К слов, и канал за один раз читает целый блок или даже не один блок. Блок - это просто порция данных фиксированного размера.

Для вашей логической работы 4000 байтов может быть много. Тогда блок делится на записи. Запись - это единица равная, скажем, 8-10 словам, в зависимости от необходимости. Итак, блок состоит из многих записей, и люди работают с необходимыми записями. Я помню, что в OS/360 режим работы с блоками назывался BSAM, а с записями – QSAM.

Итак, канал **–** специализированная микроЭВМ, отвечающая за обмен данными с определенным устройством.

Блок данных **–** порция данных фиксированного размера.

Запись **–** единица обработки данных, обычно блок состоит из многих записей

Как уже говорилось, центральный процессор запускает операцию обмена, выполняя команду SIO с параметром – номером канала, после чего центральный процессор и канал работают параллельно

**Буферизация**

Следующее важное понятие, необходимое для обсуждения , - это буферизация. Пусть есть очень большой файл, который необходимо последовательно обработать, то есть прочитать или записать. Может случиться ситуация, когда канал прочитал блок, и мы его обработали, затем нужно читать следующий блок, тогда потери необратимы. Можно эту потерю демпфировать, например, сделать 2 или больше блоков, тогда задержка будет только в течение чтения самого первого блока файла, а потом вы этот блок начинаете обрабатывать, а в другой буфер читаете следующий блок. Вы же помните, что канал и центральный процессор работают параллельно. Всё это делается для того, чтобы центральный процессор все время был в работе, а не простаивал. Данный процесс и называется буферизацией.

На самом деле это не простое дело играть с размером блока и записи. Хорошо когда время обработки блока примерно равно времени его чтения, тогда всё будет хорошо, а если это не так, то вы всегда можете увеличить размер блока или уменьшить, смотря, что быстрее происходит, то есть в случае, если обработка очень быстрая, тогда блок необходимо делать небольшим, а если обработка очень медленная, то блок лучше делать побольше, чтобы пока читается блок вы завершили обработку предыдущего блока. Тут я могу часами рассказывать как мы играли с размерами блоков во времена ЕС ЭВМ, например, были диски (я помню, что размер трека диска в начале 70-х годов был 7600 байтов) Если задать размер блока 7600 байтов, то будут потери, потому что вы прочитали один блок и, чтобы увидеть специальный маркер, (позже мы разберем строение диска и увидим, что в начале трека есть специальный маркер, по которому канал понимает, откуда начинается трек и, что нулевой адрес трека находится именно здесь, но пока вы обрабатываете блок, диск это место проскочит, и у вас будет лишний оборот, чтобы снова схватить маркер. Мы стали экспериментировать, искать, какую нужно делать длину блока, чтобы лишних оборотов не было, поэтому я на всю жизнь запомнил, что если делать блок размером 7340 байтов, то без лишних оборотов пройдет следующее чтение. Это реально ускорило работу в 2 раза во многих программах. Сейчас я время от времени поручаю своим нынешним студентам курсовые работы, в которых они проверяют нашу теорию, всё то же самое, что и тогда, только цифры другие. Но, тем не мение, знание того, сколько занимает по времени чтение одного трека на сегодняшних дисках, все еще актуально. Мой студент Денис Смирнов установил, что минимальное время работы программы было при размере буфера в районе 4095. При размере блока 4096 время чтения подскакивает сразу в два раза по сравнению с 4095, то есть, иными словами, можно сказать, что за полвека практически ничего не изменилось кроме цифр. Изменился только масштаб, и сегодня необходимо знать как устроен трек, чтобы оптимально читать длинные файлы.

**Завершение операции ввода/вывода**

Есть два типа завершения операции ввода-вывода: программный - когда центральный процессор сам опрашивает определенный регистр в ожидании завершения операции. Иногда это действие называется сканированием, например, когда мы делали телефонные станции, а мы сделали больше десятка разных телефонных станций, то там это сплошь и рядом. Дело в том, что регистров, которые нужно опрашивать, несколько десятков тысяч, если каждый будет вызывать прерывание, то процессор будет прерываться слишком часто. Это крайне не удобно, поэтому делается по-другому: каждый регистр снабжается триггером, который принимает значение 0, состояние регистра если не изменилось, и 1, если изменилось, тогда каждые, скажем, 10 миллисекунд центральный процессор просто опрашивает эти регистры, если где-то появилась единичка, значит, процессор регистр читает, а канал просто выставляет эту единичку, когда слово изменилось. Подобное действие называется программируемым вводом/выводом - когда центральный процессор сам, когда хочет, по определенным правилам опрашивает устройства ввода/вывода.

А второй тип завершения операции ввода/вывода - это прерывание по завершении операции. Обработчик прерывания запускает следующую операцию обмена. Этот способ заметно более эффективен, но если у вас 1000 каналов - это не реально, поэтому диски и ленты работают по принципу прерывания, в устройствах типа телефонных станций и подобные действия в операционных системах программируются.

**Часть 4. Внешние накопители, физическая организация, диски и ленты**

**Внешние накопители**

На изображении (Рис. 3) вы увидите жесткий диск, по-другому говорят Винчестер. Винчестер - это несколько соединенных магнитных дисков, сидящих на одной оси, и между ними расположены головки, одна головка смотрит вверх, а другая вниз. Диск делится на треки по окружности от меньшего радиуса к центру. Несколько треков, которые обрабатываются при фиксированном положении головок, называются цилиндром.

Немного об истории происхождения термина Винчестер. Первый винчестер придумали, как и многое другое в нашей науке, в компании IBM. Они выпустили жесткие дисководы, которые пошли в продажу с названием 3030. В то время очень популярны были охотничьи винтовки –винчестеры, которые также назывались 3030. Инженеры начали называть свои диски как винтовку, в итоге это название закрепилось и уже более полувека мы так и говорим - Винчестер, хотя изначально это было просто шуточным названием. Один или несколько дисков сидят на одной оси, между этими дисками двигается лапа (рычаг перемещения головок), которая может стать близко к центру или далеко, на самый край диска. Каждая лапа имеет головки чтения/записи, смотрящие вверх и вниз на микронном расстояние от дисков, поэтому за счет небольших электрических токов могут писать и читать на диске. Прочитать или записать что-либо внутри цилиндра можно относительно быстро, как раньше это было на барабанах, а если есть необходимость перейти с цилиндра на цилиндр, то это уже намного медленнее, потому что механическое движение не может быть очень быстрым. Ни о какой скорости тут говорить не приходится.

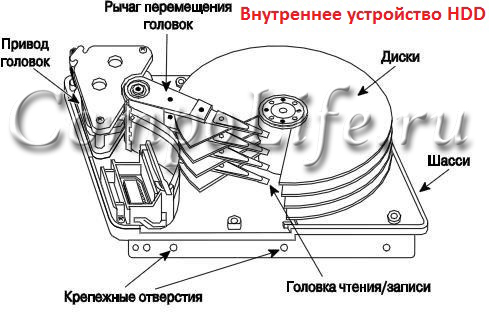


Рис. 3

На самом деле ленты появились раньше дисков, но они были слишком медленными, поэтому ими пользовались относительно мало, а затем появилась их реинкарнация, так называемые, стримеры. Это огромные ленточные накопители, изображённые на фотографии (Рис. 4), на них держат резервные копии дисков.

Подобные накопители используют в атомных исследованиях. Мне на всю жизнь заполнился случай в Протвино (город ученых под Москвой), где проводили эксперименты на ускорителях (в шестидесятых - начале семидесятых годов самый большой в мире ускоритель был у нас в Протвино). Результаты экспериментов записывались на больших лентах. Как-то одному ученому пришла в голову идея, что может существовать интересный физический эффект. Он взял штук 500 лент, которые были с записями предыдущих экспериментов и просто хранились на складе, поработал с ними и обнаружил этот эффект. Когда не знали об этом эффекте, то в данных на этих лентах ничего не видели, а когда поняли, что искать, то открыли новую частицу без единого физического эксперимента, только пересмотрев информацию на магнитных лентах, на которых были записаны результаты от предыдущих экспериментов. В то время я был еще совсем молодой, но на меня произвело огромное впечатление, что на лентах информации больше, чем мы думаем.

Сегодня стример всё-таки в большей степени устройство для резервных копий дисков, хотя, возможно, физики и по сей день используют стримеры в своих работах.



Рис. 4

В последние годы стали популярными CD диски Read only CD Rom (Рис. 5). Они предназначены только для чтения. Впервые их начали использовать для записи и воспроизведения музыкальных треков, затем их стали применять для хранения программ и данных. Не знаю, какая плотность у дисков сейчас, но во времена выпуска первых CD дисков на один CD Rom можно было записать 768 МГб данных. Затем появилась разновидность CD RW read and rewrite - разновидность компакт-диска (CD) для многократной записи информации.



Рис. 5

**Шины**

Внешние устройства соединяются с центральным процессором с помощью шин. Существует много разновидностей, их пытаются стандартизировать, чтобы можно было к одному процессору присоединять разные устройства. Одна из самых популярных шин - это SCSI (Small Computer System Interface). В основном диски соединяются с процессором с помощью шин SCSI.

Лично я в большинстве случаев пользуюсь интерфейсом USB, в принципе, как и вы. Это специальные разъемы в ноутбуках, в которые вставляются флешки или принтер и многое другое. Этот разъем считается универсальным. Переводится как Universal Serial Bus USB, модели 2.0 довольно быстрые.

Внутри машины чаще всего используется PCI-Express. На сегодняшний день элементы компьютера связаны между собой посредством шины PCI-Express. Этот разъем в современных машинах можно увидеть, сняв крышку компьютера. В слот (разъем) можно вставить плату ускорителя, и мы этим часто пользуемся в компании. Данный интерфейс считается очень быстрым, но он работает на очень небольшое расстояние, порядка 10 см, то есть вывести провод из шины нельзя. Частоты настолько высокие, что на длинных проводах шина не работает.

**Прямой доступ к памяти**

В первых ЭВМ любое обращение к памяти шло через центральный процессор, а потом стало понятно, что хорошо было бы сделать этот процесс независимым, поэтому появились устройства DMA (Direct Memory Access). Они организуют обмен данными между дисками и оперативной памятью, минуя центральный процессор, что позволило существенно ускорить обмен.

**Арбитраж**

Иногда возникают конфликты, например, водопровод обращается к памяти, и сам процессор в это же время обращается к памяти. Понятно, что необходимо дать приоритет процессору, так как он уже работает, а водопровод только впрок читает, ещё не известно, может и не пригодится из-за передачи управления. Для разрешения конфликтов была придумана электронная схема Арбитраж, которая вступает в силу, когда возникает конфликт, то есть когда несколько устройств обращаются в одно время к памяти, одновременно, в одном такте.

**Часть 5. RAID-архитектуры**

RAID- Redundant Array of Independent Disks, избыточный массив независимых дисков. То есть несколько дисков рассматриваются с точки зрения пользователя как один повышенной надежности. RAID массив - это один логический диск, хотя за ним могут стоять несколько физических дисков. RAID дисков бывает несколько типов.

RAID 1 (Рис. 6) - зеркальный дисковый массив, это два диска, один из которых является полной копией другого, то есть если один сломается, у нас есть информация на втором подобном диск. Такой тип считается очень дорогим за счет удвоения дисковой памяти.

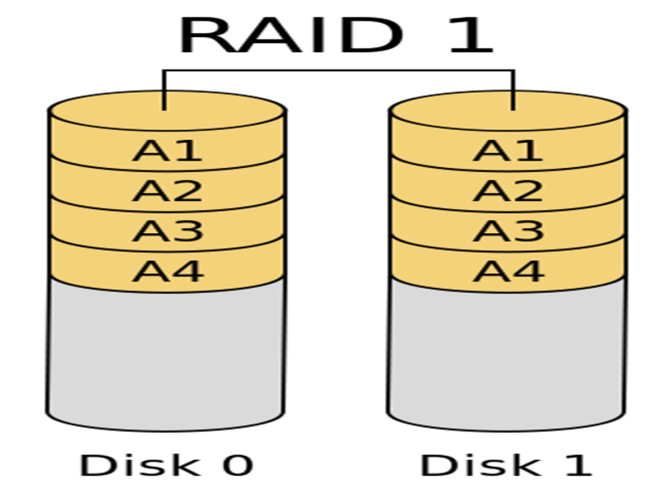
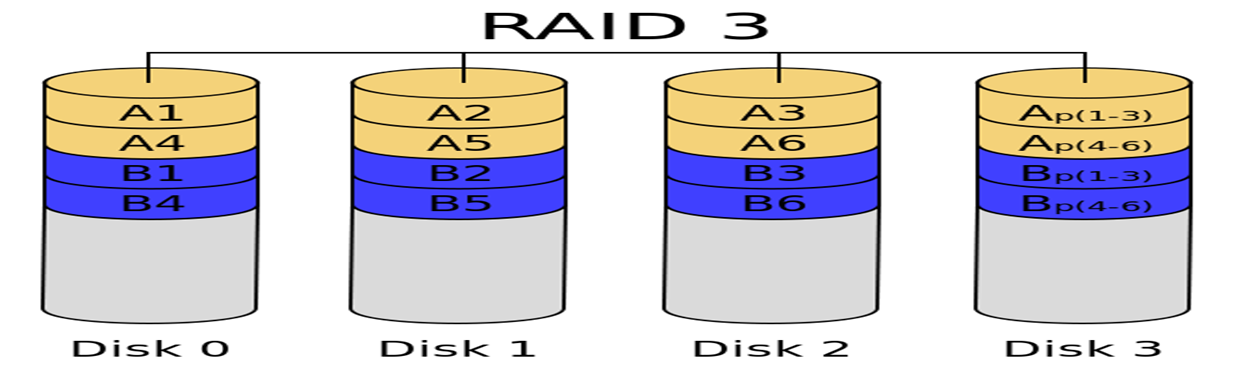


Рис. 6

RAID 2 — зарезервирован для массивов, которые применяют код Хемминга.

Разберем подробнее RAID 3. Данные хранятся на n-1 диске, причем разбиваются на куски размером меньше сектора диска. На одном диске хранятся блоки четности этих кусков, что дает возможность восстановить данные в случаях сбоев. Ранее применялся RAID-2, в котором на один диск данных использовалось n-1 дисков четности, что давало возможность исправления ошибочных фрагментов, но это решение было слишком затратным.

Рис. 7

Основным недостатком уровней RAID от 2-го до 4-го является невозможность производить параллельные операции записи, так как для хранения информации о чётности используется отдельный контрольный диск. RAID 5 не имеет этого недостатка. Блоки данных и контрольные суммы циклически записываются на все диски массива, нет асимметричности конфигурации дисков.

Контрольные суммы формируются с помощью операции XOR, что дает возможность восстановить информацию в случае сбоя одного из дисков. RAID 5 обеспечивает высокую скорость чтения — выигрыш достигается за счёт независимых потоков данных с нескольких дисков массива, которые могут обрабатываться параллельно.

**Глава 3.**

**Увеличение производительности устройств**

**Часть 1. RISC –Архитектура**

Давайте рассмотрим, как можно ускорять машины. Один из таких методов - это RISC архитектура - Reduced Instruction Set Computer. Эта архитектура была придумана в двух американских университетах.

Что это такое? Представьте себе начало 80-х годов, люди уже умели делать кристаллы, в которых располагалось до 100-150 тыс. транзисторов. Оказалось, что встроить в один такой кристалл реализацию 2 сотен команд никак не удается, так как кристалл слишком маленький, а в это время люди уже хорошо понимали, что всё, что происходит внутри кристаллов, делается быстро. Даже мы проводили такие эксперименты: если взять две ПЛИС, то внутри одной микроросхемы можно добиться частоты примерно 100 мегагерц, а если эту же работу разделить на 2 кристалла, поставить их рядом на плате и проводок сделать в 2 см, то частота падает до 20 мегагерц. Это чистая физика, так как внутри кристаллов толщина проводов 100 атомов меди, а тут больше 10000 - другое сопротивление, другая емкостная нагрузка, то есть размер в данном случае имеет значение. Так вот Patterson и Hennessy придумали такой прием - они создали процессор, в котором всего около 25 команд, эти 25 команд поместили в один кристалл, остальные команды уже программируют из нескольких таких команд. Что для этого требуется: необходимо уметь загружать в регистр процессора данные из памяти и обратно из регистра в память, нужно уметь делать плюс, минус, умножить, разделить, необходим также условный переход. Идея оказалась очень здравая: длина кода обычно увеличивается раза в два, то есть код в командах RISC обычно длинней, чем в традиционной архитектуре. Все команды, которые вошли в RISC - это часто используемые команды, остальные используются редко, поэтому удлинение не более, чем в 2 раза. То есть некоторые действия приходится прописывать в несколько команд в архитектуре RISC, но скорость получается в 5 раз быстрее. Эта идея оказалась очень популярной, сегодня практически все машины имеют RISC архитектуру. Другой момент заключается в том, что технология производства кристаллов сегодня сильно улучшилась – в кристалл можно поместить и 25 команд, и три сотни команд, тем не менее, основные принципы, придуманные авторами RISC архитектуры, сохранились и сегодня.

**Принципы RISC**

* Все команды имеют одинаковую длину – одно слово
* Все (или почти все) команды исполняются за один такт
* Много регистров (500-1000)
* Работа с памятью только в двух командах – Load и Store
* Программирование только на АЯВУ

**Водопровод**

Еще раз, с удовольствием, повторю, что схему опережающего чтения команд из памяти придумал и опубликовал в 1955 г. в США советский ученый С.А. Лебедев. Он назвал эту схему «водопроводом», так что термин « pipeline » является прямым переводом с русского, позже в англоязычной литературе стали использовать другой термин «prefetching».

Поскольку по статистике каждая 6-8 исполняемая команда является передачей управления, делать аппаратный буфер водопровода на большее число позиций нецелесообразно.

Если 20 лет назад память была в 4-5 раз медленнее процессора, то сегодня уже в 15-20 раз, поэтому актуальность водопровода только возрастает.

**Предсказывание переходов**

После любой команды передачи управления водопровод нужно заново разгонять, при этом процессор будет простаивать, поэтому в условных предложениях и циклах нужно запускать два водопровода (после then и else , начало цикла и команды после цикла).

Современные ЭВМ умеют собирать динамическую статистику переходов и на более вероятные ветви удлинять глубину водопровода, а на менее вероятные – укорачивать.

**Чередование ( interleaving)**

Интерливинг - это тоже одна из вещей, придуманная Лебедевым и опубликованная в Америке. Одно из наиболее частых и времяемких действий – это работа с массивами, при этом идет обращение к последовательным адресам памяти.

Представьте себе, что вы работаете с массивами. Если бы не было цикла и массивов, то все программы были бы очень короткие. Именно за счёт циклов и массивов программы могут работать часами. Что можно делать с массивами, например, посчитать сумму элементов, либо сделать скалярное умножение, и тогда хотелось бы, чтобы за одно обращение к памяти можно было прочитать несколько слов, и вот Лебедев был первым, кто это придумал, и назвал этот процесс чередованием памяти. Память делится на банки. Банка - это жаргонное слово, официальный термин – банк памяти, но все программисты используют именно жаргонное выражение. Каждая банка памяти имеет свою систему управления, что означает, что все банки памяти могут работать независимо друг от друга. Адресация идет следующим образом: нулевой адрес в первой банке памяти, адрес первого слова во второй банке, следующее (второе) слово в следующей банке памяти и т.д.

И тогда центральный процессор может заказать информацию сразу из 4 или более банок одновременно в одном такте, безусловно для этого требуется широкие шины. То есть в один такт можно записать или прочитать несколько элементов массива для векторных ЭВМ. Это сильно ускоряет работу, потому что есть возможность читать не одно, а 4, 8 или больше слов, в зависимость от того, сколько банок памяти есть.

**Часть 2. Конвейер инструкций**

**Введение в аппаратный параллелизм уровня машинных команд**

ILP Instruction-Level Parallelism

Мы уже говорили о водопроводе – своеобразном конвейере, читающем команды из памяти впрок.

Позже появились конвейеры исполнения команд, начиная с Intel 80486 (1989 г. )

Суперскалярное выполнение – если ЭВМ имеет несколько сумматоров, умножителей и других устройств, можно выполнять несколько команд параллельно, первым суперскалярным процессором стал Intel Pentium (1993г.)

Спекулятивное выполнение, например,

If a > 0 then x := y+z; здесь можно начать исполнение присваивания параллельно с проверкой условия, но собственно запись в x произойдет только тогда, когда станет известной истинность условия.

**Многопроцессорные и альтернативные архитектуры**

SIMD – single instruction, multiple data, несколько ЭВМ каждый такт выполняют одну команду, но со своими данными, одна выделенная ЭВМ раздает команды на исполнение.

Во-первых, нужно сказать кому это в принципе нужно. Есть задачи решения систем уравнений в частных производных. Есть метод сеток - это когда считаются параллельно значения функции в разных точках, потом шаг между точками уменьшается в 2 раза. Так вот, первой такой исторической SIMD машиной была ILLIAC IV.

Пример: ILLIAC IV(1975 г.) 64 компьютера. Планировали сделать 256 компьютеров, но довели до исполнения только 64 плюс 1 управляющая, у каждой из 64 компьютеров свой процессор, своя память - всё свое, но так устроено, что все они исполняют одну и ту же команду, но со своими локальными данными. Все, что может допустить каждый из этих компьютеров, он может одно действие пропустить, если он попал на границу области, то есть либо все исполняют одну команду, либо какие-то команды компьютеры пропускают.

Был советский аналог ПС 2000.

К архитектуре SIMD можно отнести векторные ЭВМ, например, CDC STAR-100, Cray-1, Fujitsu AP1000 и AP3000, была оригинальная советская векторная ЭВМ ПС 3000 - эта машина была действительно оригинальная. Ее очень любили геологи.

MIMD - multiple instruction, multiple data

Пример: транспьютер фирмы Inmos (1985 г.) микропроцессор с 4-мя каналами, по которым он может соединяться с такими же микропроцессорами, для транспьютеров был придуман специальный язык параллельного программирования ОККАМ.

Пример : Intel Xeon Phi(2012 г.)

VLIW - very long instruction word (1980 г.), одна длинная машинная команда содержит в себе много инструкций. Это существенно компиляторный подход в отличие от аппаратного параллелизма – только оптимизирующий компилятор может заполнить все или почти все позиции команды. Представьте себе, что у вас шкала на 15 команд, допустим, 7 загрузок и выгрузок и 8 арифметических команд, если вам удается составить программу таким образом, что в этих длинных командах все позиции заняты, понятно, что абсолютно все занять тяжело, но пускай будут заняты 70-80% позиций - это уже будет хорошо, то тогда машина начинает работать очень быстро, поскольку подобным образом вы заставляете машину работать параллельно, выполнять несколько команд одновременно. Понятно, что никакие машины динамической архитектуры такого сделать не могут, а вот хороший компилятор такое сделать может. Дело в том, что если вам надо сделать программу, которая будет, например, управлять космической станцией, которая летит на Марс 3 года, чего стоит затратить на трансляцию не минуту, не час, а трое суток. Пусть ваш транслятор покрутит, поработает, найдет самый оптимальный вариант заполнения и выполнения всех команд, будет там представлять команды, соблюдать корректность преобразования, зато эта программа будет работать три года в космическом корабле и выполнять всё будет быстро-быстро.

Я лично знаю в России людей, которые делали Эльбрус. Эльбрус сначала делали, как машину с аппаратным параллелизмом, и один из ее главных конструкторов Б.А.Бабаян говорил: “Кто не верит в динамизм, тот не верит в коммунизм!”. Я еще в 1974 г. с ним спорил, что так нельзя, что это должен делать компилятор, а не аппаратура. Ну вот, я оказался прав. Сейчас все новые модели Эльбрус имеют архитектуру VLIW. Периодически я встречаю Бабаяна и спрашиваю, как он сейчас относится к коммунизму.

VLIW сегодня одна из наиболее применяемых архитектур. Например, те персоналки, которые стоят у вас на столах, лежат на коленях - практически все VLIW-ие, то есть старая система команд x86 интерпретируется некой VLIW-ой аппаратурой.

Первые компьютеры с такой архитектурой – это Cydrome, MultiFlow, DSP C6000, Эльбрус2000, Эльбрус S

 Рис. 8

При всех достоинствах VLIW по сравнению с аппаратной реализацией, возникли и проблемы, например, синхронизация кэша для разных действий внутри одной длинной команды. Для решения этих проблем в 1997 г. фирмы HP и Intel предложили развитие архитектуры VLIW.

EPIC- explicitly parallel instruction computing. Первый пример Itanium фирмы Intel. Здесь сочетание VLIW архитектуры с аппаратной поддержкой.

Предикация – команды из разных веток условного предложения запускаются параллельно, но со специальными полями условий. Например, в архитектуре ARM (последний вопрос нашего курса) все команды построены по типу предикации.

Спекулятивная загрузка в регистры из памяти, команды Load запускаются существенно раньше, но перед реальным использованием регистра проверяется его готовность.

**Часть 3. Нестандартные архитектуры**

**Систолические архитектуры**

Ранее под термином систолические архитектуры понимали бензольные кольца. Из химии бензольное кольцо – это когда 6 атомов углерода соединены в определённую структуру. При моделировании химических реакций оказалось удобно, когда структура вычислительной сети подобна структуре молекул моделируемого вещества. С течением времени это как-то сильно обобщилось. Что такое сегодня систолическая архитектура - это несколько аппаратных конвейеров, состоящих из небольших вычислительных элементов (систол), «заточенных» под конкретную задачу. Все систолы работают от единого тактового генератора. Систолические архитектуры применяются для задач массовой обработки данных (обработка сигналов, машинная графика, биоинформатика, базы данных). Представьте, что надо выполнить сложное действие, например, найти решение на дереве, и сразу запускается несколько веток. Надо сделать так, чтобы все ветки имели примерно одинаковую длину и тогда можно чисто аппаратно запустить всё сразу.

Современные систолические решения основаны на парадигмах SIMD/MIMD, являются репрограмируемыми и реконфигурируемыми, то есть чаще всего реализуются на ПЛИС.

**Архитектуры с общей памятью**

Пусть есть N процессоров, работающих с одной общей памятью (Рис. 9). Реального ускорения в N раз добиться невозможно из-за конфликтов обращений к общей памяти. Ситуацию несколько сглаживает наличие у каждого процессора большой кэш-памяти. Тогда большую часть времени процессор будет работать со своим активным кэшом и только изредка обращаться в основную память. Поскольку частота обращения в основную память во много раз уменьшается, то и вероятность конфликтов также сокращается. По своему опыту знаю, что больше, чем на 6 процессоров с большими кэшами, такая система работать не будет. То есть добавление 7-го, 8-го процессора практически не даст ускорения.

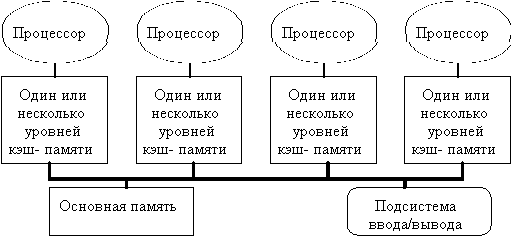


Рис. 9

**Обеспечение соответствия кэша и памяти**

Кэш (Cache) – относительно небольшая быстрая память для часто используемых данных и команд.

Белка в лесу собирает орехи, прячет их в маленьких норках. Норки, в которых она прячет орехи и называются кэшем. Точно такой же принцип и в вычислительной технике.

То есть вы делаете быструю память, куда кладете часто используемые данные и команды. Если получается настроить машину так, чтобы все часто используемые данные хранились в кэше, а не в основной памяти, то можно добиться существенного ускорения. Рассмотрим мой любимый пример. У меня есть приятель - профессор Борис Яковлевич Штейнберг из Южного Федерального Университета, его студенты увлекаются распараллеливанием и участвуют в международных соревнованиях. Есть такое соревнование: берётся персональная машина, делают умножение матриц и дают задание - кто сможет на этой же машине, при этом же тактовом генераторе, на той же самой аппаратуре выиграть по времени счёта. Был достигнут мировой рекорд 62, то есть можно выиграть в 62 раза на одной и той же машине, изменяя алгоритм расчета. И вот студенты Штейнберга достигли 62.5 раза!

Давайте разберемся, за счёт чего это достигается - матрицы, которые надо умножать, разбивают на блоки, так вот, надо сделать так, чтобы все эти блоки оседали в кэшах. То есть нужно знать как работает машина: каков объём кэша, какие правила, конфликты, какая шина. Студенты Штейнберга в совершенстве освоили эту науку, и вы тоже сможете это сделать.

Кэш состоит из набора записей (обычно несколько десятков слов), являющихся копией соответствующих слов основной памяти.

*Кэш – это hash -таблица,* входом в которую является адрес без нескольких младших битов. Если какой-то элемент данных нашелся в кэше, он помещается в начало списка элементов с одинаковым hash . Редко используемые данные выпадают из кэша.

**Часть 4. Современные архитектуры**

**Мобильные устройства**

Современный мобильный телефон – это мощная ЭВМ с огромной памятью и высокой производительностью.

Главные преимущества мобильного устройства (по сравнению с персональной ЭВМ) –  *всегда включен, всегда в кармане, всегда в сети!*

Сегодня многие пользователи предпочитают искать информацию в интернете на мобильном устройстве, а не с помощью персональной ЭВМ.

Тем не менее у мобильных устройств есть и принципиальные ограничения, связанные с форм-фактором:

* Маленькие размеры клавиатуры
* Маленький экран
* Часто протоколы связи с базовой станцией (GPRS, EDGE, 3G) имеют существенно меньшую пропускную способность по сравнению с локальными сетями ЭВМ.

**Встроенные системы**

Встроенная система – это программно-аппаратная система, в которой одна или несколько ЭВМ управляют каким-то специализированным оборудованием (цифровая телефонная станция, ракета, сердечный стимулятор и т.д.). Я в 1991 году защитил докторскую диссертацию по технологии программирования встроенных систем реального времени. В то время это считалась одной из самых сложных задач в программировании, и за прошедшие 25 лет мало что изменилось.

Часто вместо термина «встроенные системы» говорят «системы реального времени» (СРВ), поскольку проблемы и их решения этих систем действительно очень схожи. Многие авторы признают программирование СРВ самой трудной задачей нашей специальности по следующим причинам:

1.Необходимость укладываться в жёсткие временные ограничения, вне рамок которых важная информация просто пропадает.

2.Обычно СРВ управляются не одной ЭВМ, а многими, соединёнными в сложноустроенные сети, поэтому в СРВ входит вся многообразная тематика сетей (протоколы, устойчивость к сбоям и отказам, реконфигурация, пропускная способность и т.д.). Приведу еще один пример – в восьмидесятые годы мы делали телефонную станцию для правительственной связи, по которой говорили лидеры нашей страны. Брежнева я не застал, а Горбачёва застал! В то время техника в правительственной связи была только на советских микросхемах, в те годы популярна была шутка, что советские микросхемы - самые большие микросхемы в мире. Требования правительственной связи заключались в том, что всё необходимо было резервировать, дублировать, в случае сбоя переходить сразу в резерв. Но иногда станция ломалась: не соединяла, не идет гудок, трубка молчит. На этот случай у нас был жаргонный термин “подванивала”, то есть она то потухнет, то погаснет, как в известном анекдоте, то есть то работает, то не работает. И как отличить в такой ситуации сбой от отказа? Либо это случайный сбой, который можно повторить и все будет работать, либо это отказ, когда необходимо блоки вынимать и все менять. Так вот, анализ на отличие сбоя от отказа на нашей телефоной станциии занимал полторы секунды. Сотрудники КГБ страшно ругались, потому что было немыслимо, чтобы Горбачёв остался без связи! Как Вы видите в этой области есть реальные научные проблемы, которые мы тогда, разумеется, преодолевали.

3.Любая цифровая телефонная станция включает в себя большие базы данных (абоненты, блоки аппаратуры, учёт трафика и т.д.), поэтому и проблематика БД, сама по себе весьма не простая, в полном объёме включается в СРВ.

4.Часто (особенно для больших станций) станцией управляет один или несколько операторов, поэтому все проблемы человеко-машинных интерфейсов также приходится решать в процессе проектирования цифровых телефонных станций. Я на всю жизнь запомнил, что красный цвет - это цвет тревоги, а зелёный - это успокаивающий. Все аварийные ситуации сигнализируются оператору красным цветом, а все остальные зеленым.

5.Самая большая трудность при отладке СРВ заключается в неповторяемости ситуаций. Цифровая телефонная станция может успешно работать 2 года, а потом вдруг «вылететь», причём понять причину отказа очень трудно.

**Направления развития архитектуры процессоров**

Долгие годы соблюдался закон Мура «каждые полтора года производительность процессоров удваивается». Достигалось это за счет увеличения количества вентилей в кристалле, уменьшения размеров вентилей и шага между соединительными линиями. Сегодня лучшие фабрики по производству кристаллов работают с шагом 18-22 нм, но при шаге меньше 10 нм уже будут наблюдаться квантовые эффекты, когда вместо значений 0 и 1 придется рассматривать континуум непрерывных значений от 0 до 1

Кардинальное направление развития – параллельность (много процессоров, много ядер в одном кристалле, много блоков памяти). Уже сейчас есть 64-ядерные процессоры, в ближайшее время появятся процессоры с тысячами ядер, но здесь инженеры существенно опережают программистов-технологов.

Есть технологии MPI , OpenMP и некоторые другие, но все они далеки от совершенства. Создать параллельную программу по-прежнему очень трудно.

**Глава 4.**

**Архитектуры HLL на примере организации УВК Самсон**

**Часть 1. Почему мы решили создать свою ЭВМ**

Я уже много раз вспоминал, как я начал работать с военными. В те годы оборонная промышленность испытывала большие трудности, поскольку им приходилось переходить с аналоговой техники на цифровую. Никто в то время этого не умел делать. Именно поэтому оборонный отдел Обкома партии Ленинграда (всемогущая организация) обратился в Университет с официальной просьбой помочь преодолеть возникшие трудности. Тогда ректором ЛГУ (Ленинградский Государственный Университет) был Алесковский Валентин Борисович, а я был одним из самых молодых руководителей лаборатории системного программирования в Университете. Помню, вызвал меня как-то раз к себе в кабинет ректор, и вот я прихожу к нему, первый раз в жизни, а у него за столом сидят человек 20-25, большинство из которых одеты в военную форму, меня им представили, а мне их нет. Мне просто сказали, что ты будешь с ними работать. На мой вопрос кто эти люди, ответили, что они военные, я спрашиваю можно ли мне подумать, а мне ректор ответил, что ты сюда вошел как руководитель лаборатории, а можешь выйти вовсе и не сотрудником Университета. Конечно, мой ответ был “Хорошо. Всё понял.”, и я перестал задавать ненужные вопросы. Около года ушло на раскачку, знакомство, я стал инструктором оборонного дела Обкома партии, наверняка я был единственным беспартийным инструктором в Смольном. Моей задачей было внедрение цифровой вычислительной техники в оборонную промышленность и военные организации. Я мог по своему пропуску в Смольный входить в любое оборонное предприятие, тогда было 9 оборонных министерств, просто показав пропуск-вездеход. Меня везде пускали, принимали, я читал лекции старшему поколению, директорам крупных заводов, типа Кировского завода. Работать мы начали ассенизаторами, они работали на тумблерах, перфокартах, то есть на старых технологиях и на плохих, но секретных ЭВМ, а мы их переучивали. Одно из самых ярких моих воспоминаний - это спасение самого крупного заказа КГБ Кавказ 1 – станция правительственной междугородней связи. Было сделано 27 станций, потрачено много бюджетных денег, но станция не то, чтобы ломалась, а то работала, то не работала, поэтому сдать этот заказ было совершенно невозможно, так как в случае сбоя станции во время разговора Горбачева, полетело бы много голов.

Я хорошо помню большую миллиметровку, метр на метр, на ней цветными карандашами расписаны, где какой модуль сидит. Если вставили в модуль несколько команд, то вам надо делать какую-то заплатку, чтобы передать управление туда, потом обратно, в результате весь Кавказ1 был “заплатка на заплатке, заплаткой погоняем”. Естественно такую плохо структурированную программу им было не отладить. И вот мы наладили использование ЕС ЭВМ в качестве инструментальной, кросс трансляцию, сделали редактор связей, чтобы можно было автоматически модули состыковывать, а не разрисовывать, сделали систему документирования, так как они печатали все документы на машинке, а это означало, что любое исправление влечет за собой много перепечатывания. При всем при этом, мы прекрасно понимали, что мы наводим минимальную структуру производства, но принципиального рывка в производительности программистов и надежности не будет, потому что нужны языки высокого уровня.

Нас все воспринимали «в штыки», так как мы внедряли новшества. Например, на вопрос “Какая программа лучше: которая короткая или которая правильно работает?”, мне всё время отвечали что, конечно, короткая, потому что, если специализированная машина имеет 64 Кб памяти и если программа и данные будут длиннее на один байт, то программа работать не сможет. Мне это было очень больно слушать, я спорил, но меня никто не хотел слушать.

Сначала я как-то стеснялся своей работы, думал, что люди вообще перестанут со мной здороваться, потому что в советские времена партийных сотрудников не очень любили, но так как мы занимались делом, а не «пыль в глаза пускали», то почти никто из друзей от меня не отвернулся. Конечно, были люди, которые на меня пальцем показывали “человек работает в Смольном”, подобное всегда происходит, и к этому надо относиться спокойно. С другой стороны, у меня была определенная власть, я мог позволить себе довольно жестко говорить с руководителями других предприятий, а если требовалось, то привлекал начальника оборонного отдела Обкома партии. В общем, с муками, корчами мы в этом деле продвигались, и решили силой заставить их перейти на языки высокого уровня, потому что мы свято верили, что только это даст существенные сдвиги в повышении производительности труда и надежности программирования. Мы разработали язык - подмножество языка Алгол 68, ориентированный на задачи коммутации и назвали его А68К.

Начали делать кросс-трансляторы (транслятор, работающий на одной ЭВМ, но генерирующий код другой ЭВМ). За 5 лет мы разработали более 20-ти кросс-трансляторов для специализированных ЭВМ УК 1010, СУВК СС, СУВК СМ, НЕВА и др. НЕВА была сделана в Киеве - одна из самых отвратительных машин, но была названа таким красивым словом. Мы занимались внедрением трансляторов, обучением людей, страдали, конечно, страшно, потому что машины были очень странные. Не могу сказать, что они были плохие - они выдерживали тряску, езду по лесной дороге, броски питания, механические удары, то есть, с точки зрения инженерии, это были хорошие надежные устройства, но, с точки зрения программистов, на них программирование было невозможно, а транслятор сделать, чтобы он эффективно работал - просто никаких сил не хватало! И вот мы решили, что «спасение утопающих дело рук самих утопающих», поэтому приняли решение сделать свою машину. Думали, что математики могут сделать для себя машину лучше, чем инженеры для математиков.

**HLL компьютеры**

В то время мы уже знали термин HLL – High Level Language Computer (компьютер, ориентированный на алгоритмические языки высокого уровня). Эти компьютеры аппаратно поддерживают основные конструкции языков программирования, поэтому трансляция в их коды существенно упрощается.

Существовавшие на середину 80-х гг. HLL компьютеры (МИР, Burroughs, Эльбрус) были существенно дороже аналогичных по мощности традиционных компьютеров. Все HLL машины этого времени были дорогие, многие требовали водяного охлаждения - это было страшно неудобно. Кстати, мы считаем, что первая HLL машина была сделана в СССР, под руководством академика В.М.Глушкова - МИР. Она была создана для работы с текстами, построена в Киеве. В Америке сделали машину, которая аппаратно поддерживала язык Алгол 60, в том числе, рекурсивные процедуры, тегированную память - Burroughs. Чуть позже наши сделали Эльбрус, но злые языки говорили Эль-Burroughs, потому что некоторые идеи были похожи на Burroughs, ну, конечно, это были злые языки. Машина была абсолютно оригинальная, в неё были заложены авторские идеи, патенты. Лично я Эльбрус очень любил, у нас было принято считать, что Эльбрус - это своя машина, сделанная москвичами, а ведь в это время все занимались копированием западных ЭВМ. Наша лаборатория стала работать с авторами Эльбруса с 1974 года. Мы сделали интерпретатор Автокода Эльбруса на ЕС ЭВМ (как всегда, разработка аппаратуры задерживалась, а программное обеспечение нужно было разрабатывать), весь мат-мех ЛГУ разрабатывал пакеты прикладных программ для Эльбруса.

И с этих разработок, кстати, мы начали богатеть. Конечно, во времена социализма, деньги заработать было нельзя, но как я уже говорил, машины поставлялись по фондам ЦK КПСС. Например, кто-нибудь из наших сотрудников едет в какой-нибудь штаб военно-морских сил с целью поставить автокод Эльбрус, научить людей работать. И вот он сидит там месяц, а возвращается с фондом на следующую машину. В результате вычислительный центр лаборатории Системного программирования стал богаче, чем Вычислительный Центр всего Ленинградского Университета, конечно же, нам завидовали, пальцами на нас показывали, писали на меня бумаги, но мы же зарабатывали всё сами «своим хребтом».

Затем случилось несколько грустных событий. Так как я являлся членом Государственной комиссии по приемке Эльбруса 2 я лично видел, как пришел Леонид Райков, начальник отделения НИЦЭВТ и поставил пример умножения больших матриц и сравнил с ЕС 1060, у которой заявленная паспортная скорость была 6 млн, то есть медленнее, чем у Эльбруса в 2 раза (по паспорту Эльбрус 2 имел 10 процессоров по 12 млн операций в секунду), а умножение матриц было в несколько раз быстрее, чем на Эльбрусе. То есть на реальных задачах наша машина уступала машине, которая была копией американской машины. Для нас это был очень болезненный момент.

Мат-меху в награду за наши труды выписали фонд на получение машины Эльбрус 2. Мы выкопали гигантский бетонный пруд, который до сих пор сохранился, для охлаждения воды для Эльбруса, подготовили специальное помещения в 500 квадратных метров, уже начали получать оборудование, когда наши инженеры, которые должны были эксплуатировать Эльбрус, решили съездить в Москву на завод Счетно-Аналитических Машин САМ, где Эльбрус выпускался. По возвращении они заявили нашему директору Самосюку Георгию Петровичу, что не могут принять Эльбрус, потому что для его эксплуатации необходимо иметь целый полк солдат: держать помещение в чистоте, соблюдать влажность воздуха (примерно от 30 до 50%, ни больше, ни меньше). Если влажность меньше, то проскакивают искры и машина вылетает, если больше, то возникает конденсат, и капает капелька воды на раскаленную плату, и опять происходит сбой. Я до сих пор машину Эльбрус привожу своим студентам как пример, когда машину делают за пределами технологических возможностей. То есть они выжимали из машины скорость, которую никто в СССР повторить не мог, из-за этого они принимали запредельные технические решения, как когда-то было сделано с машиной БЭСМ, про которую я вам рассказывал ранее. Мы были вынуждены отказаться от этого фонда, и по этому поводу у меня «на сердце осталась рана». Ну мы же люди конструктивные и не будем просто так стонать из подворотни: “Ах плохо мне, плохо!». «А кому нынче хорошо?”, как в известном анекдоте.

Мы начали думать, почему так происходит. Попутно изучили ещё один отрицательный рекорд серии кристаллов, ориентированной на реализацию языка Ада, iAPX432 – производительность была в 100 раз хуже, чем могла бы быть при такой же технологии. В книге Г. Майерса «Архитектура современных ЭВМ» (Москва, МИР, 1985г.) были подробно описаны архитектура и система команд iAPX432. Мы всё это тщательно изучили и, с удивлением, обнаружили, что производительность кристаллов получилась много хуже, чем могла бы быть при такой базовой технологии производства кристаллов. Мы вновь стали разбираться в чём дело. Понимаете, там 7 типов адресации, причём, они иерархически устроены: прямая, косвенная, константа, непосредственный операнд с индексом, без индекса. Если, например, выполняется *a*:=*b*+*c* в цикле, каждый раз проверяется нужный тип адресации и каждый раз аппаратура всё проверяет заново. А ведь всё это можно сделать на этапе трансляции, зачем этим заниматься во время счета? То есть каждый раз аппаратура проверяет, какое действие выполняется – безусловно, это вызывает большие потери. Если вспомнить, что к этим потерям плюсуется также водяное охлаждение машины Эльбрус и ей подобных, а также плюсуется отрицательный рекорд a iAPX432 - всё это было неправильно, и мы поняли, что нужна какая-то свежая идея.

**Часть 2. Архитектуры HLL на примере организации УВК Самсон**

В 1984 году мы начали искать решение, например, попытались сделать специализированную машину. В то время мы занимались коммутацией, поэтому решили сделать машину под названием «Коммутация». На телефонной станции собственно коммутация - это фиксированная точка, логика, а управление сетями - плавающая запятая, это расчеты надежности, работа с операторами - это работа со строками, с дисплеем - то есть везде абсолютно разная математика. Например, телефонная станция «Броня» на 1 тысячу абонентов Министерства обороны Российской Федерации имела три ЭВМ, три разных комплекта инженеров, три разных комплекта ЗИП, разные операционные системы - это тоже было нелогично, и подобного аналога мы себе не хотели. И вот мы думали, но ничего так и не придумали! Тогда, впервые в моей жизни от меня ушли 2 разработчика, перед этим долго с нами работавшие. Стоит заметить, что я на тот момент руководил лабораторией с 1976 года, а эти 2 ушли, сказав, что «Андрей, ты ведёшь нас в тупик. Это решение невозможно. Мы не хотим изобретать невозможное. Воду в ступе толочь» и так далее.

А идея появилась, правда, не сразу. Основная идея управляющего вычислительного комплекса (УВК) Самсон (мы назвали свою машину не столько в честь библейского героя, сколько в честь самого большого фонтана в Петергофе, где мы живем и работаем) может быть выражена простой схемой:

*Программа на АЯВУ -> компилятор -> УВК -> результат*

Причем никаких обходных путей нет.

То есть любая программа должна пройти путь от текста на алгоритмическом языке высокого уровня, до результата, сначала через компилятор, потом на аппаратуре.

И компилятор, и УВК разрабатывались в одном коллективе, поэтому всегда можно было договориться, что если что-то проверено компилятором, то УВК эти проверки не повторяет.

Поскольку мы хотели большинство проверок выполнить во время компиляции, мы решили ограничить множество входных языков: Алгол 68, АДА, Модула 2 и другие языки статического типа.

Например, в ЕС ЭВМ есть 16 типов прерываний (неправильный код операции, неправильная адресация, неправильный номер регистра и так далее). А в машине Самсон таких аппаратных проверок нет. Мы считаем, что компилятор не может ошибиться, и код операции всегда правильный, всегда правильная адресация, всегда правильный номер регистра и так далее, поэтому заново всё проверять компьютору нет необходимости.

Немного лирики. Мы получили самую первую среди открытых организаций в СССР ЕС 1030 на наш мат-мех (то есть до этого момента машины подобного типа поставляли только военным), опять-таки за то, что мы перед этим 4 года работали на оригинальной машине IBM 360 модели 50 и хорошо изучили ее операционную систему. Их (IBM 360) было всего 2 штуки в СССР. Ходили слухи, что купили их чуть ли не через фашистскую в то время Южную Африку, а ведь СССР в то время было запрещено покупать подобную технику, но их установили!

В то время мы считались редкой организацией, которая знала IBM 360 и имела большой опыт работы на ней, к нам ходили толпы людей - задавали вопросы. Все произошло в 1974 году, когда ЕС 1030 была еще очень сырая. Когда мы работали в Москве на американских машинах, к ним постепенно подключали советские блоки. Поскольку нам давали машинное время только ночью (”Спорт любят только москвичи!»), никто за нами не следил, и мы всегда начинали работу с того, что переключали принтер, диски и ленты с советских на американские, а потом спокойно всю ночь работали. А уже дома, на мат-мехе у нас произошел один случай, который вошел в историю, когда работа машины ЕС1030 вылетела чуть ли не на месяц. Договоры, деньги - всё горит! А потом, одна сотрудница, (та самая, кто, кстати, отказалась в свое время принимать Эльбрус в Университет) обнаружила, что сбой заключается в схеме контроля. То есть она просто оторвала схему контроля от машины, и все стало работать.

Как математики мы стали исследовать этот вопрос и пришли к решению, что нужна схема контроля, а дальше нужна схема контроля схемы контроля, и так далее до бесконечности. А контроль - это большой объем оборудования. Из-за этого случая мы сильно пострадали, поэтому одна из первых идей Самсона заключалась в том, чтобы распределить обязанности между компилятором и оборудованием – то, что компилятор проверяет и генерирует, УВК не перепроверяет, но для качественной проверки необходимо знать типы обрабатываемых значений в каждой точке программы, а это возможно не для всех входных языков, в некоторых языках можно было написать процедуру с тремя параметрами, а вызвать с четырьмя или двумя, и никто слова не скажет. Сейчас некоторые варианты трансляторов какие-то проверки делают, но далеко не все. Уже в то время мы понимали, что такое статический язык, то есть Алгол 68, Ада, Модула 2 и Паскаль, в его лучших реализациях. В программах на этих языках трансляторы всегда знают в каждой точке программы типы обрабатываемых значений. За счёт этого мы можем выполнить многие проверки. Мы решили, что для Самсона будем писать программы только на статических языках.

**Разделение обязанностей между компилятором и УВК Самсон**

Наличие в УВК команд типа «вызов процедуры», «вырезка элемента массива», «цикл любой самой сложной природы» резко упрощает компилятор, поэтому мы решили, что Самсон будет строиться на принципах HLL. Мне многие говорили, что мы сделали машину под Алгол 68, но это не так. В любом современном языке программирования есть цикл, вырезка, вызов процедуры, условное предложение, присваивание, формулы, таким образом, мы аппаратно поддерживаем эти базовые конструкции многих языков программирования, а не конструкции одного конкретного языка. Вторая особенность - это ортогональность. Нам страшно не нравилось, что в ЕС ЭВМ есть команды сложения, вычитания и умножения полуслов (AH, SH и MH), а команды деления (DH) нет. То есть, когда ты программируешь формулы, то вместо команды DH ты должен писать подпрограмму, иными словами, «стоять на голове».

Поэтому мы приняли решение, что наша система будет абсолютно ортогональной, то есть если у нас было четыре типа адресации и 13 действий, то в УВК будет 52 команды для того, чтобы трансляция была простой. Когда мне кто-то из молодых сотрудников говорил, что какая-то команда почти никогда не используется и, возможно, не стоит ее включать в систему команд, то я всегда отвечал: «Без проблем, только транслятор в этом месте будешь реализовывать ты». На этом все вопросы заканчивались.

**Стековая архитектура УВК Самсон**

Как любая HLL машина Самсон - это стековая машина, то есть действие выполняется на стеках – стек целых 16 позиций по 16 разрядов, стек вещественных, всего 8 позиций по 32 разряда и стек адресов 16 позиций по 24 разряда. У нас максимальная память была 16Мб – тогда это было очень много.

В Самсоне аппаратных проверок переполнения и исчезновения стеков нет, но иногда какой-нибудь умный студент мат-меха говорит: «Профессор, как же так?! А если будет рекурсивная процедура? Вы же не можете предугадать глубину «дыхания» стеков внутри рекурсии?» Всегда можно посчитать глубину «дыхания» стеков внутри процудуры, но, поскольку бывает рекурсия, там может быть переполнение стека, поэтому я уточняю, что есть единственное исключение - при вызове процедуры необходимо следить за переполнением стека. Когда вызывается процедура, то должно быть гарантировано, что позиций стеков хватит, но эта проверка нужна для одной команды, а не для всех, как это сделано в других машинах.

**Работа с памятью**

Во времена проектирования Самсона мы спорили на тему, какую надо делать структуру памяти - сегментную или страничную. Страничная организация, конечно, проще, потому что все страницы одинаковой длины, из-за этого проще работать, но возникает такое страшное понятие, как внутренняя фрагментация. Например, вам нужно 100 байтов, а страница имеет размер 4096 байтов, так вот методов использования оставшихся байтов страницы нет, их и до сих пор нет, а в сегментах с переменной длиной проблема внутренней фрагментации не возникает, но там возникает внешняя фрагментация. В сегментной памяти каждый раз захватывается только нужное количество байтов, но при освобождении сегментов возникают «дырки» переменной длины.

Возникает вопрос, как бороться с потерей памяти между сегментами? Так вот - с внешней фрагментацией мы бороться можем. Этот метод называется методом граничных признаков - это когда сегмент ограничивается по одному слову с каждой стороны, в котором хранится длина, и можно поставить условие, что если знак длины - плюс, то сегмент занят, если минус, то сегмент свободен, благодаря этому при освобождении сегмента можно «склеить» его с соседними сегментами, если они тоже свободны. Поэтому мы остановились на сегментной организации. Таким образом, вся память разбита на сегменты, каждый процесс представлен двумя сегментами - сегмент кода и сегмент данных. В сегменты кода никто не пишет, поэтому параллельные процессы могут исполняться на базе одного и того же экземпляра кода. Вы можете использовать один сегмент кода и 100 сегментов данных для 100 параллельных процессов, и каждый сегмент данных будет ссылаться на один и тот же сегмент кода. Таким образом, представителем процесса является не его код, а его сегмент данных. Сегмент кода - это что-то вторичное, которое используется через сегмент данных. Для каждого процесса заводится большой сегмент данных, у нем будет содержаться стек статик процедур. Статика процедуры - это память для локальных данных и рабочих значений процедуры, ее размер можно подсчитать во время трансляции. В случае, если сегмент данных переполняется из-за многочисленных вызовов, операционная система может свободно его расширить и скопировать данные.

На начало сегмента данных указывает регистр G (global), на начало статики текущей процедуры – регистр L (Local).

**Целочисленная арифметика**

УВК Самсон имеет безадресную систему команд, с памятью работают только команды Load и Store (отдельные команды для целых, вещественных и адресов)

Все команды устроены следующим образом: берут один или два операнда из нужного стека и в нужный стек кладут результат.

Команды:

+, -, \*, /

+П, -П, \*П, /П – для вещественных чисел

Важная оптимизация: пусть есть формула *а-f(x),* если сразу загрузить в стек *а*, то эта позиция стека будет занята, возможно, долго, пока вычисляется *f(x).* Для некоммутативных команд – и / мы ввели команды –обр и /обр. Тогда в формуле *а-f(x)*, если левым операндом является переменная, константа или выборка поля из переменной, сначала вычисляется *f(x),* только потом загружается *а*, а затем командой –обр получается нужный результат.

**Логические команды**

Я уже говорил, что в ЕС ЭВМ логические команды, мягко говоря, были странными, то есть вырабатывались какие-то разряды кодового слова, но при этом нарисовать программу для ЕС ЭВМ *a < b & c > d* - это целая проблема. Нам, математикам, это не нравилось.

В УВК Самсон значение истина представляется 1 в стеке целых, а ложь – 0. Любое сравнение вырабатывает 1 или 0 на стеке целых, даже при сравнении вещественных чисел.

Логические команды: AND, OR, XOR снимают со стека целых две позиции, выполняют команду и кладут результат на верхушку стека. Команда NOT берет одну позицию и кладет результат на стек целых.

Таким образом, в УВК Самсон программирование логических команд ничем не отличается от программирования арифметических команд

**Передачи управления**

Исполняемые команды располагаются в специальном сегменте кодов, длина которого не превышает 64 Кбайт

* П М16 – это команда безусловной передачи управления на адрес, отстоящий на М16 байтов от начала командного сегмента
* П0 М16 – это команда условной передачи управления, переход произойдет, если с верхушки стека целых снимается 0, иначе будет исполняться следующая команда.
* П1 М16 – условный переход по единице в стеке целых

Переходы по длинному 16-ти разрядному смещению встречаются довольно редко, поэтому мы ввели переходы с коротким смещением.

Об этом следует поговорить подробнее. До того как был спаян первый провод в Самсоне, больше года мы писали тесты на интерпретаторе, причём, интерпретатор был очень детальный, который считал такты с учётом того, что, например, если переход произойдет, то 5 тактов, а если условие не выполнено, то всего 2 такта. И с помощью этого интерпретатора мы прогнали много тестов. Этот потактовой интерпретатор писала моя дочь Карина. Она в то время была студенткой 1 курса мат-меха. Так как всё это делалось по заказу Управления правительской связи КГБ, интерпретатор также был темой одного из этапов, я ей поручил сдавать этот этап, при этом я никому не говорил, что она моя дочка, просто студентка мат-меха. И вот, приехали 3 офицера, стали задавать вопросы, я и привёл в кабинет Карину, а сам ушел. Она с ними осталась разговаривать, а через 2 часа я решил заглянуть в кабинет, чисто так проверить, как дела идут. Захожу, сидит моя дочка с красными пятнами на лице от злости и восклицает: «С чего это вы решили, что не бывают автоматы с нулевым состоянием? Кто из нас математик? Я или вы?». Они ей говорят: «Ладно, ладно! Успокойся!». В итоге она сдала этот этап. И уже после того как все было решено я им признался, что это была моя дочь. Они были в шоке, спросили почему не сказал заранее, но мой ответ был прост - если бы они знали заранее, то тогда тест был бы нечестным, а так фактически она сама сдала этап.

**Короткие передачи управления**

У нас было семейное развлечение - каждый вечер Карина давала мне распечатки по полтора метра - результаты статистики исполняемых команд Самсона, упорядоченные по частоте использования. Я ногтями отрезал 15-20 верхних по частоте использования команд, потом мы эти команды внимательно изучали. Таким образом мы обнаружили, что мировая статистика полностью совпала с нашей статистикой.

По нашей и международной статистике в любой программе:

* 40% занимают команды чтения из памяти
* 20% занимают команды передачи управления
* 10% занимают команды сравнения

Таким образом, передачи управления относятся к числу часто встречающихся команд, которые стоит оптимизировать, поэтому мы решили ввести группу коротких передач управления.

PC – это счетчик команд, то есть адрес команды, следующей за исполняемой

В М8 безусловная передача вперед от PC на М8 байтов

В0 М8 условный переход по 0 в стеке целых

В1 М8 условный переход по 1 в стеке целых

Н М8 безусловная передача назад от PC на М8 байтов. Коротких условных переходов назад нет.

При реализации конструкций if и while часто возникает пара команд

= и В0 М8, поэтому мы ввели команду В= М8, сэкономив порядка 10% длины кода и пару тактов времени исполнения. Для ортогональности ввели и В!=, В<, В<=, В> и В>=, хотя они встречаются значительно реже.

На лекциях я иногда шучу, что когда проектируешь собственную систему команд, чувствуешь себя Господом Богом. Я являюсь автором машины - захотел сделать такую команду, сделал и никто мне слова поперёк сказать не может. Я до сих пор с удовольствием вспоминаю времена, когда мы только придумывали Самсон, даже не воплощали, а именно, придумывали. Я помню всё, до мелочей, - что мы делали в тот вечер, когда придумали эти команды с Кариной. Я помню, что мы ели, о чём говорили, что конкретно происходило в момент, когда мелькнула эта идея.

Хочу отметить, что мы вводим новую команду только тогда, когда есть пример на языке высокого уровня, где эта команда может пригодиться.

**Циклы**

Циклы бывают разные и сложные, но, опять-таки, всё дело «в прОцентах, как говорят наши дОценты».

Самыми частыми циклами являются for i to n do и to n do

Первый цикл занимает до 90% всех циклов, второй - примерно 8%, все остальные циклы занимают не более 2%, поэтому смысла в их оптимизации нет. Самый простой вариант - это цикл по счетчику:

to n do

реализуется следующим образом: перед входом в цикл n загружается на верхушку стека целых, затем исполняются команды:

* НСЧ выход
* Тело цикла
* КСЧ начало тела цикла

Команда НСЧ проверяет, что верхушка стека (счетчик) меньше или равен n, если это не так, то цикл не будет исполняться ни разу.

Команда КСЧ вычитает из верхушки стека 1, если результат больше 0, передает управление на начало тела цикла, иначе управление передается на следующую команду, а верхушка стека сбрасывается.

for i to n do

реализуется следующим образом: перед входом в цикл на верхушку стека целых загружаются 1 (переменная цикла) и n

* ЦК выход
* Тело цикла
* КЦ начало тела цикла

Аналогично, КЦ проверяет, нужно ли исполнять тело цикла хоть раз, КЦ прибавляет к подверхушке стека 1 и проверяет, что результат меньше или равен верхушки стека, тогда управление передается на начало тела цикла, иначе управление проваливается на следующую после цикла команду, а с верхушки стека снимается две позиции.

**Часть 3. Работа с массивами**

Тут я начну с грустной истории. Когда Самсон был уже реализован в аппаратуре, и мы гоняли множество разных тестов, то всегда выигрывали в 3 раза у IBM PC AT на 286 процессоре с частотой 16 МГц, а у нас была частота 3.25 МГц, вообще-то, инженеры обещали сделать 4, но немножко не дотянули, то есть, раз мы выигрывали по времени у этой американской машины в три раза, то это наш архитектурный выигрыш в 15 раз: у нас было в 5 раз меньше частота и при этом в 3 раза выигрывали. Мой любимый ученик Петя Лавров написал тест «Маленькая телефонная станция» на Алголе 68, и на этом тесте мы умудрились проиграть американской машине, Я рвал и метал, этого не могло быть, как так, везде выигрывали, а именно тут проиграли. Мы всё сто раз проверили, профилировали, на голове стояли, в конце концов, нашли, в чём проблема. В тесте была большая таблица - массив из структур с номерами абонентов и характеристиками и был цикл, который просматривал всю эту таблицу Когда абонент А желает соединиться с абонентом Б, мы ищем абонента Б в этой большой таблице, причем в этот момент нас интересует только одно поле таблицы – номер абонента. Оказалось что вырезка из массива с шести байтовыми элементами работает 33 такта (медленное умножение на шаг), из-за этого было такое замедление, поэтому я изменил тест, сделал отдельный массив номеров и отдельный массив характеристик каждого элемента. Я ищу номер, а когда номер нашёл то лезу в характеристики и беру оттуда информацию. В итоге мы добились того, что эти команды вырезки стали работать 3 такта, а не 33 как было изначально, после чего этот тест получил традиционный выигрыш в те же 3 раза по времени исполнения.

Как уже говорилось, исполнение вырезки a[i] эквивалентно исполнению формулы C0+i\*d, где C0 – адрес элемента массива с индексом 0, а d – это шаг (размер одного элемента). Умножение – это довольно дорогая операция.

Если учесть, что наиболее частыми массивами являются массивы из литер, целых и вещественных чисел, то в дополнение к команде ИНД, исполняющей вышеприведенную формулу, мы добавили ИНДБ (байты), ИНДЦ (целые числа) и ИНДП (вещественные числа). Соответственно в этих командах умножать надо на 1, 2 и 4, что можно сделать и без всякого умножения. Транслятор может определить тип элемента и сгенерировать соответствующую команду, сэкономив на умножении.

**Хорошие массивы**

В УВК Самсон каждый массив занимает отдельный сегмент, который начинается с паспорта масства (16 байтов). В языке Си все массивы начинаются с индекса 0, но в других языках это не так, причем во время трансляции нижняя граница не всегда известна. Нужно как-то передать информацию о нижней границе команде вырезки во время счета

Мы решили, что если генератор массива видит, что нижняя граница 0 (а если 1, то вставляем фиктивный элемент, тем самым приводя нижнюю границу к 0) и шаг равен 1, 2 или 4, такой массив объявляется хорошим, и он представляется не адресом начала сегмента, а адресом, смещенным на 17. Микрокоманда вырезки в первом же такте определяет четность индекса, если он нечетный, то прибавлять С0 не нужно (1 вычитается на фоне других действий).

Таким образом, команды ИНДБ, ИНДЦ и ИНДП, примененные к хорошему массиву, исполняются всего за 3 такта, исполнение же команды ИНД для нехороших массивов, требует более 30-ти тактов.

**Часть 4. Виртуальная память УВК Самсон**

В предыдущих главах я уже затрагивал вопрос организации виртуальной памяти. Несколько раз было сказано, что самый главный принцип виртуальной памяти заключается в полном запрете использования прямых физических адресов, то есть можно пользоваться только математическими. Математический адрес - это пара: номер сегмента и смещение внутри сегмента. Преобразование математического адреса в физический осуществляется с помощью таблицы сегментов, то есть по номеру сегмента мы «лезем» в таблицу сегментов, определяем, есть ли такой сегмент в оперативной памяти и, если его нет, то вызывается прерывание операционной системы. Затем, довольно долгим путем, во много тысяч команд, этот сегмент загружается из диска в память и, при этом какие-то другие сегменты могут быть выгружены из оперативной памяти на диск, если в памяти не хватает места. А если сегмент оперативной памяти есть, то из таблицы берется адрес его начала, прибавляется смещение, и вот физический адрес готов. Во всех машинах мира, в которых есть виртуальная память, а это сейчас практически все машины, этот физический адрес используется ровно один раз, запомнить его никак нельзя, потому что сегмент, на который он ссылается, может быть перемещен в памяти или вообще отгружен на диск. И тогда оказывается, что физический адрес указывает в никуда, в чужую память - это явная ошибка.

В УВК Самсон все эти действия выполняются абсолютно традиционным образом, когда выполняется команда читать адрес (ЧА), которая преобразует математический адрес в физический и загружает полученный физический адрес на верхушку адресного стека. С этого момента начинается новшество - нарушение организации всех правил виртуальной памяти. Мы разрешаем физический адрес использовать без дополнительных преобразований. Почему мы этого не боимся?

Поскольку и аппаратуру, и операционную систему, и транслятор реализовывал один коллектив, мы договорились между собой, что, если на какой-то сегмент есть хоть одна ссылка из адресного стека, то этот сегмент не подлежит перекачке на диск или перемещению в оперативной памяти. Но это небольшое ограничение, ведь сегментов может быть максимум 32k (на номер сегмента отводилось 15 битов), а ссылок в адресном стеке всего 16 позиций, причём многие из них ссылаются на один и тот же сегмент, чаще всего это сегмент данных текущего процесса. Поэтому мы запрещаем перемещение максимум 10 сегментов. Ну что такое 5-10 сегментов из 32000? То есть такое ограничение совершенно незаметно, зато позволяет нам переиспользовать адрес.

В дополнение к команде ЧА, о который я только что сказал, есть команда ЧРА с параметром - номер регистра, которая позволяет на верхушку адресного стек загрузить любой адрес из адресного стека, то есть это однатактовая команда, которая позволяет полученный физический адрес размножить. Обычно, при чтении лекции своим студентам, я рисую на доске пример:

for i to 1000 do a[i] :=0;

a - это массив, который представлен математическим адресом, тогда тысячу раз будет преобразование математического адреса в физический. Мы с помощью транслятора, который мы же сделали, выносим команду ЧА из цикла, ставим перед циклом, а внутри с помощью команды ЧРА размножаем, дублируем этот адрес и , таким образом, преобразования математического адреса в физический не будет, при этом мы ничего не боимся, потому что мы гарантируем, что в этот момент сегмент с массивом не может быть выгружен на диск или даже перемещен по оперативной памяти, поэтому мы считаем, что это действие абсолютно безопасно, более того, выигрыш налицо - вместо 1000 преобразований математического адреса в физический это делается всего один раз.

Существенную трудность нам составило обратное преобразование. Есть команда писать адрес (ПА), которая берет физический адрес с верхушки адресного стека и преобразует его в математический. Понятно, что такая команда безусловно нужна, хотя используется намного реже, чем команда читать адрес, но, если вы работаете с цепными списками, адресами, указателями - это довольно важная часть работа программиста, вам придется выполнять это обратное преобразование. Это трудно, ведь мы же не знаем адрес начала сегмента, но, если мы будем перебирать всю память и смотреть где наш физический адрес находится, внутри какого сегмента, то на это уйдет уйма времени.

Во время лекции я произношу студентом сакральную фразу: «когда мы проектируем свою машину, мы можем ощущать себя Богом - что хотим, то и делаем!», и в данном случае именно это и помогло. Мы просто взяли и добавили ещё одну микросхему, даже в те годы эта микросхема уже существовала (16 регистров по 16 разрядов в одном чипе) и поместили ее рядом с адресным стеком таким образом, чтобы она являлась продолжением адреса, то есть указатель стека один и тот же, просто он указывает и на адрес, и на позицию в новой микросхеме. При исполнении команда читать адрес в эту микросхему параллельно записываем номер сегмента. Таким образом, у нас есть указатель на адресный стек, который с одной стороны указывает на физический адрес, а с другой стороны на номер сегмента.

Шины при этом пришлось сделать шире, поэтому в Самсоне ширина шины 40 (24 бита на физический адрес и 16 битов на номер сегмента), но в этом нет ничего страшного. Команда читать адрес одновременно размножает и физический адрес, и номер сегмента, а команда писать адрес теперь работает тривиально. У нас есть номер сегмента, по нему «лезем» в таблицу сегментов, где точно есть адрес начала, потому что мы работаем с сегментом, который в памяти находится, получаем адрес начала сегмента, вычитаем его из физического адреса и получаем смещение. Вот вам и готовый математический адрес!

**Глава 5.**

**Архитектуры ЭВМ с повышенной надежностью**

**Часть 1. Как обеспечивается повышенная надежность**

В этом модуле мы будем говорить об архитектуре ЭВМ с повышенной надежностью. Кроме того, я приведу два конкретных примера существующих архитектур, потому что мне кажется, что нельзя рассказывать курс Архитектура ЭВМ без конкретных примеров архитектур.

Надежность - это всегда избыточность. Если схема абсолютно не избыточна, то она абсолютно не надежна, поскольку выход из строя любого элемента приведет к отказу всей схемы. Есть обратная теорема, согласно которой можно построить любую сколь угодно надежную сложную схему из сколь угодно ненадежных элементов. Это кажется противоречием, но на самом деле никакого противоречия нет. Выберите много ненадежных элементов и, каким-то образом их зарезервируйте, задублируйте, утройте, учетверите и так далее. Таким образом, можно добиться высокой надежности, но при этом будет огромная избыточность.

Когда мы говорим о компьютерах, то прежде всего приходит в голову дублирование. Если одна из ЭВМ выходит из строя, то вторая перехватывает ее работу. Разделяют горячее и холодное резервирование, при горячем резервировании обе ЭВМ все время работают и получают одну и ту же информацию, поэтому в случае выхода из строя одной из них, другая мгновенно может продолжить полноценную работу - это горячее резервирование.

При холодном резервировании резервная ЭВМ простаивает, не потребляет энергию, но при переходе управления на нее требуется определенное время для загрузки текущей информации.

**Дублированные ЭВМ**

Как определить, какая из ЭВМ дала сбой и как вообще понять, что сбой произошел? Для этого применяют специальные схемы сравнения, которые регулярно сравнивают результаты работ двух ЭВМ. Примерную схему вы можете увидеть на Рис. 10. Есть две машины, и они всё время выдают свои результаты, есть специальная схема сравнения, которая сравнивает эти результаты. Если результаты не совпадают, то она понимает, что в системе произошел сбой.

Рис. 10

Я всё время упоминаю термин «сбой», стоит отметить, что этот термин тоже требуетобъяснения. Бывает, что машина случайно сбилась - какая-нибудь альфа-частица, грубо говоря, «пролетела», и машина выдала неправильный ответ, но если эту ситуацию быстро схватить и повторить действие, при котором произошел сбой, то с вероятностью в четыре девятки, всё пройдёт без проблем, и сбой не повторится, это уже мировая статистика. Сбой происходит в 10000 раз чаще, чем постоянные отказы, то есть отказ - это ситуация, когда повторяй, не повторяй, а машина всё равно врет.

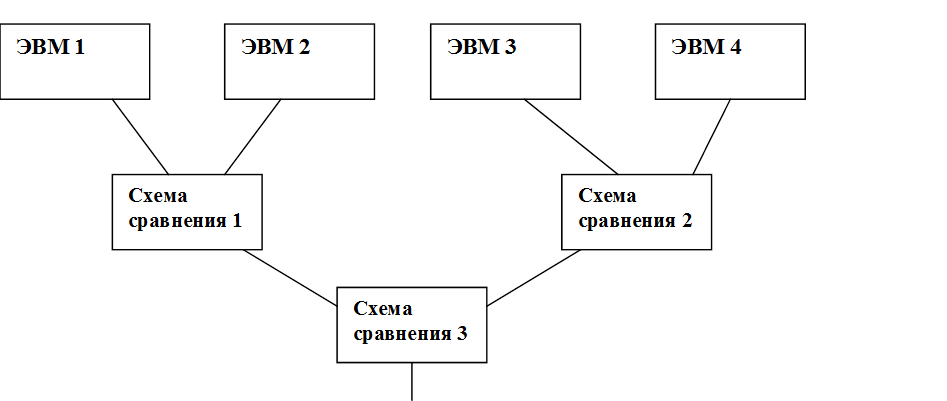
Мне было лет сорок, когда мы делали машину Самсон. Военные попросили нас посчитать надежность машины - все коэффициенты надежности, вероятность отказа и так далее. Вдруг оказалось, что несмотря на то, что я закончил мат-мех с отличием, этому меня никто не учил. Слава Богу, в это время был еще жив мой отец. Я к нему приехал (поскольку я уже был давно женат и жил отдельно) и говорю: «Папа, выручай!» Он, не вставая с кресла, протянул к полке руку, достал книгу 1939-го года издания (в то время почти все учебники издавались в Институте связи имени Бонч-Бруевича. Тогда это была одна из самых сильных научных школ связистов, и большинство учебников, в том числе математических, издавались именно там.). Достал книгу, полистал и сказал: «Прочитай от сих до сих и иди отсюда». Я прочитал, там действительно всё было написано. Тогда я впервые узнал, что распределение Пуассона (без последействия), которому меня, конечно, учили на мат-мехе, очень важно. Потому что именно в законе Пуассона есть вероятностное распределение e в степени минус лямбда t, по которому распределены сбои аппаратуры, а лямбда - это коэффициент интенсивности отказов. Вот этому нас не учили. Тогда я разобрался, что к чему и впервые узнал, что сбой бывает не в 100 раз чаще, а в 10000 чаще, чем постоянные отказы. Причём это трудно объяснить математически, потому что это своеобразный закон природы. С тех пор я это понял и до сих пор применяю в работе.

Смотрите, две ЭВМ вам выдали разные данные, которая из них врёт - левая или правая? Никто не знает! Поэтому применяют усложнённые схемы, по которым можно определить причины расхождения - запустить тесты и тогда с помощью тестов определить, кто врет, но это займет много времени. В системах реального времени этот прием совершенно не приемлем. Кроме того, может сломаться схема сравнения. Для наглядного примера давайте вспомним, рассказанный мной ранее случай, когда вылетела схема проверки в машине ЕС 1030 и когда сотрудница Вычислительного Центра ЛГУ, которая в свое время отказалась принимать машину Эльбрус на мат-мех, отключила схему контроля, и машина заработала. Абсолютно аналогичная ситуация и здесь. Можно применить 2 схемы сравнения (Рис. 11), но тогда нужна схема сравнения схемы сравнения и так далее. То есть, на наш взгляд, дублированная ЭВМ - тупиковое решение.

Рис. 11

**Учетверенные ЭВМ**

Поскольку мы работали на военных, нам время от времени давали информацию, которую получали, мягко сказать, незаконным путем. В том числе я получил довольно подробное описание управляющего вычислительного комплекса американского космического корабля Шаттл. Эту информацию вы можете увидеть на приведенной схеме (Рис. 12). Я ручаюсь, что Шаттл управляется именно по такой схеме, по крайней мере, управлялся, когда он летал.

Рис. 12

При детальном изучении оказалось, что эта схема имеет свои недостатки. Предположим одна ЭВМ дала сбой или отказала. Для определенности назовем ее ЭВМ1. Вы уже видите, что для ЭВМ1 и ЭВМ 2 схема сравнения выдала что-то сомнительное, но она ещё не сказала, которая из ЭВМ врёт. В результате правая половина хорошо работает, всё совпало, но на выходе правая половина выдаёт одно, а левая половина выдаёт совсем другое, то есть эта ситуация полной неопределённости. Затем всё это было аккуратно математически просчитано, и мы пришли к выводу, что данная схема не слишком хороша.

**Троированные ЭВМ**

Мы решили делать троированный комплекс, причём для нас было очень важно, чтобы не было ни одного устройства, выход из строя которого, приводит к отказу всей системы. Мы даже получили заказ и для военной организации сделали комплекс по такой структурной схеме. То есть все 3 ЭВМ были персоналками военного исполнения - обычные персоналки в нашем современном понимании. Они были соединены в треугольник и на каждой линии стояла схема сравнения, как указано на Рис. 13 ниже.



Рис. 13

Три обычных ЭВМ каждые, скажем, 100 мс посылают соседям информацию о своих результатах. Три схемы сравнения могут определить, какая из трех ЭВМ врет, простым голосованием. Но эта схема оказалась непригодной к использованию, поскольку сломавшаяся ЭВМ, могла испортить свою память и даже выдать какие-то команды до момента срабатывания схемы сравнения. Из теории вероятности видно, что вероятность отказа сразу 2-х ЭВМ в один такте, ничтожно мала. Если у вас 10 в минус 6 - вероятность сбоя, а у нас всегда было 10 в минус 6, то вероятность отказа 2-х ЭВМ в один такт – это 10 в минус 12. Величина настолько мала, что ей можно просто пренебречь. Так вот мы сделали эту схему, получили деньги за работу, но в жизнь она не пошла, потому что оказалось, что при выходе из строя любой ЭВМ приходится откатываться очень далеко назад, потому что никто не знает где именно произошло расхождение: в середине, в начале или в конце такта. Каждые 100 миллисекунд мы сравнивали результаты и обнаружили, что одна из машин врет, а что она за эти 100 миллисекунд наделала и какие она команды наружу выдала, никто не знает. Поэтому перезапуск после ремонта машины оказался очень длинным, и эта схема по этой причине в жизнь не пошла.

**Троированный УВК Самсон**

Наконец-то мы остановились на конечной схеме, которая изображена на Рис. 14. Это и есть троированная схема УВК Самсон. То есть здесь вы видите три процессора, три блока памяти и между ними есть схемы мажоритаров, иными словами, схемы голосования.

Рис. 14

Эти схемы голосования мы так и называли – мажоритары. Работают они следующим образом - принимают сигнал от 3 процессоров поразрядно, то есть, если шина 40 битов, значит 40 проводов и будет, и голосуют, если все три выдали одинаковый ответ – отлично. Если по какому-то разряду два процессора выдали один ответ, а третья - другой, то признается, что сбой у 3 процессора, причём всё это происходит при каждом обращении к памяти, соответственно вы имеете возможность повторить команду, и, как я уже сказал, с вероятностью четыре девятки всё будет хорошо, и перезагрузки никакой не будет, даже никакой ремонт не требуется, так как произошел просто случайный сбой, но так как мы действие повторили, мы не дали этому сбою распространиться.

Аналогично есть можоритары, которые проверяют работу памяти, на схеме они указаны, но без линий связи, но все три памяти выдают выходные результаты на все входные мажоритары, если две памяти выдали один ответ по какому-то разряду, в третья память выдала другой, то признается, что она дала сбой.

Мы определили для себя такую константу - мы повторяем три раза и, если одна и та же ситуация сбоя повторилась три раза, тогда мы эту ситуацию признаём отказом. Но ремонт при этом изумительно простой - каждый из 6 блоков (процессоры и памяти) - это отдельная плата, то есть Самсон сам говорит оператору, что отказала такая-то память или процессор, тот вынимает такой же блок из корзины ЗИП (ЗИП - это запасные изделия и приборы) и вставляет ее вместо отказавшей. На этом весь ремонт закончен, то есть здесь не только надежность, но и ремонтопригодность! Откуда взялась именно цифра 3, сам уже не скажу, но практика доказала, что это идеальная цифра! На время ремонта у нас работала машина только с двумя процессорами или с двумя памятями, в зависимости от того, что вынули. Но это разрешалось на короткое время, так как если произойдет еще один отказ и надо будет оставить только один процессор или одну память, нам работать запрещалось. Ситуация, когда в комплексе осталась только одна память или один процессор, признавалась, что машина не работоспособна. Считалось, что, как минимум, два однотипных элемента должны остаться: два процессора и 2 памяти. Стоит отметить, что машина стоит на вооружении с 1992 года и подобных ситуаций ни разу не происходило, то есть это действительно доказывает высокую надежность техники, несмотря на то, что она была сделана только из советских микросхем.

В завершение хочу сказать, что уже в наше времена была найдена информация о том, что американцы делают машины следующим образом: применяют дублированный троированный комплекс, то есть два троированных комплекса, которые задублированны. Как по мне, так это сильный перегиб, поскольку схема, примененная нами в Самсоне, за 25 лет эксплуатации показала высокую надежность. Я только хотел сказать, что путь к повышению надежности ЭВМ не заканчивается и по сей день.

**Часть 2. Архитектура IBM/360 как пример классической архитектуры**

Теперь мы приступаем к рассмотрению конкретных примеров классических архитектур. Начну с IBM/360, потому что это действительно настоящая классика. IBM/360 была объявлена в 1964 году и даже сейчас, когда люди говорят мэйнфрейм, имеется в виду именно эта архитектура, несмотря на то, что в свое время их было много. Важно отметить, что машины, которые сегодня называется мэйнфреймами, используются во всевозможных сложных расчётах, допустим в банках. Они имеют ровно такую же архитектуру, которая была объявлена в 1964 году. Результат, сами понимаете, немыслимый. То есть это архитектура живёт уже более полувека и в современном мире не собирается умирать. Стоит отметить, что у компании были и тяжёлые моменты, когда много кварталов подряд она приносила миллиардные убытки. Здесь интересно упомянуть историю, связанную с областью, которая не имеет прямого отношения к архитектуре, но имеет прямое отношение к менеджменту и крупнейшей в истории компьютерной фирме мира.

В девяностых годах был период, когда компания терпела огромные убытки и тогда совет директоров принял беспрецедентное решение - выгнали предыдущего генерального директора и пригласили на на его должность Лу Герцтнера в апреле 93 года. Это был действительно беспрецедентный шаг, потому что Лу не был ни инженером, ни программистом, до этого момента он возглавлял компанию фасованных товаров, а до этого и вовсе булочки продавал. То есть человек абсолютно не из нашего мира! И вот он смог спасти компанию от краха и даже после этого лет десять был в должности генерального директора. Мало того, что он спас компанию, так он еще и вывел ее на высокий доход. В учебниках по менеджменту это объясняется как принцип: он был чужой человек, поэтому никто не мог похлопать его по плечу по-братски и предложить пойти совместно на ланч, к нему никто, грубо говоря, переводя на наши реалии, не мог обратиться «на ты». Как результат, он выгнал половину менеджеров, безусловно у него были хорошие консультанты, так как он не специалист в этой области, какие-то направления бизнеса прикрыл, какие-то направления сильно развил, в частности при нём компания IBM практически перестала быть только компьютерной фирмой. Сегодня компания IBM стала в основном софтверной компанией. Безусловно, IBM всё ещё выпускает мейнфреймы, но этот бизнес уже даже главным не является. Это можно считать одиним из примеров грамотного менеджмента, когда человек, абсолютно ничего не понимающий в области, в которой работает, самую крупную старинную компанию в мире вывел на высокий уровень. Углубляясь в историю, хотелось бы акцентировать ваше внимание на том, что одним из создателей операционной системы OS/360 был Фредерик Брукс, автор книги «Мифический человеко-месяц», он являлся руководителем огромной группы создатели программного обеспечения, которая делала операционную систему OS/360.

Его книга “Как создаются программные комплексы” впоследствии стала Библией программистов. После ухода из компании он стал профессором в Университете Чапел-Хилл, Северная Каролина. Я с ним лично знаком. Кстати, термин «Архитектура ЭВМ» придумал именно он. Немаловажным является то, что он был учеником самого Говарда Эйкена (создатель Марк 1) и, более того, Говард был его научным руководителем.

Итак, какие особенности серия IBM/360 можно выделить.

Во-первых, это серия ЭВМ, то есть не одна машина, а много машин от самых маленьких и дешевых до самых больших и самых дорогих, но на единой схеме команд, единой архитектуре, поэтому можно, допустим, отладку делать на дешёвой маленькой машине, а счёт делать на дорогой большой. Грубо говоря, колоду карт можно отладить на маленькой машине, поставить на большую машину, и она будет работать. Это очень выгодно, потому что когда происходит отладка, нет необходимости в миллионах операциях в секунду, которые стоят огромных денег, зато, когда уже нужен счёт и в это время уже гарантированно, что программа работает и выдает правильный ответ, наоборот, нужна большая вычислительная мощность даже по высокой цене.

Во-вторых, с этого момента операционная система является неотъемлемой частью ЭВМ. До 1964 года компьютер был «голый», и на него могли ставить какой-то монитор, операционную систему, а могли ничего не ставить и работать непосредственно на тумблерах. На IBM/360 работать непосредственно на машине не является возможным, только через операционную систему.

Каналы ввода/вывода работают независимо от центрального процессора.

Собственно, ОС и каналы применялись в машинах IBM и раньше, но на регулярной основе впервые они стали применяться в серии IBM/360.

Теперь поподробнее на деталях, как она устроена: есть 16 32-разрядных целых регистров, которые пронумерованы от 0 до 15 и 4 64-разрядных регистров для чисел с плавающей запятой.

**PSW в архитектуре IBM/360**

Есть специальный регистр PSW (Program Status Word), в котором есть адрес следующей исполняемой команды (24 бита) и 2 разряда кодов условия СС (Condition Code), устанавливаемые по результатам исполнения последней команды, и еще много полезной информации.

00 – нуль или равно, 01 – отрицательный, 10 – положительный, 11 – переполнение

Соответственно в команде BC (Branch on Condition, переход по условию) можно установить 4-х битовую маску:

1000 – равно, 0100 – меньше, 0010 – больше, 0001 - переполнение. Команда может собрать переход по любой комбинации условий, например, маска 1100 соответствует переходу по меньше или равно

**Команды в архитектуре IBM/360**

Первым байтом команды всегда является код операции. В этой машине всё просто - 256 кодов и больше ни одного кода нет. В сегодняшних машинах очень часто операция размазывается по команде для того, чтобы сделать больше команд или для упаковки. В IBM/360 всё просто - код операции занимает один байт.

Регистровые команды занимают 2 байта (код операции и два 4-х битовых номера регистра), например, SR 1,2 означает вычесть содержимое 2-го регистра из 1-го и записать результат в 1-ый.

Команды формата память-память занимают 6 байтов, например,

MVC 8 *a, b,* означает переслать 8 байтов из адреса *b* в адрес *а*

Самые частые команды – 4-х байтовые команды формата регистр-память, например,

L 1, *а*

означает загрузить слово из 4-х байтов, размещенное по адресу *а*, в 1-й регистр, а команда

A 1, *а*

добавить к 1-му регистру содержимое слова по адресу *а*.

Еще один пример 4-х байтового формата:

MVI 255, *а*

Эта команда имеет однобайтовый непосредственный операнд (255), который засылается в байт по адресу *а*.

**Адресация в архитектуре IBM/360**

Адрес в команде (то, что ранее мы обозначали буквами *a* и *b*) занимает 2 байта:

Базовый регистр (4 бита)

Смещение (12 битов)

Базовый регистр указывает на начало какой-то страницы из 4K байтов, а смещение занимает 12 битов.

Дело в том, что предполагается, что память (максимальный размер 16Мб) разбита на страницы размером 4К, возможно, с большими пересечениями, чтобы добраться до нужного адреса, надо загрузить адрес страницы, в которой находится этот адрес, в один из целых регистров (кроме нулевого) и задать нужное смещение.

**Формат команды регистр-память**

* Код операции (8 бит)
* Номер регистра первого операнда, он же результат (4 бита)
* Номер индексного регистра (4 бита)
* Номер базового регистра (4 бита)
* Смещение (12 бит)

**Критика архитектуры IBM/360**

К моменту создания УВК Самсон мы имели многолетний опыт работы на оригинальной IBM/360, а потом на ее копии ЕС ЭВМ. Нас страшно раздражало, что практически в любой команде до половины битов – это нули. Индексный регистр используется редко, 15 базовых регистров никому не нужны, в УВК Самсон их всего 2, а 4095 байтов в смещении – это слишком много для процедур (обычно в процедуре 5-10 локальных переменных) и слишком мало для глобальных данных.

Мы работали на ЕС ЭВМ, более 20 лет у нас в на мат-мехе. Перед этим мы практически полных 4 года работали на оригинальной машине IBM/360, которую обходными путями достали и поставили в Москве в НИЦЭВТ. Мы туда ездили и ночами работали. За время работы мы успели хорошо ее изучить и, к моменту получения ЕС ЭВМ, мы уже стали экспертами в OS/360. К нам приезжали из разных городов для консультаций, так как разобраться в этой операционной системе было крайне тяжело. В то время мы не знали о хороших отладчиках, так как их еще не было. Мы распечатывали память, и нас раздражало, что там было много нулей. Вставал вопрос, почему мы терпим огромные потери памяти?

В команде до половины битов не используются. Весь полученный опыт нам пригодился при создании нашей машины Самсон. Программа для УВК Самсон, в среднем, в два раза короче, чем такая же программа для IBM/360 за счёт специальных команд, которых нет в IBM/360, и более плотной упаковки командного кода.

**Часть 3. Архитектура ARM как наиболее массовая в настоящее время**

Вы должны понять, что эта архитектура действительно самая массовая, именно по причине того, что большинство мобильных устройств используют внутри себя процессор ARM. Архитектура ARM - Advanced RISC Machine, то есть это RISC компьютеры компании ARM Holdings. Довольно забавная компания, должен я вам сказать, так как у них нет своего производства, завода. Представляете, самая массовая компания в области микропроцессоров в мире без собственного производства! Есть такие производители IP ядер (IP ядро - Intellectual Property), в том числе, компания ARM, которая проектирует процессорную архитектуру, которую она же придумала, описывает на VHDL, подготавливает файлы для автоматов производства чипов на фабриках (разводка, тесты на электромагнитную совместимость и т.п.), а после просто продает готовый продукт фабрикам, которые могут по этому продукту произвести компьютерный чип с этой архитектурой и с такой системой команд. На этом компания зарабатывает миллиарды. Но при всем при этом компания сохраняет у себя интеллектуальную собственность, а производство самих камней отдает на заводы.

Почему они стали самыми лучшими в мире в архитектуре - много однотипных, взаимозаменяемых регистров, применяется глубокий конвейер, то есть основные операции - однотактовые, а сложные операции организуется в виде конвейера таким образом, что каждый такт часть действия выполняется, результат передается на следующую часть аппаратуры, в это время предыдущая часть аппаратуры выполняет кусок следующей команды.

Режимы процессора ARM

* ARM (32 бита)
* Thumb-1 (16 бит)
* Thumb-2 (смешанный 16/32 бита)
* Jazelle (8 бит) – аппаратная реализация Java-байткода
* Регистры r0 - r15
* PC (r15) – Program Counter
* LR (r14) – Link Register
* SP (r13) – Stack Pointer

Вещественные регистры (FPU и векторного сопроцессора)

**Флаги условий в архитектуре ARM**

CPSR (Current Program Status Register): флаги условий, обработка прерываний:

* V oVerflow Переполнение (знаковое)
* C Carry Перенос бита (беззнаковый)
* Z Zero Нулевой результат (равенство)
* N Negative Отрицательное значение (бит 31 установлен)

**Спекулятивное исполнение в архитектуре ARM**

Практически все команды ориентированы на спекулятивное исполнение, условие приписывается после команды, и она будет выполнена, только если условие истинно, например, следующая команда выполнится, но запись в регистр произведет только, если в CPSR установлены признаки «больше» и «равно»:

addge r1, r1, r1

Установкой флагов можно управлять, если команда АЛУ имеет суффикс ’S’, то флаги будут установлены в соответствии с результатом команды:

subs r1, r2, #1

**Модификация операндов в архитектуре ARM**

Второй аргумент команд АЛУ может быть представлен в виде

ARG2 = R shift\_op B, где R – регистр, B – величина сдвига (0-31), shift\_op – один из LSL, ASL, RSL, ROR или RRX

add r1, r1, r1 lsl #2 // r1 = r1 + r1 \*4 = r1 \* 5

Величина сдвига может быть представлена не только константой, но и младшим байтом какого-то регистра, то есть вычисляться.

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

CONST\_32 = CONST\_8 << (2 \* N), 0 <= N < 16

**Команды работы с памятью**

Команды загрузки/сохранения LDR / STR:

ldr r1, [r2, #+/-imm12]

ldr r1, [r2, +/-r3, shift imm5]

ldr r1, [pc, #256]

ldr r1, [sp, r2, asl #2]

**Вызов функций**

Вызов функций выполняется при помощи команды

bl – branch with link

Текущее значение регистра pc сохраняется в регистре lr

Возврат из функции выполняется с помощью команды bx lr

some\_func: bl some\_func

push {lr}

… some\_func:

pop {pc}

bx lr