Московский Государственный Университет имени М. В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Кафедра Автоматизации Систем Вычислительных Комплексов

НЕ ЯВЛЯЕТСЯ ОФИЦИАЛЬНЫМ УЧЕБНЫМ ПОСОБИЕМ

Вычислительные системы

(VII семестр)

лектор — профессор А. Н. Томилин

составитель — a_s@interwave.ru

Москва 2006

Содержание

Введение 3
Классификация архитектур вычислительных систем 11
Параллелизм работы основных устройств машины 12
Параллелизм работы АЛУ 22
Векторно-конвейерная ЭВМ: Cray 26
Модульно-конвейерный процессор (МКП) 32
Электроника СС БИС («Красный Крей») 36
ILLIAC VI 38
Многопроцессорные вычислительные комплексы 40
NUMA 40
MPP 40
Иерархия памяти 53
Кэш (cache) 54
Структурная организация оперативной памяти 59
Сегментная организация памяти 62
Страничная организация памяти 65
Сегментно-страничная организация памяти 67
Средства поддержки операционной системы 70
Переключение процессов 70
Аппаратура прерываний 75
Защищённая подсистема. Многоуровневая защита 77
Аппаратная поддержка взаимодействия программных модулей 79
Типы устройств внешней памяти и ввода-вывода 80
Дисплеи 82
Диски 83
Печать 85
Многомашинные комплексы 91
Заключение 99
Вопросы по курсу «Вычислительные системы» 101
Литература к курсу «Вычислительные системы» 102

Введение

Сегодня мы начнём (будет много вводных предложений) курс: «Вычислительные системы» для 3-го потока 4-го курса. Фамилия моя Томúлин, Александр Николаевич, профессор кафедры АСВК со времён её основания, а также основания факультета, т.е. с 1970 года, когда её возглавил Лев Николаевич Королёв, которому послезавтра исполняется 80 лет. Будет большая конференция, будут выступать все его ученики, которых очень много, много всяких корреспондентов академии наук, докторов наук.

Лев Николаевич, конечно, это большая школа, школа программирования, школа разработки с точки зрения моделирования, проектирования архитектуры вычислительных машин и их использования. Очень развита школа и современных компьютеров. Ну так вот, он человек совершенно замечательный, человек очень дружественно расположенный, ну, собственно говоря, он мой научный руководитель был, да и сейчас коим, конечно, является. И, конечно, все кто его знает, те отмечают его исключительное дружелюбие, исключительный такт в общении, исключительно положительную настроенность. Вот я сообщаю вам положительный факт, что в хорошем настроении, в хорошей отдаче живёт наш замечательный профессор Лев Николаевич Королёв, выпускник механико-математического факультета МГУ, воевавший на фронте в Великую Отечественную войну. Тут больше ничего не добавить.

Вычислительные машины вначале так и назывались — «вычислительные машины». Слово «компьютер», во всяком случае, в нашем отечественном лексиконе, привилось далеко не сразу. Но уже, когда вычислительные машины стали обладать многими уровнями параллелизма, подсоединениями многих внешних устройств, каналов, высокий потенциал внутренних устройств развивался всё больше с развитием параллельной обработки информации в самих центральных устройствах машины, их стали называть вычислительными системами. Ну и конечно объединение вычислительных машин, оно началось с самого начала. Как только в какой-то организации появилась вторая машина, так сразу появился уже двухмашинный комплекс. И это, естественно, для разных целей. Вторая машина, как правило, являлась более производительной, на неё и перекладывалась задача обработки информации, а предыдущая машина приспосабливалась к выполнению функций ввода-вывода. Или, если машины одного уровня производительности, то они использовались и для разделения труда по обработке информации, для резервирования в случае таких систем, как когда в случае выхода из строя одной машины тут же переключаться на продолжение вычислений с сохранением информации на каком-то этапе на внешней памяти машины. По разным причинам возникали эти многомашинные комплексы. Т.е. они возникли на много раньше многопроцессорных. Ну и продолжают существовать и сейчас.

Кстати, о том, что они были актуальны: как-то, после того, как уже было выпущено какое-то количество машин БЭСМ-6, была конференция на Украине в палаточном городке Института кибернетики Академии наук Украины. Конференция была такая: «Связь БЭСМ-6 с другими ЭВМ». И выяснилось, что число многомашинных комплексов (от 2-х и более машин в каждом комплексе), в который входит машина, БЭСМ-6 в Советском Союзе превышает количество выпущенных машин БЭСМ-6. Это вот что касается вообще понимания вычислительных систем. Ну а затем стали появляться многопроцессорные вычислительные комплексы. Сейчас существуют системы с громаднейшим количеством процессоров. В данном случае сейчас это всё на микропроцессорах строится, на достижениях технологий последних десятилетий. И эти системы — они требуются. Постоянно возрастают мощности машин в списке TOP500 (http://www.top500.org) — это список наиболее высокопроизводительных вычислительных машин в мире. И вот, в каждом новом списке они несколько раз в год меняются. Сейчас эти наиболее высокопроизводительные установки — это десятки и уже даже сотни Тфлопс, т.е. тысяч миллиардов операций в секунду. Но это именно Тфлопс, т.е. операции над числами, представленными в системе с плавающей запятой, полноразрядными. Конечно, это для вычислительных задач существенно. Правда, есть задачи, которые не требую большого количества арифметических операций над числами с плавающей запятой, логические задачи. Но так уж традиционно, вычислительные машины с самого начала производились для вычислительных задач, но поскольку их область применения расширилась у вычис-

лительных систем так же, но, как видите, оценивается в Тфлопс. Ну, т.е. есть Мфлопс, есть Гфлопс (миллион операций в секунду с плавающей запятой). Кроме того, есть MIPS (million instruction per second) — это количество операций, проходящих через устройство управления машины. Вовсе не все эти операции участвуют в арифметических вычислениях. Есть какие-то операции подготовки, считывания данных, какое-то ещё преобразование для дальнейшего использования в вычислениях. Поэтому, как правило, даже для счётных задач вот таких операций с плавающей запятой (это, собственно говоря, операции +, –, ×, ÷) их от числа всех операций приблизительно 1/3. Да и вообще надо смотреть в контексте, когда говорят о числе операций в секунду, что это за операции. Если это операции действительно над числами с плавающей запятой да ещё 64-х разрядными числами.

Вот такой гротескный пример. Как-то довольно давно в зале президиума академии наук обсуждался ход разработки отечественных суперЭВМ, которые по своей архитектуре, структуре да и возможностям не на много уступали состоянию дел американской техники того же времени. Конструктор рассказывает, какие проблемы есть с организацией электропитания, организации охлаждения. И этот вопрос мы с вами тоже рассмотрим, что очень интересно, какие пертурбации произошли за какой-то период. В технической реализации вычислительных систем был момент, когда казалось, что всё: наступает тепловая смерть вычислительной техники. Как когда-то, не то, чтобы я увлекался шахматами, но интересно было исследовать этот вопрос. И вот в своё время, как в мире, так и в литературе, говорилось, что всё: мир ждёт ничейная смерть шахмат. Потому что сейчас самые выдающиеся гроссмейстеры друг другу ни в чём не уступают, и не будут уступать. И всё большее число людей будут становиться столь совершенными в этом вопросе, что выиграть на одном уровне будет просто не возможно. Турнир будет обречён на погибание: это же не интересно, когда практически все встречи будут заканчиваться вничью. К счастью этого не случилось до сих пор, но угроза такая существовала. Так же и тепловая смерть вычислительной техники, т.е. ситуация сводилась к тому, что уже на столько быстро протекают процессы в интегральных схемах, что при большой частоте разогревается схема. Это естественный физический процесс. Нужно отводить тепло. Начали схемы помещать (машина Cray, которую мы будем рассматривать как образец достижения нескольких уровней параллелизма информации от первой до следующей) на трубу, через которую протекает хладагент, отбирая выделяемое тепло. Потому что каждая схема выделяла 5 Вт, этих схем было очень много, и если это охлаждение убрать, то человек в момент вспыхнет, и в этом пламени ничего не останется от этой машины. Конечно, приходилось очень следить за тем, чтобы система охлаждения правильно работала. Если неправильно, то нужно отключать. Специально систему мониторинга для этого вывели, и машина этим контролировалась.

Cray достиг производительности 1 Гфлопс (миллиарда операций в секунду). Но это дело уже 80-х гг. Потом для достижения большей производительности от фирмы Cray некая американская фирма «Эта» решила достичь 10 миллиардов и вынуждена была окунать всё оборудование процессора и памяти в криогенную среду, т.е. с высокой степенью охлаждения, чтобы отбирать это тепло. Тогда были частоты порядка 100 МГц.

За счёт интенсивных исследований был достигнут исключительный прогресс. В результате при тех же скоростях перешли обратно от водяного охлаждения или даже вот такого вот экстремального, перешли обратно к воздушному охлаждению. На это тратились огромные усилия разработчиков, ну и развивались технологии не только самого вычислительного дела, но и многие технологии материаловедения в мире. Вычислительная техника была заводчиком этого дела.

И вот в этом TOP500 вы можете всё увидеть. У нас, кстати, есть TOP50 (http://www.supercomputers.ru) — это список наиболее производительных вычислительных систем, установленных в России и странах СНГ (по-старому — в Советском Союзе). Имеются тоже крупные системы, наиболее крупная система — 15 Тфлопс, установлена в так называемом здании президиума академии наук с высокой навороченной верхушкой, Ленинский проспект, дом 32. Почему «так называемом», потому что сам президиум академии наук там не сидит. Он сидит по прежнему в прекрасном особняке среди зелени, отодвинутом от шум-

ного Ленинского проспекта, на Ленинском, 14. Так вот, там находится известный суперкомпьютерный центр, очень хорошо оснащенный входными телекоммуникациями, естественно внутренней сетевой средой. Там не одна, несколько машин. Но вот одна из наиболее производительных — это машина 15 Тфлопс, разработанная у нас, однако на зарубежных микропроцессорах, хороших, конечно, и т.д., но вот, тем не менее, такова сейчас ситуация.

Кстати, на это вы можете выйти через сайт НИВЦ МГУ: www.parallel.ru — это сайт вычислительного центра, созданный коллективом и под руководством Владимира Валентиновича Воеводина (сейчас там зам.директор ВЦ), выпускника кафедры АСВК. Сайт замечательный: там есть и история, и все характеристики современных различных классов высокопроизводительных вычислительных систем, там есть и рассмотрение фирм, учебные материалы там представлены наших сотрудников, ВМиК и Петербургского университета.

Какова же будет наша с вами задача? Конечно, как специалисты в области программирования, вы должны быть грамотны в области состояния современных вычислительных систем разных классов, разного назначения использования, применения. Поэтому эта цель естественна. Она достигается в обзорном порядке в каком-то смысле, потому что рассмотреть подробно все вычислительные системы не представляется возможным. Но эта цель попутная, которую мы должны достигнуть. Какая же цель главная? А главная цель такая: рассмотреть развитие идей в области архитектур, структур вычислительных систем, развитие мышления конструкторов, разработчиков средств поддержки развития архитектуры, интегрирования. А рассмотрение развития их мышления — это есть основа развития собственного мышления, и вашего, и моего, и всех тех, кто работает в нашей области. Поэтому вот это мы будем рассматривать в первую очередь. На базе какого основного стержня мы будем это рассматривать? На базе стержня рассмотрения развития параллелизма обработки информации в вычислительных системах. Естественно здесь будут привлекаться исторические аспекты. И рассматривая всё более новые и новые уровни развития параллелизма, достигнутые разработчиками вычислительной техники, на примере машин, наиболее хорошо отражающих появление и реализацию этих идей, мы и будем вести рассмотрение нашего курса. В какой-то мере будет повторение, может быть в несколько новом аспекте, некоторых моментов, которые у вас были и в курсе ЭВМ, и программирования, может быть ещё где встречались. В этом нет ничего плохого, это только хорошо и послужит вам для закрепления этого всего материала.

Конечно, есть компромисс между тем, что решается аппаратно (на уровне hardware), и тем, что решается программно (на уровне software). Этот компромисс существовал всегда. В какой-то момент, hardware, развившись до какого-то уровня, позволило существенно развить software. Кстати, есть очень интересный момент в этом смысле при создании машины БЭСМ-6. Когда была создана машина БЭСМ-6, вдруг было создано 5 операционных систем, 6 автокодов (языков и, соответственно, трансляторов ассемблера с этих языков). Потом, в конечном счёте, конечно, всё это свелось к одной-двум системам, активно используемым и хорошо поддерживаемым. Зачем всё это было произведено? Казалось, что в каждом конкретном случае есть определённые причины. Например, в Институте прикладной математики (ИПМ), который возглавлял президент академии наук Мстислав Всеволодович Келдыш, он очень ревниво относился к приоритету своей организации в самых разных вопросах: в вопросах вычислительной математики и в вопросах программного обеспечения. Поэтому появилась ОС ИП. Казалось бы в этом причина. Появились операционные системы варианты на Урале, появилась операционная система, естественно, в организации, создавшей машину — Институт точной механики и вычислительной техники, Ленинский, 51. Тут и Лев Николаевич Королёв, Виктор Петрович Иващенко, заведующий кафедрой системного программирования, и ваш покорный слуга являются альма-матер, для которого кроме, естественно, факультета это этот институт во главе с академиком Сергеем Алексеевичем Лебедевым — основателем отечественной вычислительной техники. Оттуда, собственно, сюда пришёл Лев Николаевич Королёв, ну и продолжает работу ту же самую: создание новых машин и программного обеспечения. Институт покинул, но работает параллельно в том же направлении.

Вот казалось бы, что везде есть своя причина появления нескольких ОС и автокодов. На самом деле, это были поводы. А причина была очень глубокой: появилась среда (машинная) и возможность развить себя, создать сложную программную систему, т.е. таким образом развить своё мышление на имеющихся средствах, т.е. в машине (создать для неё). И готовые к этому коллективы, где эта потребность существовала, она прорвалась, и люди сделали эти вещи, развив тем самым себя и создав основу для затем продолжения разработки других. Т.е. речь идёт вот о чём: в своё время постулаты диалектического материализма (это вещь наблюдательная, это выводы, сделанные на основе наблюдения окружающего нас мира). Вот одним из положений диалектического материализма было такое (потом это положение убрали и понятно почему): мир развивается по законам развития живой и неживой природы, нам неподвластным. Отсюда следует, что это либо от божественного начала, либо от «большого взрыва» — образовались какие-то физические константы и всё дальше идёт. Человек может лишь временно ускорить где-то или замедлить этот процесс, но повлиять кардинально на его изменение не может. И действительно, повинуясь этому закону, вот эти появились параллельные разработки. Потом этот постулат убрали, потому что никак не совпадало с руководящей и направляющей ролью коммунистической партии. Точно так же, повинуясь этому закону развития производительных сил, сразу несколько групп исследователей пришли к разработке вычислительных машин.

Первой вычислительной машиной считается машина ENIAC, разработанная американскими учёными Джоном Мачли и Джоном Проспером Экертом (на электронных лампах). Эта машина была не машиной с хранимой в памяти программой. Это была электронная вычислительная машина, но программа набиралась на штекерных устройствах. Поэтому полностью электронной вычислительной с хранимой программой назвать нельзя. Тем не менее, считается первой электронной вычислительной машиной. Но параллельно с американцами шла разработка и у нас С.А.Лебедевым в Киеве тогда (потом уже и здесь). Сам он родился в Нижнем Новгороде в 1902 году, и он также развил эти идеи. Он был одним из беднейших мировых учёных электротехники: расчёт линий электропередач, регулировки электрогенераторов, у него было много книг. И вот будучи 40 с чем-то лет отроду, он занялся созданием вычислительных машин. Дело в том, что тогда на специальных анализаторах, моделирующих решение дифференциальных уравнений на процессах протекания электрического тока, проводились расчёты, и это их устраивало: и точность и время. И, видимо, это толкнуло на то, чтобы создавать средства расчетов, которые ему были необходимы в своей активной деятельности. Потом он полностью переключился на деятельность в области вычислительных машин. И уже в 50-м году появилась Малая электронная счетная машина (МЭСМ) в Киеве, она была самой производительной машиной в Европе, которая была уже машиной с хранимой в памяти программой.

Параллельно с Лебедевым этими же вопросами занимался и другой отечественный исследователь и его группы — это Исаак Семёнович Брук. И если Лебедевым была основана школа высокопроизводительных машин, то Брук — школа малых машин, управляющих машин, которые предназначались для управления различными производственными объектами, для малых вычислений.

Так вот смотрите: оба родились в 1902 году с разницей в несколько дней (Лебедев — 2-го ноября, а Брук — 8-го ноября), оба сдали первые вычислительные машины (Лебедев в Киеве, Брук в Москве) в одном и том же 51-м году почти одного и того же числа (25 декабря) Государственной комиссии. Оба умерли в 1974 году с разницей в 2 месяца. Случайны ли эти совпадения?

А потребности, конечно, были. Для этого делались попытки механизации вычислений — это машина Бэббиджа (ещё у него первой программисткой была его дочь). Многие другие исследователи делали это дело.

За рубежом есть премии Computer Pioneers (общество III) — они даются живым людям — пионерам вычислительной техники. И вот несколько лет тому назад этой премией наградили 3-х, но уже не живых, пионеров вычислительной техники нашего отечества, а именно С.А.Лебедева, А.А.Ляпунова и В.М.Лужкова. Аргументация была такая: в связи с железным

занавесом, не возможно было получить достоверную информацию по достижениям учёных, и после того, как это удалось сделать, стало ясно, что они являются полноправными участниками мировой шеренги, общество решило эти медали вручить.

После МЭСМ почти сразу появилась машина БЭСМ — большая электронная счётная машина академии наук — разработка коллектива С.А.Лебедева.

И вот первая международная конференция в Кронштадте в 1955 году, где подводились итоги первого десятилетия компьютерной эры. На эту конференцию едет наша делегация. Из-за всяких проволочек наши приехали, когда конференция закончилась: конференция закрылась и в зале появляются наши. Но т.к. доклад Лебедева был запланирован, решили собраться на утро следующего дня: собралась вся конференция, Лебедев сделал доклад, было признано, что это самая высокопроизводительная машина в Европе, практически равная американским вычислительным машинам того времени. И сразу было выпущено несколько описаний этой машины на английском, французском и немецком языках.

Таким образом, наши разработки ни сколько не уступали американским аналогам, и даже кое-где их превосходили. Долго мы не проигрывали в области архитектуры, структуры, программирования, и начали проигрывать технологии с какого-то момента времени.

Появилось довольно много различных конструкций, появились высокопроизводительные машины. В 60-х годах появилась машина БЭСМ-6, которая практически совпадала по параметрам с зарубежными аналогами. Появились разные другие машины: в Минске — «Минск», появились малые управляющие машины (последователи Брука), вычислительные комплексы для систем реального времени. Потом появился Научно-исследовательский центр электронной вычислительной техники (НИЦЭВТ). Направление машин единой серии (ЕС) — это головное направление разработок НИЦЭВТ.

В 1956 году прошла конференция, где Лебедев высказал идеи, все те, которые сейчас реализуются: идеи сверхбыстрой оперативной памяти (кэш), идеи многопроцессорности. Интереснейший был доклад, и конференция называлась совершенно музыкально: «Пути развития советского математического машиностроения».

Естественно, что программисты писали очень хорошие программы, старались максимум ужать, ускорить, уменьшить количество используемых ячеек памяти. Сейчас как говорят: «У тебя медленно работает? Купи ещё сколько-то мегабайт памяти!» Но даже на имеющихся мегабайтах памяти можно в 10 раз уменьшить, в 100 раз ускорить, но годочек посидеть над этим делом. Это всё идёт от зарубежных фирм: приходит человек на работу, вот тебе задание. Он говорит: «Я могу сделать это очень хорошо, но мне нужен год» — «Нет, ты сделаешь это за 2 недели, используя имеющиеся средства» — «Да, но это будет неэффективно!» — «Чёрт с ним! Им не понравится, будет медленно, пусть купят новую машину. Она с большей частотой и тогда у них будет проходить нормально». Есть даже сведения, что в некоторые места в операционных системах вставляются задержки, чтобы вам было менее комфортно, и вы купили новую машину! Главное — быстро вывести на рынок.

Момент по системе противоракетной обороны. Это был многомашинный комплекс. Под руководством Всеволода Терентьевича Бурцева, а программный комплекс делался под руководством Льва Николаевича Королёва, этот комплекс на машине М-40 (40 тысяч операций в секунду, память — 4 килослова). И на этом работали программы системы жёсткого реального времени. Первое испытание — 1961 год (американцы подобное сделали только через четверть века), и в первое же испытание первой же ракетой была сбита нападающая баллистическая ракета — восторг! А всё записывалось: получаемые данные, выдаваемые данные и т.д. Дело в том, что подрыв заряда противоракеты должен происходить, когда расстояние до нападающей ракеты было порядка 25-30 метров. Если больше, то смысла нет. И вот когда всё проверил, то оказалось, что расстояние было значительно больше. В чём же дело? Оказалось, что всё произошло по ошибке программиста: расстояние определилось как маленькое, произошёл подрыв, и т.к. конус разлёта осколков был удачным, то нападающая ракета была сбита. Потом научились сбивать, как отче наш.

Потом научились проигрывать имеющуюся музыку, писать новую. Выпускник мехмата 58-го года, прима артист театра МГУ Андрей Михайлович Степанов как раз писал програм-

му определения траектории нападавшей ракеты. А потом написал программу воспроизведения музыки на машине БЭСМ-6: 6 динамиков вешались на 6 разных регистров, он задавал программу, которая как-то закодированные ноты переводила в коды, которые передаются на регистры. Ни один орган в мире не мог воспроизвести подобного. А потом он написал программу сочинения музыки строгого стиля а-ля фуги Баха. Музыку записывали на магнитофон и проводили заслушивание на учёном совете в консерватории. После прослушивания встал из них самый главный и сказал: «Своим студентам за эти произведения я ниже «тройки» бы не поставил». А потом шла программа разворота противоракет — программа жесткого реального времени: рассчитывались поправки на машине и передавались указания на рули — за 0,1 с — это была программа Геннадия Георгиевича Рябова.

Вспомнился анекдот про диалектику из серии анекдотов про Петьку и Василия Ивановича:

Петька спрашивает у Василия Ивановича:

— Скажи, пожалуйста, ты в верхних кругах вращаешься, что такое «диалектика»?

На что Василий Иванович отвечает:

— Это, Петя, такая тонкая материя. Так вот на словах не объяснишь. Давай я тебе пример?

— Ну хорошо, Василий Иванович, давайте.

— Ну вот: встречаются два солдата — один чистый, другой грязный. Перед ними баня. Который пойдёт в баню?

— Грязный пойдёт.

— Нет, Петенька. Чистый пойдёт. Он привык себя блюсти. А грязный… Ну ты же знаешь нашего брата — солдата.

— Совершенно верно! Именно всё так и бывает. Вы совершенно правы, Василий Иванович. Всё так и будет.

— Хорошо. В другой день встречаются два солдата — один чистый, другой грязный. Который пойдёт в баню?

— Чистый же пойдёт. Вы же сами говорили…

— Нет, Петенька. Грязный пойдёт в баню. Его вошь на столько заела, что сил уже нет. А чистый — подождёт.

— Истинно так! Всё так и будет. Как Вы говорите — всё правда.

— В третий раз встречаются два солдата — один чистый, другой грязный, перед ними баня. Который пойдёт в баню?

Далее следует специфический энергичный взгляд со стороны Петьки, на что Василий Иванович и отвечает:

— Вот, Петя. Это и есть диалектика.

Как же быть с Джоном фон Нейманом? Все говорят: «архитектура фон Неймана», «машина фон Неймана» и т.д. Всё что нас окружает, все установки (большие и малые, персональные), всё это — машины архитектуры фон Неймана. В чём тут дело? А дело вот в чём. Фон Нейман — это знаменитый очень известный математик, по происхождению — американец, был очень известным учёным в 40-х годах. Есть масса книг о нём. Джон фон Нейман участвовал в атомном проекте, видел то, что для крупных проектов не хватает имеющихся средств. Он, конечно, заинтересовался первой машиной, которая была сделана до него, но он не принимал участия в создании вот этой первой машины ENIAC. Он заинтересовался, включился в коллектив разработчиков, и уже следующая машина была создана с его участием. Как крупный учёный, он обобщил имевшиеся до него достижения, и опубликовал основные принципы (так они и стали называться «принципами фон Неймана»), а архитектура, соответствующая этим принципам стала называться архитектурой фон Неймана. Что это за принципы: это управление программой, которая храниться в памяти машины, это двоичная система счисления, набор операций (арифметических, логических), управление, переход на

другие ветви вычислений и т.д. Так что, с одной стороны он не является родоначальником этого дела, с другой — он активный участник этого процесса.

В противовес появляются машины, управляемые потоком данных (не командами). Если у вас достаточно в машине оборудования, то эта машина действительно потоком данных реализует заложенный в алгоритме параллелизм. Конечно, команды как-то заложены, но опосредовано. Важно, что получающийся результат какой-то операции находит себе пару в ассоциативной памяти, и вместе с этой парой (обычно двуместные операции) там идёт кодирование операции. Т.е. она не выступает как главное действующее лицо, а как подчинённое. Такая ассоциативная память становится устройством управления. Управление осуществляется путём сравнения. Конечно, программирование здесь очень сложное, и хотя заманчиво иметь полный параллелизм, но до сих пор таких машин в мире в больших количествах нет.

Лебедев со своими лучшими учениками выполнял проект интересной машины Dataflow (управляемой потоком данных). И сейчас этот проект продолжается, но под руководством А.М.Степанова.

Есть понятия «архитектура ЭВМ» и «структура ЭВМ». Особенно нехорошо сейчас обстоит дело с термином «виртуальность». Что такое «структура ЭВМ»? Скажем так: вот у вас есть какие-то каналы связи с внешними устройствами одного типа или другого типа (селекторный или мультиплексный, какой-то канал на базе общей шины, к которому подсоединяются многие устройства). Работу этих кагалов обслуживает операционная система, и для пользователя это может быть совершенно не интересно, ему это не важно. Какому угодно пользователю: прикладному, системному. Вот это — структура ЭВМ. Сколько у вас памяти сверхбыстродействующей, в которой автоматически накапливаются наиболее часто используемые данные (память cache). Одноуровневый, двухуровневый. Это к программированию никак не имеет отношения. Вот это структура — внутренняя организация, которая никак не отражается на проблемы программирования. А вот архитектура ЭВМ… Есть два основных подхода. Первый — чёткий — система команд машины. Например: вот у вас система команд для выполнения действий над числами, представленными с фиксированной запятой; другая система команд — для чисел с плавающей запятой. Наборы команд совершенно разные. Ещё пример: содержаться команды обработки векторов или не содержаться. Если таких команд нет, то выполняется обработка циклом. Вот это существенно. Второй подход — более расплывчатый в том смысле, что описывается система в виде некоторой многоуровневой системы. Естественно уровни аппаратуры, уровни системного ограничения, уровни библиотек, уровни middleware, прикладные программы, административный уровень и т.д. И обязательно интерфейсы между уровнями.

Мы с вами обсуждали, что был период «золотого века» отечественной вычислительной техники, когда наши разработчики, инженеры, программисты находились над уровнем зарубежных коллег. Это признавалось и сейчас во всех исследованиях признаётся. Сейчас же нас существенно обошли. Этому есть ряд причин. Что же на самом деле произошло?

Статья «Москва компьютерная» (www.computer-museum.ru — сайт виртуального компьютерного музея — раздел «Развитие вычислительной техники в СССР») была заказана к 850-тилетию Москвы:

«…Компьютеростроение стимулировало и вбирало в себя лучшие достижения смежных областей науки и техники: сначала – электроники, а затем и микроэлектроники, вложившей информационную мощь в миллионы мельчайших транзисторов. Микроэлектроника в последние 15–20 лет стала главной движущей силой компьютерной революции. В результате в условиях открытого информационного пространства в мире сформировалась массовая компьютерная индустрия, осуществляющая принципиально новую социальную функцию: обеспечение лавинообразного накопления информации, которая становится главной регулирующей силой жизни человечества.

…В конце 40-х годов к созданию ЭВМ оказались готовы только три страны: США, Англия, СССР.

Для разработки, развития и применения средств вычислительной техники (ВТ) необходимы следующие условия:

- постановка масштабных актуальных задач, не поддающихся решению без применения средств ВТ;
- наличие технической инфраструктуры и передовых технологий для разработки и применения ВТ;

- наличие вузов с профессорско-преподавательским составом, способным вести подготовку кадров в данной области…

Быть или не быть?

Всякое новое социально-значимое явление в жизни человеческого общества подвергается со стороны власть предержащих самой тщательной проверке и экспертизе. Так велит закон самосохранения власти. Колоссальная роль вычислительной техники для будущего – не только как новой области науки и техники, но и как принципиально нового фактора социального влияния – стала предметом пристального внимания политиков и идеологов с обеих сторон "железного занавеса".

В начале 50-х годов в США по инициативе правительства (о чем у нас почти совсем неизвестно) была развернута общественная дискуссия на тему "Несут ли компьютеры угрозу американскому образу жизни?". Правящие круги испытывали серьёзные опасения из-за возможного нарушения баланса на рынке труда. К экспертизе социальных последствий были подключены активно работавшие в области кибернетики ученые с мировыми именами: Дж. фон Нейман, Н. Винер, К. Шеннон и др. Вердикт всесторонней и независимой экспертизы поражает своей дальновидностью: при грамотном использовании компьютер усилит позиции общества, основанного на свободной конкуренции. Только после этого власти США дали зеленый свет свободному рыночному развитию вычислительной техники.

Этот шаг, на первый взгляд, локального (для одной страны) значения воспринимается сегодня как открытие, масштабы которого только начинают осознаваться социологами. Широкое и свободное развитие вычислительной техники с неизбежностью приводит к созидательной информационной трансформации общественного сознания в целом.

Так искали и находили ответы на вопрос "быть или не быть" власти на Западе. В нашей стране блюстители идейной чистоты, к сожалению, поспешили определить кибернетику, а вместе с ней и вычислительную технику, как "буржуазную лженауку". Чтобы лечить "болезни" с таким диагнозом, надо было иметь гражданское мужество. В защиту вычислительной техники активно выступили академики А. И. Берг, А. А. Дородницын, С. А. Лебедев. В результате в эшелонах власти вычислительная техника получила зеленую улицу, но только для решения задач достижения военного паритета. Ревнители идей обобществления вслепую отвергли для широкого общества то огромное богатство обобществленной собственности, которое сейчас лавинообразно реализуется в среде мировых информационных ресурсов. Таковы парадоксы истории…»

Раньше был лозунг: «От каждого по способности, каждому по труду, его общественно-полезных результатов». Первое как-то достигалось: никто не мешал работать, кто не хотел — тот не работал (недостаток был). А вот «каждому по труду» — это не получалось. Таким образом, выдвинутый лозунг для существования общества оказался нереализованным. Отсюда и крах общества: уравниловка и прочие.

Когда мы работали, мы видели — вокруг дело новое, а вот это — дело тёмное. Поэтому, конечно, хотелось что-то сделать, да ещё к тому же, понимая, что это нужно для развития общества. Нас поддерживало то, что оценивалась наша работа не хуже других направлений деятельности в нашей стране. И это создавало хорошее настроение.

Сейчас этот лозунг тоже не достигается (от каждого по способности, каждому по труду). К сожалению, произошло резкое расслоение. И те, кто получает что-то большое, это уже не по труду, не по количеству вложенного в общество. А просто-напросто у него оттягивает. Т.е. переход на новые рельсы жизни лоббируют в первую очередь те, кто потащили на себя одеяло государственных средств.

Что делается в Америке? А в Америке делается следующее: как только подназревает новый этап, чтобы поддержать конкурентную способность Америки, они выделяет сотни миллиардов долларов на развитие программного обеспечения супермашин. Куда? Конечно, безвозвратно!

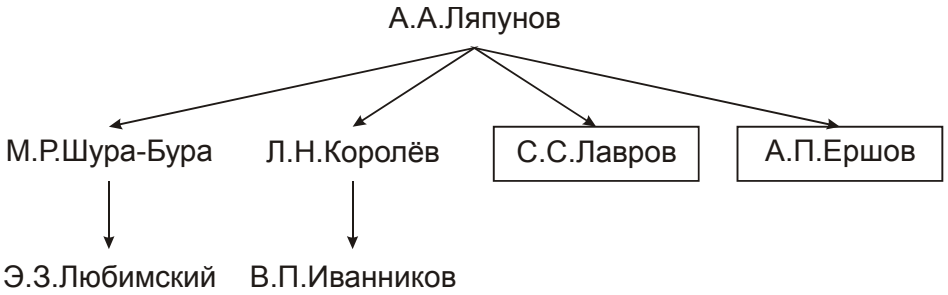
«…История не злонамеренна. Она дает шанс всем, кто в нем нуждается. Сегодня мы должны и можем начать вторую попытку. Но для этого надо до конца выправить идейные ошибки прошлого, которые обрекли страну на самоизоляцию.

Вплоть до переломного момента начала реформ Москва оставалась для страны компьютерным локомотивом. Налицо парадокс: лидеры реформ, сумевшие снять с великой страны "железный занавес", не видят для нее своих путей к информационно-компьютерному суверенитету…»

Им не дают видеть негодяи, которые всю страну направили на собственное накопление.

«...Накопленный в стране интеллектуальный потенциал, способный активно влиять на будущее страны, остается невостребованным. Неужели на ухабах реформ Москва отказалась от бремени лидерства? Не хочется думать, что после 850 лет преодолений на очередном затяжном подъёме стало невмоготу крутить компьютерные педали».

Статья давняя, но всё-таки достаточно актуальная.



А.А.Ляпунов считается основателем программирования. М.Р.Шура-Бура — долго возглавлял кафедру Системного программирования, затем был В.П.Иванников. М.Р.Шура-Бура возглавлял всё это дело в Институте прикладной математики (ИПМ), Л.Н.Королёв — в Институте точной механики и вычислительной техники. И всюду работали до кафедры и в содружестве, и в соперничестве по вопросам системного программного обеспечения. С.С.Лавров — он работал и по программированию космических дел, потом возглавлял программистов по тем же делам в Вычислительном Центре Академии наук. Э.З.Любимский сейчас возглавляет центральную комиссию в Институте прикладной математики (ИПМ). А.П.Ершов — член-корреспондент Академии наук, выдающийся человек, есть школы и институты его имени по информатике.

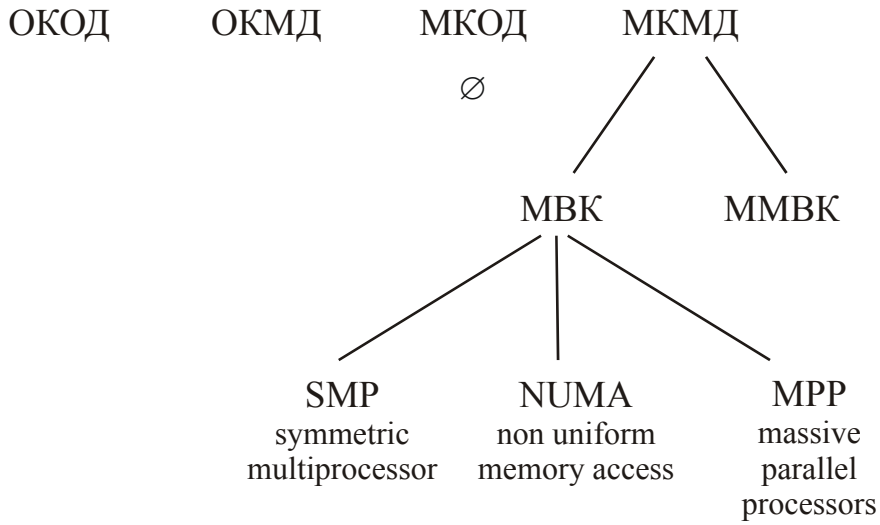
Ладно, теперь к делу.

Есть 5 групп вопросов. Первая группа — это вопросы по параллелизму как внутри вычислительной машины, так и по параллелизму многопроцессорному. Вторая группа — это средства поддержки работы операционных систем. Третья — это организация памяти. Четвертая — это тоже для параллелизма важно — это различные способы подключения, различные каналы связи с внешними устройствами. И пятая — это многомашинные комплексы.

Классификация архитектур вычислительных систем

Мы с вами говорили о том, что вначале были многомашинные комплексы, пока ещё не было многопроцессорных. Потом появились многопроцессорные, параллельно существовали многомашинные комплексы. Существуют многомашинные многопроцессорные — каждая из машин многопроцессорная, машины соединены между собой. Логическое подмножество многомашинных комплексов — это сети ЭВМ. Мы с вами изучать сети не будем.

Напомним классическую классификацию архитектур:



ОКОД — одиночный поток команд одиночный поток данных — один процессор. Что важно — одно устройство управления (УУ). Общее арифметико-логическое устройство (АЛУ). По-английски это — SISD (single instruction single data).

ОКМД (SIMD) — одна команда выполняется на разных процессорах над разными данными. Но одно УУ. Классический пример — ILIAC IV.

МКОД (MISD) — много команд, один поток данных. Попробуйте написать программу, которая бы имела операции +, −, ×, ÷, &, ∨, *сдвиг* только над одними данными (например, находящимися в ячейках памяти 3 и 6). Полная чушь! Такого не бывает. Этот класс пустой. Однако некоторые относят сюда машины с общей памятью: материал общий, над ним работает одна задача. Но у неё разные ветви, каждая ветвь идёт через своё УУ, это разные программы.

МКМД (MIMD) — подразделяют на 2 класса: многопроцессорные вычислительные комплексы (МВК) и многомашинные вычислительные комплексы (ММВК). МВК — это разные способы организации многопроцессорных комплексов, а именно SMP (symmetric multiprocessor), т.е. с общей памятью, в которой находятся данные, над которыми выполняются программы, выполняющиеся в разных процессорах. MPP (massive parallel processors) — это противоположный вариант — система, состоящая из многих узлов, в каждом из которых процессор и своя память, передача данных между ними идет сообщениями посредствам операционной системы. Т.е. именно из таких создаются комплексы на 1000 процессоров. И в промежутке NUMA (non uniform memory access) — вообще говоря, это идея общей памяти с точки зрения организации программирования. Узел так же как в MPP, процессор и память соединены общей шиной. Процессор может работать со своей памятью и с памятью другого узла. Ясно, что во втором случае процесс взаимодействия с памятью более долгий.

Только что прошла конференция, основным организатором которой был Московский университет в лице Вычислительного центра — «Интернет. Технологии параллельного программирования». Лидером во всех вопросах является Вычислительный центр МГУ. Надо сказать, что все ведущие фирмы учитывают эти моменты и как спонсоры они выступают здесь: IBM, Hewlett-Packard, Microsoft. Вещи очень актуальные, совершенно естественные.

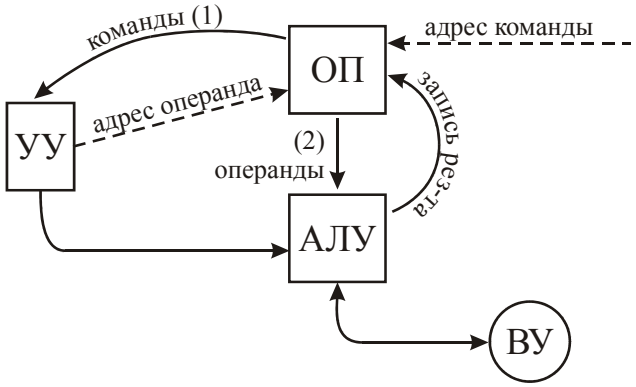
Скажу вам прямо — я крайне недоволен докладами в области образования.

Дело в том, что в министерствах образования и науки, электронной связи и информатизации, в этих министерствах очень сильна чиновная прослойка, которая заботится только о себе. Это чистое растаскивание государственных средств. Им выгодно объявлять всяческие конкурсы, программы с очень расплывчатыми названиями, расплывчатыми позициями в содержании, потому что под это дело склонные до халявы люди с удовольствием подбираются. А чиновники потакают этому делу, и подбираются люди, которые занимаются якобы научной деятельностью и выдают якобы результаты этой деятельности. К сожалению, на конференции, не только на эту, это начинает проникать.

На прошлой моей конференции меня развели с этими людьми: дали мне заведовать другой секцией. На другой лекции были доклады, но нашлись другие, которые разумно покритиковали эти доклады. Но что толку критиковать? Это же просто пролезающая в нашу среду порча! И против этой порчи надо решительно выступать.

Параллелизм работы основных устройств машины

Мы с вами рассмотрим первый уровень параллелизма — это параллелизм работы основных устройств машины, который был достигнут в мире к концу 50-х годов. Вначале рассмотрим схему выполнения обработки информации, которая была в первых машинах.



ОП — оперативная память, УУ — устройство управления, АЛУ — арифметико-логическое устройство. Попадая на УУ, операция вызывает из памяти тот операнд, который необходим. Первое действие — это выборка команды. Чтобы команду выбирать, подаётся адрес команды. Второе действие — подача результирующей информации. После того, как команда дешифрировалась на УУ, выполняется операция на АЛУ. В зависимости от системы команд, результат может либо сохраняться в регистре, либо записывается в ОП.

Итак, цикл: считывание команды — в это время задействована ОП, не работает УУ и АЛУ; когда считана команда, работает УУ, происходит дешифрация, передаёт адреса одного или нескольких операндов ОП, и эти операнды передаются АЛУ; потом работает АЛУ, а эти простаивают; ну и потом происходит запись результата (не обязательно в каждой команде).

Очень интересно, как подключались раньше внешние устройства (ВУ) — магнитные ленты, магнитные барабаны, магнитные диски, печать и т.д. — подключались к АЛУ. Почему так? Потому, что вместо того, чтобы делать дополнительный тракт (усложнять машину), уже есть тракт передачи, и когда команда обмена данными между ОП и ВУ попадала в АЛУ, она там застревала до тех пор, пока весь массив данных не будет записан или помещён в ОП. Если вам нужно тысячу чисел передать в ОП, то операция здесь и стоит, пока это всё не окончится. Это всё потому, что машины были большие.

Ясно, что нужно обеспечить параллелизм работы этих устройств, сделать так, чтобы они не простаивали. Главное, конечно, это работа АЛУ, где выполняются операции. Вот если АЛУ работает 100% — это значит, что структурная схема машины идеальна. По целому ряду причин это не получается, и мы с вами эти вещи увидим.

Итак, нужно как-то организовать работу этих устройств, чтобы они работали параллельно. Ну и чтобы операции передачи данных на ВУ или приём данных с ВУ тоже выполнялись независимо от работы центральной части машины. Таким образом, память могла работать параллельно на выдачу команд, на выдачу операндов, на приём результатов и на работу с ВУ.

Машины были разные. Были машины одноадресные (решение вопроса организации параллелизма работы основных устройств на примере БЭСМ-6 (см. Рис. 1 на стр. 19) — эта машина была одноадресная). Были машины двухадресные, трёхадресные, четырёхадресные, ну и машины, так называемые, безадресные (когда у вас все данные находятся в определённом массиве — стеке — и вы работаете только со стеком, у вас программа идёт лишь что-то записать в стек, снять со стека — работа со стеком). Производительность работы машины в условиях вычислительного центра, где решаются самые разные задачи, осталась всё-таки независимой. В каком плане? Можно себе представить такое вычислительное ядро, которое требует только одноадресные операции. Для разных операций более эффективно использование какой-то определённой адресности. Поскольку и те, и другие сущности встречаются в вычислениях — это всё равно, какую систему использовать. Просто какая-то машина с какой-то адресностью будет более эффективна на отдельных участках вычислений, менее на других. Поставите другую машину — будет наоборот. Конечно, какая-то есть задача, в которой сущность вычислений требует наиболее эффективной одно- или двухадресной системы, но это уже будет неспециализированная машина. Зависимость от того, какого класса машина: для широкого класса задач, универсальная или специализированная машина.

В своё время, в системе противоракетной обороны была использована машина М-40 с системой команд работы с данными с фиксированной запятой. Безусловно, быстрее, чем с плавающей запятой, от природы. Но алгоритм развился, приходилось работать с данными разного масштаба, и необходимость масштабирования (загнать в определённый интервал от нуля до единицы) привела к тому, что вычисления всё дольше и дольше выполнялись. Тогда была сделана машина М-50 с плавающей запятой для тех же целей противоракетной обороны. Тогда появился интересный лозунг: «Какая лучшая специализированная машина? Конечно, универсальная!»

Итак, как сделать? Сразу напрашивается решение. Нужно сделать память так, чтоб можно было параллельно читать, параллельно записывать по нескольким запросам по разным направлениям. А как это можно сделать? Разбить память на несколько работающих блоков. Как сделать так, чтобы быстро шли одна команда за другой? В машине БЭСМ-6 адресация шла по словам (есть машины, где адресуются байты) — 48-разрядным словам. Мы подаём адрес на соответствующий блок, и после какого-то времени работы блока команда начала выбираться на УУ. Тут же мы можем, чтобы следующую команду подать как можно быстрее вслед за первой, мы прибавляем единичку к счётчику адреса слова, ответ должен пойти в память. Куда? В этот же блок? Нет. Потому, что блок памяти (какой угодно памяти, которая была в 60-х годах или сейчас) всё равно имеет какое-то время цикла своей работы. Раньше, чем этот цикл не закончится, следующее обращение к нему подать нельзя. Стало быть, следующая команда должна быть в следующем блоке. И тогда со скоростью прибавления единички мы будем запускать один за другим блоки и, естественно, через какое-то время с такой же разницей во времени будут идти команды на обработку в дальнейших устройствах. Поэтому в БЭСМ-6 было 8 блоков (нумерация с 0 до 7). Сразу привыкнем, что все адреса в восьмеричной системе счисления. Конечно, чем больше степень параллелизма (или интерливинга — расслоение памяти), то, вообще говоря, в процессе работы обращения по разным направлениям за взятием операндов, запись результатов, команд, внешними обменами, они довольно заметно будут использовать (я не говорю 100%) эту возможность параллелизма просто из-за равномерности. Эта идея реализована во всех машинах начиная с 70-х годов и по сегодняшний день. Конечно, в современных машинах эта степень уже не 8 и даже уже не 16, а 32, 64 и т.д. Тогда сколько вы там потребностей обращения в память не имели, они, как правило, будут выполняться параллельно.

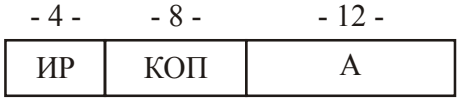
Вот — это первое вспомогательное решение параллелизма работы блоков памяти вот с таким распределением адресов: каждые следующие по адресу данные находятся в следующем блоке памяти по сравнению с данными с предыдущим адресом.

Кстати, память бывает двух типов, оба на букву «а». Сейчас мы с вами рассматриваем наиболее часто используемый тип — *адресная* память, где чтобы найти нужное, надо указать место в памяти. А есть, так называемая, *ассоциативная* память, когда вы находите по некоему признаку. Это означает, что в полном случае нужно осуществить перебор.

Теперь, команды будут поступать на УУ. Операции реализует АЛУ. Поскольку операции бывают разной длины по времени выполнения в АЛУ, соответственно, в работе УУ, которое передаёт туда операции, могут быть задержки по причине того, что там выполняется какая-то длительная команда. И поток команд из ОП нужно где-то сохранять. Для этих целей был организован буфер на 4 слова, назывался он *буферные регистры слов* (БРС). На самом деле это буфер команд. В каждом слове было 2 команды, итого, здесь было 8 команд. Т.е. если есть какая-то задержка, то буфер заполняется. Если совсем полностью заполнится, то совсем приостановится выдача новых адресов и считывание новых команд. Всё это, конечно, синхронизовано. Так или иначе, в БРС всё задержится, а когда УУ освободится, команды будут готовы. Этот буфер естественен, как буфер, который позволяет в случае задержки прохождения данных через УУ, которые возникают по многим причинам, по завершении сразу иметь команду для дальнейшего выполнения.

Прежде, чем этот буфер до конца (его функцию) рассмотреть, давайте посмотрим, что же за команды были в машине БЭСМ-6.

Команды были такие:



КОП — код операции (какую выбрать операцию), А — адрес и ИР — индексный регистр. Индексный регистр (ИР) нужен для того, чтобы избавиться от переменных команд. Как ситуация выглядела раньше? Пусть некоторая команда работает с массивом данных. Чтобы работать со следующим элементом массива нужно прибавить «1» к адресу, т.е. бралась сама команда из памяти, к ней прибавлялась «1» и записывалась на тоже место. Если цикл, то надо изменять все команды в цикле. Таким образом, цикл увеличивался в 2 раза. Это и по времени, ведь надо считать, записать и т.д. Нужно было сделать так, чтобы эти команды не менялись. А как брать следующие данные из массива данных? Очень просто: ИР заполнять информацией, и тогда истинный исполнительный адрес будет вычисляться следующим образом:

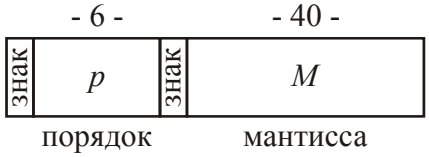
$$A_{\text{ИСП}} = \langle \text{ИР} \rangle + \text{А}$$

Таким образом, в конце цикла достаточно изменить ИР, иногда даже совмещается с командой перехода на продолжение цикла. В частности, в БЭСМ-6 такая команда и была. Она называлась КЦ (конец цикла). Эта команда проверяла значение ИР, и если он был не нуль, то она изменяла его значение (от положительной величины отнимала единичку, к отрицательной — прибавляла) и уходила на новый виток цикла.

Итак, в УУ были 15 ИР по 15 разрядов. Хватает? 15 вложенных циклов — вполне хватает, тем более в 60-е годы оборудование приходилось экономить. Поэтому 4 разряда вполне хватает, чтобы их (от 1 до 15) адресовать (0 — без индексации).

Хорошо. Сколько нужно под код операции (КОП)? 8 разрядов. Что такое 8 разрядов — это 256 кодов операций. Конечно же, столько не набиралось, но 7 разрядов не хватало.

В 60-е годы блоки памяти работали с циклом 2 мкс, среднее время выполнения операции в АЛУ — 1 мкс. Дальше совершенствовалась и ускорялась работа памяти и АЛУ, но память отставала от скорости выполнения операции. И вот такая приемлемая по скорости работы память была в промышленности доступна в несколько тысяч ячеек всего. И вот на машине БЭСМ-6 слово было 48 разрядов, чтобы в него можно было поместить число с плавающей запятой, где под порядок отводится какая-то величина и под мантиссу тоже:



где мантисса ($0 \le M < 1$) — 40 разрядов, под порядок (p) — 6 разрядов. Итого: 6 + 40 + 2 = 48 разрядов в слове позволяли получить достаточную точность (2^{-40}). А какой порядок? 6 разрядов, поэтому $2^{\pm 64}$. Поэтому диапазон достаточно большой, точность достаточно хорошая. В других машинах какие-то другие колебания, но в принципе вот это вот положение, где-то было 7 разрядов порядок, 8, 10 (это $2^{\pm 1024}$). Разрядность где-то увеличивалась, но бывало, и уменьшалась. Если взять машину 32-х разрядную (это была целая эпоха 32-х разрядных машин), то там точность немножечко поменьше. Для каких-то вычислений этого было достаточно, для каких-то — нет. И тогда в некоторых машинах были задействованы операции с удвоенной точностью. Т.е. у вас операция делалась не с 24-х разрядными, а с 48-разрядными. Помещалось 2 слова, ну и т.д. (первую половину в несколько байт, и вторую — байтная адресация). В некоторых случаях операции с удвоенной точностью не делали, а делали программно (в частности программы для любой машины сделаны с любой точностью).

Раз уж мы так говорим, давайте напишем общую формулу:

$$X = \pm 2^{\pm p} \cdot M$$

где p — целое, $0 \le M < 1$. Иногда бывает нормализованная мантисса, это когда для положительной мантиссы «1» в старшем разряде. Тогда $\frac{1}{2} \le M < 1$.

Повторяю, само количество данных в памяти было не велико — 4 килослова (в переводе на байты: 6 × 8 = 48 — в слове 6 байт). Где-то впереди — память 32 килослова. Сколько нужно разрядов при адресации словной, чтобы адресовать 32 килослова. Нужно 15 разрядов. Таким образом 8 + 4 = 12 да 15, получается 27 — ни два, ни полтора. Поэтому поместить одну команду в слове — это преступление. И решили, пусть здесь адрес будет 12 разрядов, а индексный регистр пусть будет 15-ти разрядный. Таким образом, вы складываете содержимое некоего указанного индексного регистра (он, естественно, 15-ти разрядный), и к нему прибавляется некая величина 12-ти разрядная, итоговый адрес получается 15-ти разрядный. Таким образом, появилась первая структура команд, которая даёт в сумме 24 разряда, и таким образом, в каждом слове появляется по 2 команды:

- 4 -	- 8 -	- 12 -
ИР	КОП	А

Понадобилось всё-таки иметь для некоторых команд полный адрес. И появилась вторая архитектура команд, которая выглядела следующим образом:

- 4 -	- 5 -	- 15 -
ИР	КОП	А

Индексный регистр (ИР) — 4-х разрядный, код операции (КОП) уменьшился до 5 разрядов, адресная часть — 15 разрядов. Ясно, что одну структуру от другой отличают значения старшего разряда КОП.

Таким образом, в УУ (см. Рис. 1) поступала команда на, так называемый, *регистр команд* (РК), где ИР, КОП и адрес (А). В УУ существовал *сумматор адреса* (СмА), где выполнялось вычисление исполнительного адреса. Сюда поступал адрес, по номеру ИР выбиралось содержимое ИР, и передавалось на СмА. Здесь происходило сложение, и на *регистр результата* (РР) поступал КОП и адрес исполнительный ($А_{ИСП}$). Вот такая работа УУ.

Как вы сами понимаете, некоторые операции при такой структуре здесь могли выполняться целиком до конца, не требуя задействования АЛУ. Например, операция передачи содержимого адресной части (А) в индексный регистр (ИР), операция сложения индексных регистров. Итак, выход здесь — адрес исполнительный ($А_{ИСП}$). Что $А_{ИСП}$ нам даёт? Если это адрес операнда, то его можно подать в ОП, и (не сразу, т.к. блок ОП может быть занят) хотя бы через микросекунду данные должны пойти в сторону АЛУ. Но блок может быть занят. Может возникнуть такая ситуация, когда мы и это делаем, и записываем что-то, и работаем с внешними устройствами (ВУ), тогда придётся подождать, и это будет уже дольше.

Для того чтобы операция в АЛУ встретила готовый результат, был поставлен некий буфер из 4-х ячеек, причём тогда не было интегральных схем, а были навесные элементы. Этот буфер назывался *буферный регистр чисел* (БРЧ). Когда запускался блок памяти (и если он запускался), этот регистр всегда должен быть свободен для приёма числа. На это требуется время, и мы договорились, что время может быть и заметным. Стало быть, готовую операцию, если АЛУ готово, подавать нельзя. Это бессмысленно: оно будет ждать, а мы хотим, чтобы не ждало, чтобы АЛУ работало непрерывно. Что нужно сделать? Ставился ещё один буфер, и вот эта операция попадала в этот буфер, одновременно с подачей адреса в память. Он назывался *буфер арифметических команд* (БАК). И вот этот буфер был на четыре команды вот таких вот: КОП и номер БРЧ. Таким образом, сюда приходили КОП и номер БРЧ, который зарезервирован для приёма операнда из памяти по адресу $А_{ИСП}$. Всё, эта команда попала в БАК. Впереди неё находятся ещё 3. Т.е. пока данные перейдут из памяти, эта команда постепенно подойдёт к АЛУ (на самом деле, никакого движения не было, просто переназначались регистры). И когда уже все эти операции выполнятся, и наша подойдёт к АЛУ, данные по адресу $А_{ИСП}$ будут считаны из памяти практически всегда, т.е. они будут готовы к

использованию. Можно сказать, что данные из памяти считались раньше, ну значит, эти данные будут ждать своей команды. АЛУ простаивать будет очень редко. Коэффициент полезного действия арифметико-логического устройства (АЛУ) БЭСМ-6 был порядка 80%. Не 100, но достаточно хороший коэффициент загрузки арифметико-логического устройства, оно работало 80% времени. Были какие-то моменты простоя.

Итак, за счёт БАК и БРЧ мы обеспечили параллельную работу оперативной памяти и арифметико-логического устройства. Если команда является не командой, подлежащей выполнению в АЛУ, а команда передачи управления, т.е. перехода на новую последовательность выполнения команд, то тогда этот адрес должен передаваться на *счётчик адреса* (СчАдр), и начнётся новая выборка. Всё, что набрали в БАК и БРЧ будет не нужно, и естественно будет задержка.

Раз так, то возникла идея: а что если (и это была одна из первых реализацией cache) сделать так, что если этот адрес того командного слова в памяти, которое, может быть, уже есть в БРС, т.е. если у вас есть короткий цикл, укладывающийся в 8 команд, а циклы задач линейной алгебры все такие, то можно сделать так: поставить 4 маленьких регистра (15 разрядов) и назвать их *буферные регистры адресов слов* (БАС), которые находятся в БРС (когда вы из памяти берёте слово в БРС, то в БАС помещаете его адрес, по которому взяли из памяти). И когда возникает код операции передачи управления, этот адрес сравнивается со всеми адресами в БАС — работает полностью ассоциативный cache. Если совпал, то тогда мы назначаем взятие отсюда. Время работы регистров во всех наших рассуждениях будем условно считать равным нулю по сравнению со временем работы памяти. Т.е. если у вас цикл в пределах 8-ми команд, вам никогда не будет нужно обращаться в память.

Моделирование показало, что на задачах линейной алгебры выигрыш получается 10-12% производительности по времени выполнения задач — это огромный выигрыш. Машина БЭСМ выполняла миллион операций в секунду в то время, когда предыдущие машины были порядка 100 тысяч операций в секунду, так это целая машина.

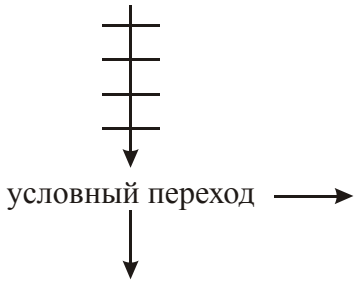
Т.е. возник кэш команд. Слова такого и не знали, так что их не было, но, как видите, сделали.

Между прочим у Лебедева в докладе «Пути развития советского математического машиностроения» он рассказывал про машину БЭСМ (БЭСМ-1 не называл, просто — БЭСМ), он тогда и эти вещи, и параллелизм, т.е. идеи машин 2-го поколения и 3-го, в этом докладе анонсировал.

Итак, всё вроде бы хорошо, циклы мы держим в БРС, не умещается, ну... Конечно, хорошо бы в этом случае иметь больше, указатель на более объёмное оборудование — нужен компромисс.

Очень интересный был момент: перед БЭСМ-6 практически в том же году была закончена работа над машиной «Весна». Главный конструктор — Владимир Константинович Левин, академик. Машина делалась в Минске. Систему команд делали так: ИПМ-овские «мудрецы», естественно прекрасные математики, говорят: «хорошо бы вот такую операцию ещё» — инженеры говорят: «будет сделано» — «а хорошо бы сделать вот таких операций» (для шифровки/дешифровки) — инженеры говорят: «будет сделано». И когда машина была сделана, в Минск приехали математики из ИПМ — огромный зал… множество стоек… — что это такое? Понимаете, должен быть разумный компромисс между аппаратурой и логической компонентой вычислительных машин (набором системы команд, программами и т.д.).

Итак, вот этих буферов хватает, если не делать запись результатов. Команды в случае циклов сидят в БРС, выборка операндов происходит даже раньше, чем приходит команда, АЛУ работает 80%. А почему? Знаменитая проблема: условный переход.



Вот идёт последовательность команд, а дальше команда условного перехода: либо сюда, либо туда, в зависимости от условия, например, положительным или отрицательным является результат предшествующей операции, пусто/не пусто после операции, положительный или отрицательный порядок, разные условия могут быть для выполнения этой проверяющей операции. Но команда предшествующая должна выполниться. И наша операция условного перехода, проходя здесь, не даёт возможности следующей операции поместить в БРС. Вот почему во многих машинах делается так: как только УУ поймало команду условного перехода, сразу начинают выбираться обе последовательности команд, чтобы они были готовы. Какая-то не понадобится, ну и чёрт с ней, зато готовая будет команда. В БЭСМ-6 такой роскоши не было, поэтому АЛУ сколько-то да простаивало. Вот это основной вклад в эти 20% простоя АЛУ.

Были и другие вещи, связанные с этим. Тогда, конечно, никто не говорил слова CISC (completed instruction set command). Вот когда сделали RISC (reduce instruction set command), тогда сказали: «А это что у нас было? Ах, CISC было…», тогда появилось это, а до этого не говорили. Вот в этом полном наборе операций существовали команды косвенной адресации, да ещё и многоярусные. Что это означает? Вот у вас есть операции:

$ИР_2$	косвенная адресация	A_2

$ИР_1$	КОП	A_1

Что делается? Адрес искался следующим образом: находился исполнительный адрес первой команды, фактически указывался адрес ячейки, из этой ячейки бралось 15 младших разрядов, и складывались так:

$$\left(\langle ИР_2 \rangle + A_2\right) + \langle ИР_1 \rangle + A_1$$

Т.е. у вас адрес был не прямой, а адрес получился косвенный. Вы указали адрес ячейки, в которой находится адрес вашего операнда. Сами понимаете, что можно использовать как «дом, который построил Джек»: адрес ячейки, в которой адрес ячейки, в которой адрес ячейки в которой находится операнд. Трансляторщики схватились за это урча и просто в восхищении, и программы запестрили вот этими командами. Больше того, транслятор в своей работе стал их использовать, оттранслированный код стал появляться с такими операциями. А что происходит: прежде чем получить $\left(\langle ИР_2 \rangle + A_2\right)$, её же надо считать из памяти (15 разрядов), значит нужно здесь задержать. А для этого нужно время. И вот вам второй компонент этих 20%.

Самое страшное: что делать с записью результата. Давайте запишем: «запись в ячейку α и считывание из ячейки α»:

Зп α
Сч α

Можно так написать? Можно. Так или иначе, есть команда, которая сейчас будет записывать в ячейку α, а сразу за ней пошла команда, которая считывает из α, а так как считывание операнда из памяти происходит раньше попадания команды в АЛУ, то вторая команда получит старое значение. И вся наша параллельность к чертям из-за одной команды записи. Что же нужно делать?

А нужно делать вот что: результат будет записываться в некий буфер, реально в БЭСМ-6 было 8 ячеек (см. Рис. 1). Они были названы *буфер записанных результатов* (БРЗ) и к тому же был здесь *буфер адресов записанных результатов* (БАЗ). И это получился второй кэш полностью ассоциативный.

Так вот, очень просто. Когда приходила команда записи, для неё всегда была одна свободная позиция. Так делалось, потому что самая старая выкидывалась в память. И тогда $A_{ИСП}$, вообще говоря, сравнивался со всеми адресами в БАЗ, потому что, может быть, уже только что недавно записывали. Если нет, то всегда идёт в БРЗ, там всегда есть свободная позиция. Как только команда попадала в УУ, то в БАК помещался соответствующий номер БРЗ. Итак, в БАК попадали КОП и номер БРЧ или БРЗ. Да, можно было сделать его общим и для чтения и для записи, но было сделано так. Когда мы придём к теме кэша, мы этот вопрос ещё подымем. Моделирование показало, что при такой организации выигрыш приличный (до 30%).

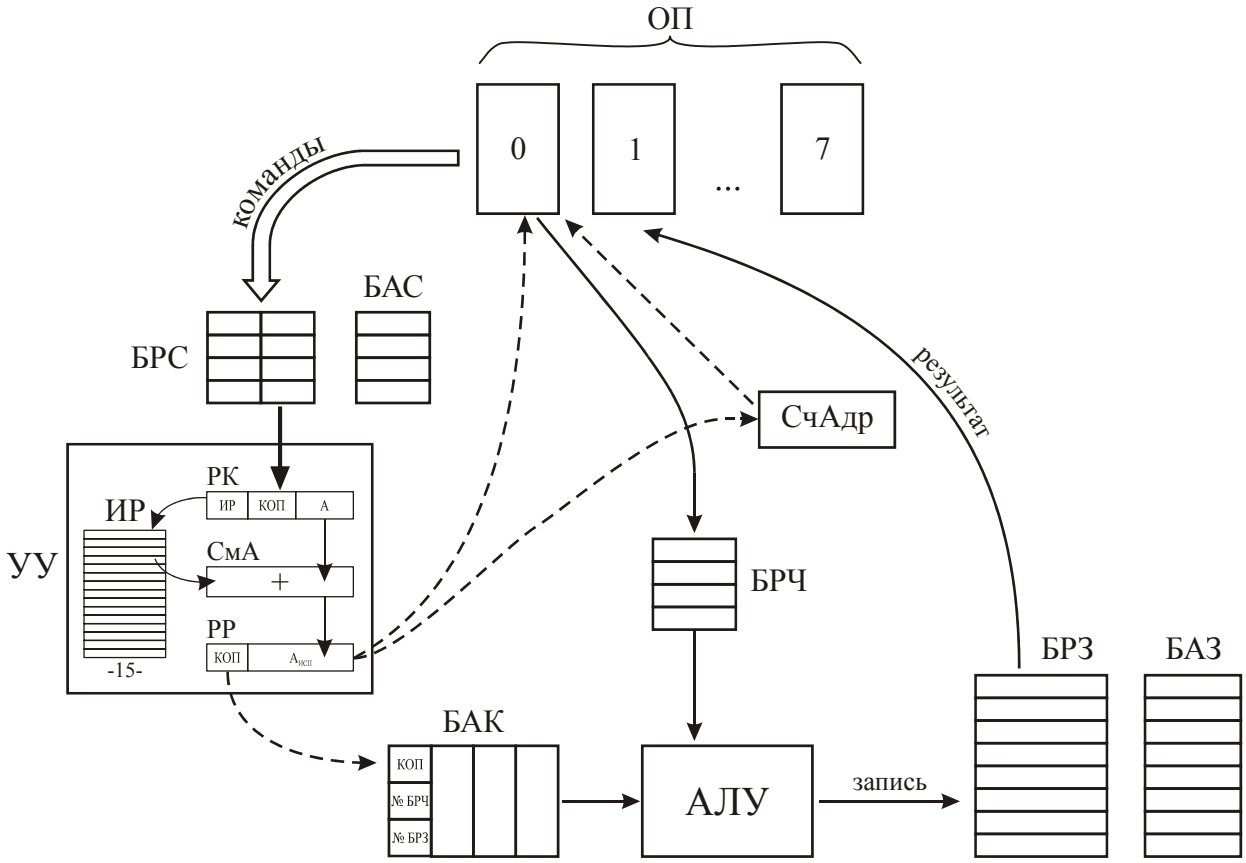
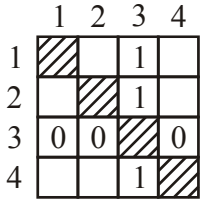


Рис. 1 Структурная схема БЭСМ-6

БАС	буферные регистры адресов слов
БАЗ	буфер адресов записанных результатов
БАК	буфер арифметических команд
БРЗ	буфер записанных результатов
БРЧ	буферный регистр чисел
РК	регистр команд
РР	регистр результата
СмА	сумматор адреса
СчАдр	счётчик адреса

Вот, собственно говоря, это всё, кроме одного: как же определяется старшинство? Оно определяется очень интересно. Рисуем игровую таблицу:



Допустим, что пишется в 3-й регистр. Тогда он делается самым молодым (по горизонтали ставим «0»). Против него (по вертикали) ставим «1». Кто набирает все «1» (в строке), тот выкидывается. И при этом это обмоложение происходит при любом совпадении, будь то по записи (если не нашло, то свободная становится самой молодой), по считыванию (пришло считывание, и тот регистр, из которого будут браться данные для операции, когда она подойдёт, делается самым молодым — раз данные нужны, значит, скорее всего, понадобятся ещё).

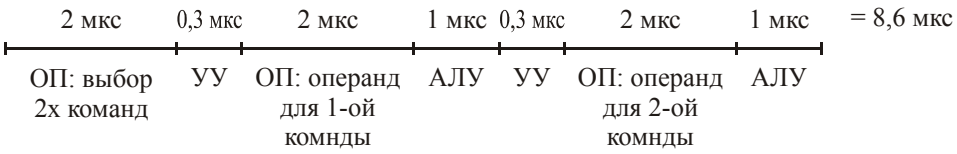
Таким образом, у нас полностью завершена структурная схема машины, и всё в ней работает, обеспечивая максимальную загрузку АЛУ. Как сами понимаете, при формульном счёте мы тоже не лезем за операндами в память (берём из БРЗ). За счёт устройств промежуточного хранения информации активных и неактивных удалось организовать параллельную работу основных устройств: оперативной памяти, устройства управления и арифметико-логического устройства (в первую очередь для обеспечения полной загрузки).

Вы знаете, дело в том, что приходили сведения, что вот Motorola сделала процессор такой-то. Господи, почему же они так сделали? (как бы становясь в позицию игры Бендера с васюкинцами: «Что мне делать?» — «Сдавайся…»). На самом деле всё очень просто: они в чём-то ошиблись, и им надо было переделать всё. Переделать — это значит задержать по времени выпуск процессора, выпуск машины и т.д. Они сделали заплату. И вот этот весь такой … самое совершенное, что только можно придумать! А оказывается, что это не совершенное, это просто вынужденная заплата, и вовсе антисовершенное решение. А здесь всё чётко предельно ясно. Когда появилась машина Cray (другой уровень параллелизма, а именно уровень параллелизма исполнения самих операций), Лев Николаевич сказал: «Батюшки! Да это ж БЭСМ-6!» Конечно, ничего похожего нет, но ясность и чёткость структуры позволили ему это сказать. Вот смотрите, что мы здесь имеем? Здесь имеем, конечно, ярко выраженный конвейер команд, который Сергей Алексеевич называл «водопроводом»: где-то задержка, другое не течёт, когда клапан открыли, кранчик открыли, всё потекло дальше. Это называется «конвейер» или «магистраль» для команд. Вот смотрите: одна команда выполняется, четыре стоят в очереди, это здесь обрабатывается, здесь какие-то выбираются. Реализован полностью довольно активно конвейер команд. Это выполнение команд на разных стадиях: эта здесь, эта здесь, эта вот здесь. Ну а конвейер внутри операции (параллелизм операций) мы рассмотрим позднее (это машина Cray).

Был задан вопрос относительно этих самых злосчастных условных переходов. Об этом, правда, уже говорилось, что выбираются из памяти оба потока команд: и те, которые идут после условного перехода, и та ветвь, на которую указывает условный переход. Но для этого действительно нужно анализировать этот условный переход сразу, как только выбирается команда. Помните, мы смотрели как пример машину БЭСМ-6: команды поступают в буфер. Так вот на входе буфера, ещё задолго до того, как она попадёт в устройство управления и начнёт дешифрироваться, и выясниться, что это команда условного перехода, нужно обнаруживать команду условного перехода, т.е. производить предварительную дешифрацию. Но это ведь сложно, потому что ведь нужно знать исполнительный адрес команды, а перед этой командой могут быть команды изменения индексных регистров. Вот такой предварительный достаточно сложный анализ в современных процессорах действует, по-разному. По разным причинам может производиться сложный предварительный анализ, прежде чем запустить команду уже в полное исполнение. Так что это очень не простая вещь. Условный переход — это головная боль для разработчиков. Но вопрос был задан на эту тему, так что ещё раз этот момент подчеркнём.

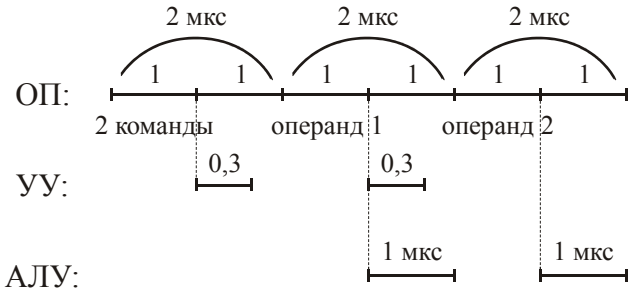
Теперь, чтобы закончить с параллелизмом основных устройств машины, давайте посмотрим, а какой эффект вносит вот этот структурный параллелизм, который мы рассматривали.

Пусть работает некая машина, которая имеет несколько физических параметров, т.е. время работы блоков памяти, время выполнения операций алгоритмов на логическом устройстве, сколько времени работает УУ, но нет никакого параллелизма.



2 мкс (такое тогда было время) работал блок памяти. За это время выбирается одно слово, содержащее 2 команды. Затем у нас работает устройство управления, которое работает 0,3 мкс, образуя исполнительный адрес команды, которая требует считывание операндов из памяти. Затем, когда выяснилось, что нужно считывать операнд (перебираем какой-нибудь массив, понятно, что он не поместится в кэш, хотя в этом случае кэша не будет, поскольку нет никаких буферов). Итак, 2 мкс тратится на считывание операнда для первой команды. Затем выполнение операции — работает АЛУ — 1 мкс, поскольку БЭСМ считала 1 миллион операций в секунду, что значит 1 MIPS. Не 1 Мфлопс, а 1 MIPS. Что касается Мфлопс, то было 0,3 Мфлопс, поскольку всегда при выполнении разных счётных вычислительных задач, количество операций +, –, ×, ÷ составляет приблизительно 30% от общего числа операций (статистика такая по Вычислительному центру: ± — 16%, × — 15%, ÷ — 1-2%, всё остальное — обслуживающие операции). В среднем команда выполняется за 1 мкс. Итак, мы закончили выполнение одной команды, у нас уже готова вторая команды. Мы её обрабатываем в устройстве управления, снова надо брать операнд (операнд для второй команды), и, собственно, выполнение операции в АЛУ. Теперь всё складываем: 8,6 мкс. В машине, которая учитывает параллелизм, это будет 2 мкс, но поскольку КПД 80%, значит, будет немного больше, чем 2 мкс. В любом случае, соотношение примерно 4 : 1 только за счёт структурного параллелизма.

Давайте чуть-чуть поиграемся вот в каком варианте: я буду с памятью параллелить работу УУ и АЛУ, саму память параллелить не буду. Дело в том, что, вообще говоря, цикл памяти даже и сейчас в интегральных схемах имеет 2 части. Если мы имеем в виду считывание операнда, сам операнд появляется при запуске некоторых блоков памяти, где нужно взять этот операнд, несколько раньше, чем заканчивается цикл. Вот память на магнитных сердечниках (маленькие ферритовые колечки, диаметром порядка десятых долей миллиметров, через которые проходили провода для записи, для считывания, для регенерации) — за 1 мкс происходило считывание командного слова. При этом разрушалось состояние намагниченности сердечников, и с некоторого промежуточного регистра он снова записывался, на что требовалась ещё 1 мкс. Короче говоря, через 1 мкс появлялись 2 команды. Можно было запускать УУ. Вот оно начало свой цикл 0,3 мкс. И дальше оно уже знает, где нужно брать операнд. Но так как память у нас не запараллелена, то мы ждём. Запускаем цикл считывания операнда. Но операнд приходит быстрее — за 1 мкс. Раз он пришёл, мы можем начать работу АЛУ. Таким образом, следующая команда у нас есть, и можно было бы её запустить параллельно после выбора первого операнда.



Но всё равно понадобится второй операнд. Что у нас получилось? Я запараллелил работу памяти, но внутри памяти не запараллелил работу отдельных блоков, а также УУ и АЛУ. Я получил 6 мкс. Грубо говоря, здесь 3 : 1. Здесь всё-таки ясно, что нужно обязательно иметь интерливинг памяти и устройства промежуточного хранения информации между основными устройствами. Вот какое значение имеет структурный параллелизм.

На счёт феррита вспоминается 2 интересных случая. Приезжали к нам американцы, ну и в Институте точной механики и вычислительной техники им показывали ферритовые сердечники. Американцы ходили, смотрели: «Да, очень интересно!» И рукой незаметно облокачивались, рука жирная — сердечники прилипали, и так незаметно в карманчик. Кстати, наши тоже самое делали, когда к ним ездили. Понимаете в чём дело, психологический момент здесь тоже важен.

Вот ещё один такой случай. Дело было в государственном масштабе. Мы с вами в скорости будем рассматривать машину Эльбрус-2. В MIPS, учитывая многопроцессорность, её производительность порядка 100 миллионов операций в секунду (100 MIPS). И вот, эта машина планировалась в дальнейшем, но ещё далеко было до отработки полностью структурной схемы. Так вот в печати (сначала в «Московской правде», потом в «Правде») было написано, что уже поставлено на поточное производство. Т.е. ну чистейший обман! Сначала были «термины Бурцева», потом статья в «Правде». Это было сделано совершенно сознательно. В конечном счете, машина появилась, но далеко не в то время, когда было объявлено. Ну и другие подобного рода были дезинформации. Они привели к тому, что действительно американцы как-то ощутили, что хорошая разработка, значит будут иметь русские (Советский Союз) машины. Раз будут иметь машины, значит всё: и расчёт ядерных вещей, бомбы, самолёты, ракеты. И это привело к определённому соглашению в области вооруженных сил. Американцы огромный доклад представили правительству и президенту на предмет срочного выделения огромного количества средств на дальнейшее развитее вычислительных систем в интересах конкурентной способности и безопасности Америки. Т.е. это всё было не случайным. Это сознательное было решение высших инстанций поместить вот такую существенно опережающую на несколько лет информацию.

Итак, мы с вами рассмотрели параллелизм работы, а сейчас попытались его ярко себе представить «что было, если бы не было». Получается так: память запараллелена, осталось запараллелить два других основных устройства машины, т.е. арифметико-логическое устройство и устройство управления. Первая часть получилась, прежде, конечно, столько, сколько нужно увеличилась параллельность работы блоков памяти. Распараллелим АЛУ — потребуется больше операндов в единицу времени, больший поток записывания результатов. Поэтому, конечно, это тоже будет с этим сопрягаться. Главное, что усилия пошли на распараллеливание работы арифметико-логического устройства. Что касается устройства управления — это очень сложно распараллелить. Только что мы с вами говорили об условном переходе, что команды условного перехода можно как-то ловить, но не всё так просто. Т.е. распараллелить работу УУ очень сложно, разве что сделать несколько (увеличение объёма аппаратуры — само собой, но миниатюризировались схемы, дешевели, и увеличение объёма стало вполне разумным). Можно взять несколько ветвей программы, если быть уверенными, что они независимы (выполняются над независимыми данными) и запустить на разных УУ. Но это не так активно в мире развивалось, и мы это не будем затрагивать. А вот АЛУ напрашивалось его распараллелить.

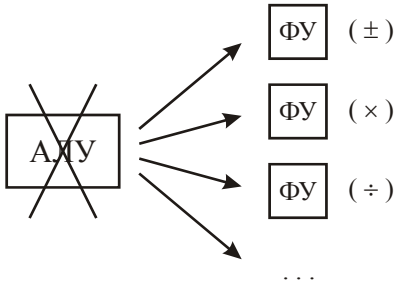
Параллелизм работы АЛУ

Распараллеливание АЛУ напрашивалось по двум направлениям. Первое — разбить его на несколько *функциональных устройств* (ФУ). Тогда и появилось название «функциональные устройства», т.е. выполняющие отдельные функции или операции, отдельные операции и группы операций. Например, устройство «сложитель/вычитатель» (понятно, что это родственные операции полностью), устройство «умножитель», устройство «делитель», устройства

логических операций (конъюнкция, дизъюнкция). Универсальное арифметико-логическое устройство, где схемы позволяли выполнять много разных операций, развели.

Каждое устройство стало попроще, но эти устройства можно запустить параллельно, если операции на них будут выполняться с независимыми данными. Если у вас выполняется операция сложения и получается результат, и он используется в следующей операции умножения, то, конечно, следующая операция выполняться не может до тех пор, пока не закончится операция сложения. А если операция умножения работает с данными, не связанными с результатом предыдущей операции сложения, то пожалуйста — очень хорошо.

И вот появилось разделение на отдельные функциональные устройства. Итак, АЛУ разбивается на отдельные функциональные устройства (ФУ).



Этот шаг постепенно прививался в разработке вычислительных машин. В начале 70-х годов активно пошла серия машин CDC 6xxx (Control Data Corporation). И уже в старших моделях этого ряда стало это введение появляться: стал выделяться сложитель/вычитатель, умножитель и делитель.

Ещё раз про фирму Control Data Corporation (CDC): мы говорили и о том, что в Советском Союзе организацией, которая разрабатывала самые высокопроизводительные машины, был Институт точной механики и вычислительной техники. В Америке тоже: фирма IBM — это была фирма, производящая огромное количество машин на весь мир, но были и фирмы-гранты. Самый грант был Control Data Corporation. И не удивительно, главный конструктор этой фирмы был Сеймур Крэй до того момента, пока он сам не выделился и организовал фирму Cray Research, и в 1976 году вышла машина Cray-1. С этого момента началась эра суперкомпьютеров (появилось это название). На самом деле, эта машина позволяла осуществлять при двух конвейерах существенно больше, чем 100 миллионов операций в секунду (машина CDC 6000 — всего несколько миллионов операций в секунду) во многом за счёт вот этого параллелизма: разбиения на многие функциональные устройства. Считается, что именно машина Cray и олицетворила, полностью реализовывала подход разбиения на множество функциональных устройств, объединяемых в различные группы — это был первый шаг. И второй: была достигнута конвейерность выполнения операций в каждом из этих функциональных устройств.

Если мы будем рисовать схему, будем говорить о некоем вертикальном параллелизме, а именно параллелизме работы ФУ, и горизонтальном параллелизме выполнения операций в каждом из этих устройств в конвейерном режиме. Всё это дало весьма большой выигрыш производительности. Это следующий этап развития параллелизма — параллелизм выполнения операций. На БЭСМ-6 мы с вами отметили в конце нашего рассмотрения активно реализованный конвейер команд, то здесь мы с вами увидим конвейеры (их много) в самих функциональных устройствах. Так что такие гранты были. Ещё к грантам можно отнести машины фирмы Boroughs. А рабочими лошадками были фирмы IBM, DEC (Digital Equipment Corporation), целый ряд других фирм. Таких машин выпускалось не так много, но они все были самой высокой производительности.

Второе, о чём будет разговор — это о конвейере. Сейчас эти компоненты с вами проговариваем, а потом посмотрим саму структурную схему машины Cray, увидим там достоинства и недостатки, и посмотрим, каковы же были решения, преодолевающие эти недостатки.

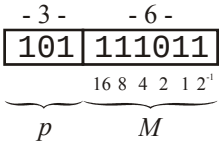
Ученики и последователи Сеймура Крэя, которые остались в фирме Control Data Corporation, сделали машину Cyber 205. И вот очень долго в мировой литературе в самых разных журналах активно обсуждалось, для каких задач удобен Cray, для каких задач удобен Cyber

205, потому что там достижение высокого параллелизма работы ФУ достигнуто было совершенно другим способом, чем в машине Cray. Можно сказать, что объединение этих двух тенденций через какое-то довольно продолжительное время очень интересным образом реализовалось в отечественной разработке модульно-конвейерного процессора (МКП), который был реализован в 1990 году, и машина уже была сдана государственной комиссии, и потом даже работала над счётной задачей в Челябинске. И вот был сделан этот МКП, и было сделано ещё несколько машин, одна для Московского университета, и всё — на этом всё завершилось. Эти несколько машин не были запущены. Точно так же несколько машин «Красного Крея». Началось новое время, начались новые взаимоотношения, и всё было разбито.

Давайте сложим 2 числа. Чтобы все действия получились, давайте возьмём

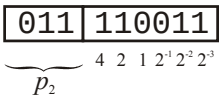
$$29\tfrac{1}{2} + 6\tfrac{3}{8} = 35\tfrac{7}{8}$$

Давайте представим в двоичной системе с плавающей запятой: $X = \pm 2^{\pm p} \cdot M$, где p — целое, M — нормализованное: $\frac{1}{2} \le M < 1$. Итак, вот у меня есть разрядная сетка: 3 разряда под порядок и 6 разрядов под мантиссу. Запишем первое число:



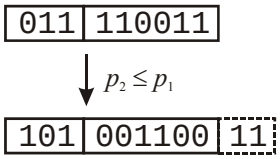
Как это легко считать? Я предлагаю вам поступать таким способом: нужно брать степень двойки, первую ближайшую, больше этого числа. В нашем случае — это $32 = 2^5$, значит $p_1 = 5$. Дальше, первый разряд в мантиссе есть половинка от 32, т.е. 16, дальше — 8, и т.д.

Теперь второе число:



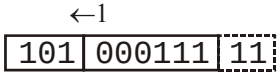
Мы представили те числа, которые намериваемся сложить. Что нужно сделать? Наверное, складывать мантиссы можно только тогда, когда $p_1 = p_2$. А если они не равны, тогда нужно привести к одному порядку. При этом приводится к большему порядку. Таким образом, второе число надо представить в разрядной сетке так, чтобы порядок его был равен 5.

Пишем порядок 5, тем самым, умножая это число на $2^2 = 4$. Но если вы умножаете на 4, то должны и разделить на 4. Чтобы разделить мантиссу на 2, нужно сдвинуть её на один разряд вправо. Чтобы разделить её на 4, нужно сдвинуть на 2 разряда вправо. И тогда число будет тем же самым, просто другое представление.



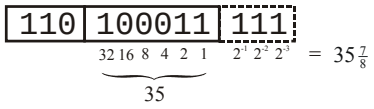
Две последних единички вылетают за пределы разрядной сетки. Но мы пока их никуда не денем, они находятся в регистре младших разрядов. В БЭСМ-6 есть такая операция: выдача младших разрядов на сумматор. Самих операций с удвоенной точностью в машине нет, а вот программно это можно сделать. Но мы не будем говорить о выполнении операций с удвоенной точностью. Будем говорить, как получаются нормальные обычные операции. В машине БЭСМ-6 мантисса 40 разрядов, т.е. цена младшего разряда 2^{-40}.

Теперь давайте посмотрим, что здесь будет происходить. Нам нужно теперь складывать вот эти два числа. Складываем:



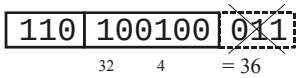
Перенос пошёл — вылезает единичка за пределы разрядной сетки мантиссы. Мантисса не может быть больше единицы, а она становится, как бы, больше единицы. Этого нельзя

допустить, таким образом, мы приходим к некоторой операции, которая по-разному называется: либо «нормализация вправо», либо «денормализация». Мантисса стала больше единицы, мы её делим на 2, а значит, к порядку прибавим 1. Итак, получаем:



Мы с вами, таким образом, произвели выравнивание порядков (к большему), затем произвели сложение (при этом потребовалась нормализация вправо). И мы имеем итоговый результат: $35\frac{7}{8}$. Всё совершенно верно, за исключением одного: если мы будем писать это число в память вот так, ничего с ним не делая, то у нас получится, что в память уйдёт 35 (младшие разряды, не вошедшие в разрядную сетку машины будут потеряны), вместо $35\frac{7}{8}$ — мы здорово не добрали.

Существует действие «округление», которое в разных машинах выполняется по-разному, есть самые разные к этому делу подходы. Один из простейших давайте посмотрим. Мы прибавляем единицу к самому старшему из вышедших разрядов. Если б здесь был 0, то это никак бы не оказало влияние на мантиссу. Поскольку в данном случае у нас получится перенос, то получим:



Мы получили величину 36 — это уйдёт в память. Это вот один из вариантов округления. Были разные варианты округления: методом наложения единицы на младшие разряды или есть хотя бы что-нибудь, вышедшее направо — это всё зависит от класса задач. И вот про то, как сделать в машинах округление, написаны кандидатские и докторские диссертации. Нехорошо иметь накапливающуюся ошибку, иначе у вас просто метод не будет работать, в вычислениях у вас будут неправильные получаться результаты. Поэтому к этим вещам нужно относиться очень внимательно. Например, есть такой институт, который называется ИТЭ — Институт теоретической электромеханики. Он получал все машины, которые только были. Больше того, кое-что делал сам по документации завода. Что они делали? Во всякой приходящей машине они откусывали сделанную в ней схему округления и паяли свои по собственному уму и разуму.

Смысл такой: сколько частей действий? Первое — выравнивание порядков, второе — сложение мантисс, третье — нормализация, четвёртая — округление. И эти операции можно выполнять в специальном месте, а результат передавать от одного этапа к другому в конвейере — устроить конвейер внутри функционального устройства. Разбить выполнение операции на 4 этапа. Т.е. у вас 4 двухместные операции будут выполняться одновременно в устройстве сложения, но на разных стадиях исполнения.

Допустим, что устройство чистое (ещё не занято, не работает). Приходит сюда первая операция — сложение x_1 и x_2. Естественно, работает первая стадия (ступень, станция конвейера — разные есть названия). Она отработала (будем говорить) за один такт. Затем на следующем такте производится сложение мантисс x_1 и x_2 с выровненными порядками, и в это же время на освободившемся оборудовании выравнивания порядков выполняется операция сложения x_3 и x_4, т.е. выполняется выравнивание порядков для x_3 и x_4. Ну и так далее. И когда конвейер заполнится, и все эти операции могут быть поданы в каждый такт с независимыми данными (ясно, что если операция использует результат предыдущей, то никакого конвейера не будет), то тогда вы разогнали (или загрузили, или запустили) этот конвейер, и он начал работать. Таким образом, все операции занимают 4 такта, а результат первой операции вы получили через 4 такта, а результат следующей ещё ровно через такт. И если у вас каждый такт подаётся операция с независимыми данными, то после загрузки конвейера вы будете на выходе получать результат каждый такт. Т.е. за счёт вот такого разбиения оборудования на 4 независимо работающие группы схем, каждая из которых выполняет определённую стадию выполнения операции, вы ускоряете работу всей операции. Вот в этом суть конвейера.

Если это операция умножения, то же самое, есть определённые шаги. Для операции умножения какая будет первая стадия? Сложение порядков: $p_1 + p_2$. Сложение порядков, умножение мантисс, нормализация, округление. Для деления: вычитание порядков, деление мантисс, нормализация, округление. Посмотрим, как это было реализовано в Cray.

Чтобы подряд шли операции $a + b$, $c + d$, $e + f$ и т.д. и т.д. — вряд ли. А ведь на вход конвейера должна приходить с готовыми данными каждый такт, тогда конвейер будет полностью загружен, тогда получим тот самый эффект. Когда это будет так? Естественно при обработке массивов данных. При сложении двух массивов данных — сложение двух векторов. Понятие вектора очень многозначно. И если у вас есть два рабочих набора, и вы как-то можете обеспечить взятие соответствующих компонент двух векторов и кидать на вход операции сложения, то конечно, в этом случае имеем $A + B$, где A — это вектор и B — это вектор (набор данных). Тогда мы эффективно используем конвейер.

Но такие задачи есть, много задач. И в расчёте на такие задачи была создана машина Cray. И потому она и была названа «векторно-конвейерная машина», т.е. машина, в которой вот эта конвейерная эффективность используется в действиях работы над векторами. Представьте себе некую векторную операцию:

$$\frac{A+B}{C \cdot E}$$

Уже эти две операции (сложение и умножение) можно выполнять параллельно. Уже 2 ФУ работают параллельно, а внутри используется конвейер обработки. Так что вполне в работе с массивами данных можно задействовать и работу параллельно ФУ.

Векторно-конвейерная ЭВМ: Cray

Итак, как был устроен Cray (см. Рис. 2): параллельно работают следующие ФУ. Была группа векторных функциональных устройств. Чтобы иметь возможность постоянно подавать в момент запуск операции до её окончания на вход этих векторных устройств компоненты векторов, эти вектора помещались на, так называемые, *векторные регистры*, которые назывались от V_0 до V_7. Это условное название «векторный регистр». Каждый регистр, на самом деле, это была группа регистров. Здесь было 64 64-х разрядных регистров (полноразрядных). Т.е. всего было 64 на 8 регистров, объединённые в 8 групп, которые адресовались как V_0, V_1, … , V_7, и эти адреса можно было указывать в командах. Транслятор подготавливал передачу данных из ОП заранее на эти регистры. Такие групповые передачи планировал транслятор до того, как начнётся векторная операция. Он загружал эти регистры и, естественно, параллельно с выполнением каких-то других в этот момент операций. Записанный вами на языке программирования алгоритм он препарировал и подготавливал выполнение всех этих операций и заносил в программу эти служебные (вспомогательные) действия, которые запускали групповые операции, подготовленные транслятором операции, проходящие через УУ. Буфер команд (БК) был довольно большого размера, и из этого буфера команд (БК) команды переходили в УУ и выходили наружу с указанием КОП и регистров. Например, операция:

$$+ \ V_2 \ V_3 \ V_7$$

Это значит, что берётся операнд из регистра V_2 (все 64 компонента), берётся операнд из регистра V_3, и результат записывается в V_7. При этом эта операция только тогда сдвинется с регистра результата (РР), т.е. начнёт выполняться на ФУ, когда оно готово, и на регистрах V_2 и V_3 готовы операнды, т.е. до этого прошло считывание данных и загрузка этих регистров. Когда команда приходит на уровень регистра результата (РР), сразу опрашивается V_2: готово? — готово, V_7 свободен, устройство свободно, и операция вылетает на ФУ и пошла. Начинают выбираться данные из V_2, V_3, результат записываться обратно в V_7. И всё это в конвейерном режиме.

За сколько тактов это всё произойдёт? Считаем, что время взятия данных с регистров равно нулю. Поэтому как только операция пришла, она сразу началась. Сколько понадоби-

лось времени на первую операцию? 4 такта. Чтобы получить каждый следующий результат, записывающийся в V_7, нужен ещё один такт. Итого: 4 + 63 = 67 — операция сложения двух векторов по 64 компонента выполнилась за 67 тактов. Если бы не было конвейера, и каждая операция выполнялась бы за 4 такта, все 64 операции выполнились бы за 256 тактов. Конвейерность дала 67. Получили весьма заметный выигрыш.

Я вам говорил, что понятие «суперЭВМ» возникло в литературе в 1976 году с появлением машины Cray. Но как определяли суперЭВМ — наиболее высокопроизводительная машина на данный момент. Тогда решили так: 100 миллионов операций — это уже вычислительный комплекс. Так же, как космос как вычисляется? Условно будем считать 100 км. Так же и тут тоже.

Ещё одно определение, которое мне больше всего нравится: супермашина — которая умеет решать те задачи, которые на порядок менее сложны тех, которые уже нужно решить.

Самая мощная супермашина всегда занимает размер одного большого машинного зала. Так было, так есть и так будет. Естественно, количество оборудования (единицы электронных схем) увеличивается, но они становятся меньше. Поэтому объём оборудования сохраняется в пределах одного большого машинного зала.

Вот, мы с вами видели, как выполняется. Итак, операция полетела и началась работа этих 67 тактов. Если у нас следующая операция умножения:

$$(+)\ V_2\ V_3\ V_7$$
$$(\times)\ V_1\ V_0\ V_6$$

и она попадёт сюда ровно через такт, через такт будет запущено умножение, и два ФУ пойдут в параллель. Значит, нужно транслятору так назначить регистры и передать на них данные, т.е. осуществить машинно-зависимую оптимизацию.

Что ещё он должен сделать? А ещё он должен сделать следующее. Представьте себе, что у вас по вашему алгоритму пойдёт вот такая команда:

$$(+)\ V_2\ V_3\ V_7$$
$$(+)\ V_7\ V_4\ V_5$$
$$(\times)\ V_1\ V_0\ V_6$$

Ясно, что вторая команда не пойдёт, пока вообще не кончится операция сложения. Хотя бы потому, что устройство сложения уже занято. И будет она, голубушка, сидеть здесь 66 тактов по крайней мер. И будет как собака на сене: сама не есть и другим не даёт. Которые, на самом деле, могли бы вычисляться. Какова задача транслятора? Переставить эти вещи. В алгоритме у вас так, но ведь можно переставить хотя бы так:

$$(+)\ V_2\ V_3\ V_7$$
$$(\times)\ V_1\ V_0\ V_6$$
$$(+)\ V_7\ V_4\ V_5$$

И вот эта перестановка команд — это шедъюринг — перестановка команд. Этим тоже занимается оптимизирующий транслятор.

Итак команды идут, запускают векторные функциональные устройства, они работают и т.д. Нужно уметь так подготовить набор векторных команд и так заранее передать эти вещи, чтоб всегда векторные ФУ были бы загружены.

Ещё были скалярные функциональные устройства. И была группа адресных функциональных устройств. И соответственно был набор регистров. Было 8 S-регистров (скалярных). И если у вас была операция над скалярами, она уходила на скалярные ФУ, где было указано от s_0 до s_7. Но это уже регистр всего лишь 64 разряда. Операнды брались отсюда, и результат записывался сюда. Все ФУ были конвейерные. Конвейерные хорошо для векторных, а со скалярными как? Просто идут операнды:

$$(+)\ a\ b\ c$$
$$(\times)\ k\ l\ m$$
$$(+)\ p\ t\ n$$

В этом случае, т.к. на следующий такт пришла операция другая (умножение, а не сложение), то один такт сложение будет простаивать. Будет недозагрузка совершенно явная.

Кстати, очень хорошие операции были такие: умножение вектора A на константу c.

Что такое адресные ФУ? Были так называемые А-регистры, тоже 8 штук. А-регистры отличались тем, что они 24-х разрядные — специально для адресной арифметики. В БЭСМ-6 можно было складывать адреса, а умножать — нет. Здесь же есть вся адресная арифметика, правда через некий промежуточный буфер из 64-х регистров, который назывался В-регистр. Для S-регистров тоже был сделан буфер тоже на 64 регистра — Т-регистр.

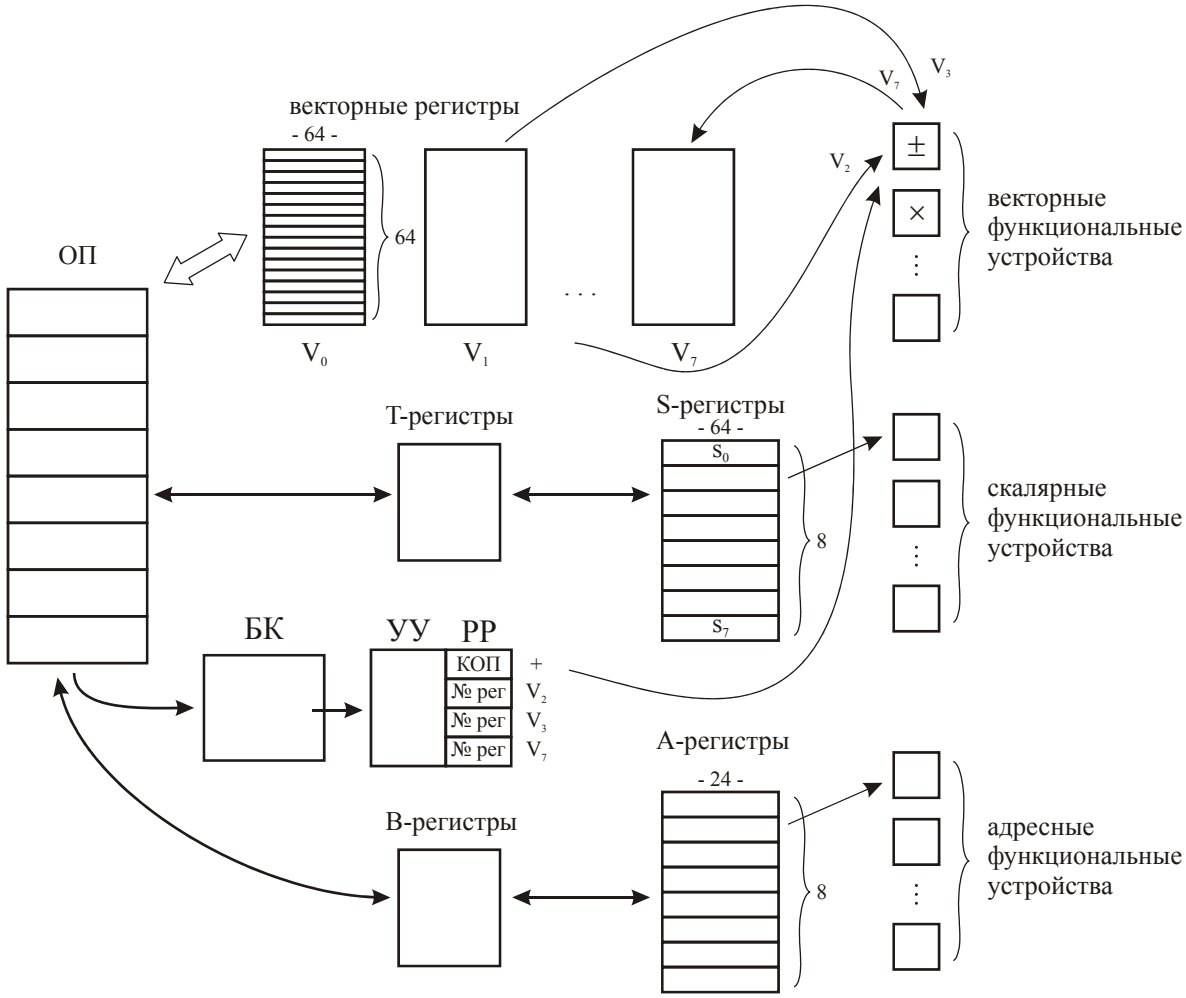


Рис. 2 Структурная схема Cray

Такова была идея Cray: иметь быструю регистровую память и возложить на транслятор обязанности так располагать данные на регистровой памяти, чтобы вызываемые форматы команды находили во время всегда себе «пищу» для выполнения. Понятно, что две команды при одном векторном ФУ сложения $A + B$ и $C + D$ надо их как-то разводить в программе, потому что сложитель был один. Это тоже задача транслятора была.

Посмотрим, какая здесь по сравнению с БЭСМ-6 разница в модели программирования: там была модель памяти, здесь модель регистров. Конечно, когда вы пишите на языке вы сразу так не очень, тут уже транслятор имеет это в виду. Там не было прямо адресуемых регистров. Но именно в расчёте на эти регистры, на их вовремя наполнение, мгновенное использование и рассчитывал Крэй, когда рассчитывал свою машину.

Сколько здесь можно разогнать? Конечно, в основном разгонка идёт здесь. Частота у Cray была 80 МГц, это приблизительно 80 миллионов операций в секунду с плавающей запятой при выполнении векторных операций. Сколько операций можно запустить параллельно? Одна операция требует 3 векторных регистра, всего их 8, значит 2 операции. Таким образом, до 160 MFlops можно было раскрутить Cray-1. Это, конечно, очень серьёзное было приобретение в мировом вычислительном хозяйстве, и потому и названа была «супермашина».

На самом деле можно было больше. И это «больше» достигалось вот каким способом:

$$Q = \frac{(A+B)\cdot C}{D}$$

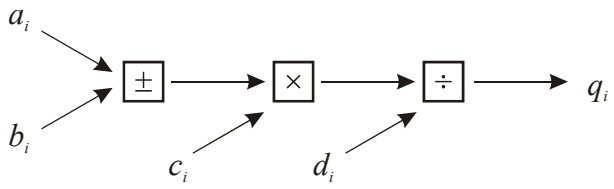


Рис. 3

Результат сложителя попадает не на регистр, а на умножитель. Туда же приходят c_i. Отсюда это идёт на делитель, куда подсовываются d_i. И отсюда выскакивает на векторный регистр Q. Итого, сколько нам нужно векторных регистров: 1 — вектор A, 2 — вектор B, 3 — вектор C, 4 — вектор D и 5 — вектор Q. Даже пяти хватает, чтобы сколько получить операций в секунду? Через 12 тактов появится q_1 и плюс ещё 63. Таким образом, через 75 тактов у вас появится результат скольких операции? 64 на 3 = 192 операций. Таким образом разогнали почти до 240 MFlops (чуть меньше). Но это ещё не предел. На Cray'е можно было в определённых случаях получать существенно больший результат. Вот это называется «*зацепление* работы векторных функциональных конвейерных устройств» или «супервекторная производительность», за счёт вот такой работы. Чуть сложнее, конечно, оборудование работает. Но вот это очень важный момент векторно-конвейерных машин и он тут реализован. Вот это два новых уровня параллелизма (вертикальный — количество функциональных устройств, и горизонтальный — конвейер) и привело к высокой производительности машины, которую изобрёл Крэй, и это всё было почтительно показано, мировое достижение, и появилось слово «суперкомпьютер».

Вспоминаются яркие впечатления студенчества, когда ходили в поход в Подмосковье и останавливались в школах, вокзалах и т.д. Вот в одной школе мы остановились города Рогачёва, и поскольку вечером пришли, утром ушли, посмотрели там учебные пособия. Запомнилось два учебных пособия. Одно с некоторой усмешкой запомнили. Например, пособие по русскому языку: «коза — козы — козочка, гора — горы — горочка, вода — воды — водочка».

А вот второе — пособие по арифметике: написана таблица 10 на 10, по краям числа от 0 до 9, но не так, как умножение. Думаем, по какому же принципу построена эта таблица? Студенты мехмата ничего понять не могут, как организована таблица. Утром мы аккуратненько, чтобы своё непонимание не светить, спросили у учителя, что это такое. Он сказал: «А вот когда ученик хочет найти число 25, он смотрит 7-ю и 3-ю колонку, он знает, что на пересечение 7-й и 3-ей колонки стоит число 25». Мы совершенно не поняли.

Итак, мы с вами рассмотрели следующие 2 уровня параллелизма: параллелизм работы функциональных устройств и параллелизм выполнения операции в конвейерном режиме в каждом из этих функциональных устройств. Мы рассмотрели с вами зацепление ФУ, т.е., напоминаю, когда у нас имеется несколько векторных операций, в которые включены операции над векторными операндами, то мы можем устроить зацепление (см. Рис. 3). Соответственно, в кодах операции указывается: у нас будет работать сложитель, со своими четырьмя ступенями, передаётся результат на следующее устройство, т.е. на умножитель, и выход идёт на делитель, и выход, естественно, идёт в свободный регистр. После разгона всех трёх конвейеров, т.е. первый результат q_1 появляется через 12 тактов, но зато уже каждый следующий q_i выходит через один такт. Итого, если взять машину Cray, где у нас 64 компонента в каждом векторном регистре, у нас получается вычисление за 12 + 63 = 75 тактов, т.е. за 75 тактов получаем, на самом деле, 192 двухместных операций. Это и есть супервекторная производительность. Если бы этого не было, нужно было бы использовать 12 на 192 — совсем много. Видите, какая большая выгода.

Называется это «зацепление работы конвейерных устройств» или «супервекторная производительность».

Кстати, мы здесь видим, что в этой ситуации у нас экономятся регистры. Всего 5 регистров задействованы. Если б мы взяли вот такие операции:

$$A + B = K$$
$$C \times P = L$$
$$D \div N = E$$

В таких случаях нам бы понадобилось 9 векторных регистров. Уже Cray с этим бы не справился, потому что у него всего 8 векторных регистров. Кстати, тут вы видите, здесь тоже 3 операции выполняются, тоже 192 элементарных двуместных операции, и выполняется это всё за 4 + 63 = 67, «×» на такт позже начинает работать, «÷» ещё на такт, т.е. к исходу 69-го такта вы получаете результат.

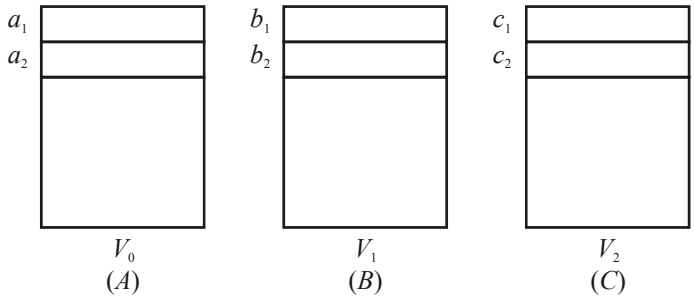
Кстати, вот в Cray было по одному устройству для выполнения операций над векторами: один сложитель/вычитатель, умножитель, делитель. Поскольку мы сейчас богатенькие стали, то можно заказать столько ФУ, сколько нужно. Сейчас нет уже такого понятия, как «серийная машина». А раньше большие машины шли сериями, т.е. машина вот такая. Хочешь — бери, не хочешь — другой нет. Теперь же любую систему берёте на заказ.

Вот я напомнил вопрос о зацеплении работы функциональных устройств. В связи с этим, имеется понятие «суперскалярный».

Суперскалярный — это когда у вас в вашем процессоре операций над векторами не задействовано, а есть несколько устройств: тот же сложитель/вычитатель, тот же умножитель, делитель — и вы можете отдельные операции практически параллелить. Вот если такое есть у вас в скалярном процессоре, там где нет соответствующих схем подачи компонентов векторов (это я очень аккуратно выразился), только такие машины называются суперскалярными.

Итак, если у вас отдельно есть группа векторных ФУ, отдельно группа скалярных ФУ, адресные — бог с ними, вот что касается скалярных операций, то можно их подсовывать внутри векторных и ничего плохого не будет. Но вот из машин типа Cray сразу вылезло две идеи. Первая идея заключается вот в чём. Вот у нас есть векторный регистр A (V_0), векторный регистр B (V_1). Я выполняю операцию

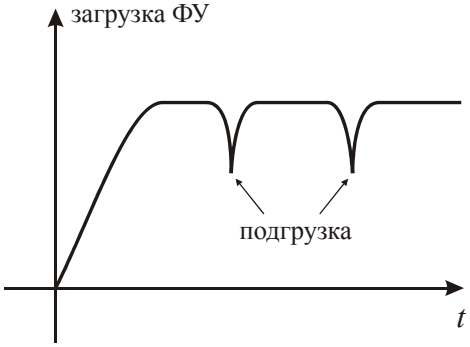
$$A + B = C$$



Значит, я из A беру a_1, из B — b_1, в C пишу c_1. Но если я так сделал, то можно считать обработанные компоненты векторных регистров A и B свободными. C я не могу считать свободным, здесь у меня заполняется. И, как бы, я мог заполнять освободившиеся компоненты уже другим материалом. Тут вопрос: команды одиночной передачи — слишком накладно будет, команды групповой — синхронизуется ли здесь всё. Можно попробовать, но всё-таки сложно. У Cray этого не было. И какая у машины Cray была проблема? А вот какая.

Естественно, возникал подъём производительности: все устройства заняты, зацепление, всё работает с максимальной производительностью. А мы пытаемся подкачать до тех пор, пока не закончится эта операция. Как только закончилась — групповая операция идёт, но она должна завершиться, на это нужно время. При этом, что это может быть? Это может быть какие-то другие данные для других операций, если вам достаточно и все операции идут над 64-х компонентными векторами. А может быть, это просто продолжение вектора A и вектора B в том смысле, что у вас вектор не 64 компонента, а 1000. Значит, вам нужно под-

загружать всё время. Поэтому, может быть падение производительности, потом снова и т.д. Вот такой будет «пилообразный» вариант графика загрузки ФУ:



Так и случилось в машине Cray.

И вот ученики Крэя сделали из этого вывод и попробовали это дело преобразить. Что они сделали? Они сделали следующее. Они убрали все регистры. Давайте смотреть ситуацию

$$Q = \frac{(A+B)\cdot C}{D}$$

Нужно при отсутствии регистров уметь подавать на вход каждый такт: a_i, b_i, c_i, d_i, и q_i записывать каждый такт. Если нет регистров, то откуда брать информацию? Конечно из памяти. Т.е. в один такт я должен уметь сделать 4 считывания из памяти и одну запись. Т.е. нужно иметь очень высокую степень параллелизма работы памяти.

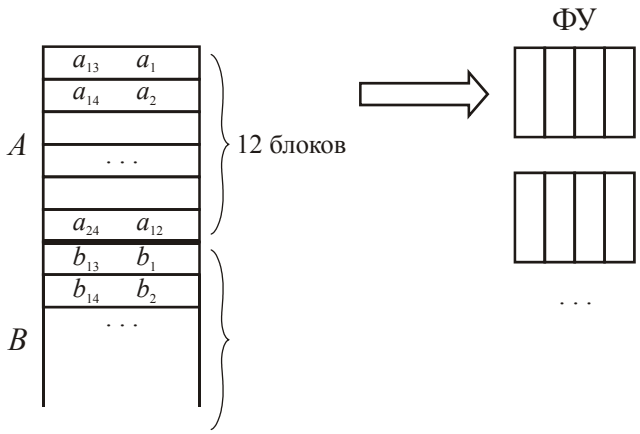
Вот в машине Cyber 205, которую ученики Крэя в фирме Control Data Corporation (CDC) сделали несколько позднее, это достигнуто было, и все ФУ работали непрерывно.

Давайте посчитаем какой-нибудь разумный вариант. Они одну трудность убрали (регистры, нужно на них вовремя подавать данные, всё это считает оптимизирующая часть транслятора, причём не всегда удачно). Они ушли от этой неприятности, но получили другую неприятность: организовать высокий параллелизм далеко не так уж просто. И вот этот подход время от времени появлялся в разработках машин векторно-конвейерного класса.

Давайте посчитаем пример:

$$Q = \frac{(A+B)\cdot C}{D}$$

Итак, разбиваем на некоторое количество блоков ОП. Как правило, цикл работы блока памяти бывает дольше цикла работы АЛУ или ФУ, как части АЛУ, в 2 раза или в 4. Пусть цикл работы блока ОП равен 12 тактам. Сколько нам нужно блоков памяти, естественно без остановки на подгрузку: $12 + (N - 1)$, где N — число компонент вектора. Хорошо. Допустим, я запустил на выборку a_1 и запустил на выборку следующий блок памяти со сдвигом на один такт. Итак, сколько мне нужно блоков для размещения компонент вектора A? Вообще говоря, 12, потому что считаем, что я запускаю через каждый такт: как только a_1 я получу, запускаю его снова на выборку a_{13}. Действительно, поскольку я запустил его через 12 тактов, через 12 тактов я получу a_{13}, а он мне как раз раньше и не нужен. Всё. Таким образом, мне нужно 12 блоков, чтобы разместить таким образом компоненты вектора A. Столько же (12 блоков) мне нужно, чтобы разместить компоненты вектора B, и т.д.



Что у меня получается: 12 + 12 + 12 + 12 + 12 = 60 блоков. Нужно 60 блоков, чтобы получить тот самый результат, чтобы они работали непрерывно. Ну а где не получается, там неправильно работать не будет, но будут задержки.

Это решение было в машине Cyber 205, и эта идея была жива, и вот мы с вами сейчас рассмотрим отечественную разработку — модульно-конвейерный процессор (МКП), правда это был 1990 год. Тем не менее, там реализовался некий симбиоз идей Cray и Cyber 205. Это характерно для вычислительной техники — учесть достоинства и недостатки зарубежных и своих разработок, и разработать некий вариант, преодолевающий эти недостатки. Институт точной механики и вычислительной техники всегда так и делал.

Модульно-конвейерный процессор (МКП)

Существовал некий многомашинный комплекс, в который входил основной вычислитель, называемый *модульно-конвейерный процессор* (МКП), или несколько таковых, и внешние машины, которые поставляют данные для вычислений (программы, данные), занимались подготовкой заданий для выполнения вычислений, постобработкой, удобной для реального вывода на реальные устройства. Cray возник тоже как многомашинный комплекс. Потом Крэй начал экспериментировать, и, в конечном счете, пришёл к достаточно устойчивому варианту, когда в качестве соединительной машины или нескольких таковых использовалась машина CDC серии 7000, которая в 10 раз менее производительна, чем центральный процессор.

Это, конечно, было совершенно ясно, уже первая суперЭВМ была двухмашинным (многомашинным) комплексом: модульно-конвейерный процессор или несколько таковых объединялся с несколькими тогда в распоряжении Эльбрусами теми же каналами, которые начали уже зарабатывать хорошее признание в использовании. И вот этой коммуникационной средой многомашинные комплексы объединялись вместе с, так называемой, внешней памятью, массовой памятью, но об этом несколько позднее будет разговор.

Итак, модульно-конвейерный процессор (МКП) — идея объединения Cray'я и Cyber 205 (см. Рис. 4).

Существовал конвейер однофункциональных устройств. Исполнительный блок — это набор конвейерных ФУ. Существовало некоторое количество S-регистров (скалярные регистры), только их было 128. Существовала оперативная память. Не было двойных как у Cray'я S- и T-регистров. Было 2 скалярных блока (S-регистр + УУ). Два устройства управления, и мы получаем 2 потока команд. Та же идея работы с S-регистрами, что и у Cray. Среди потока команд (2 потока команд), естественно, возникают векторные команды. Тогда эти векторные команды набирались в *буфере векторных команд*. И существовал *векторный блок*. Он содержал несколько регистров, но очень мало. Когда выбиралась векторная операция, естественно, она шла в векторный блок (находилась возможность чередования, смотрелась независимость данных, ко всему этому руку прикладывал транслятор, и идут аппаратные контроли этого дела), и так же как в Cyber 205, идёт поток данных широким горлом из ОП в векторный блок и обратно. А если нужно работать в зацеплении, то там были буквально не-

сколько штук регистров. Вот такая была основная идея — идея соединения решения Cray для скалярной обработки, а для векторов было взято решения Cyber 205.

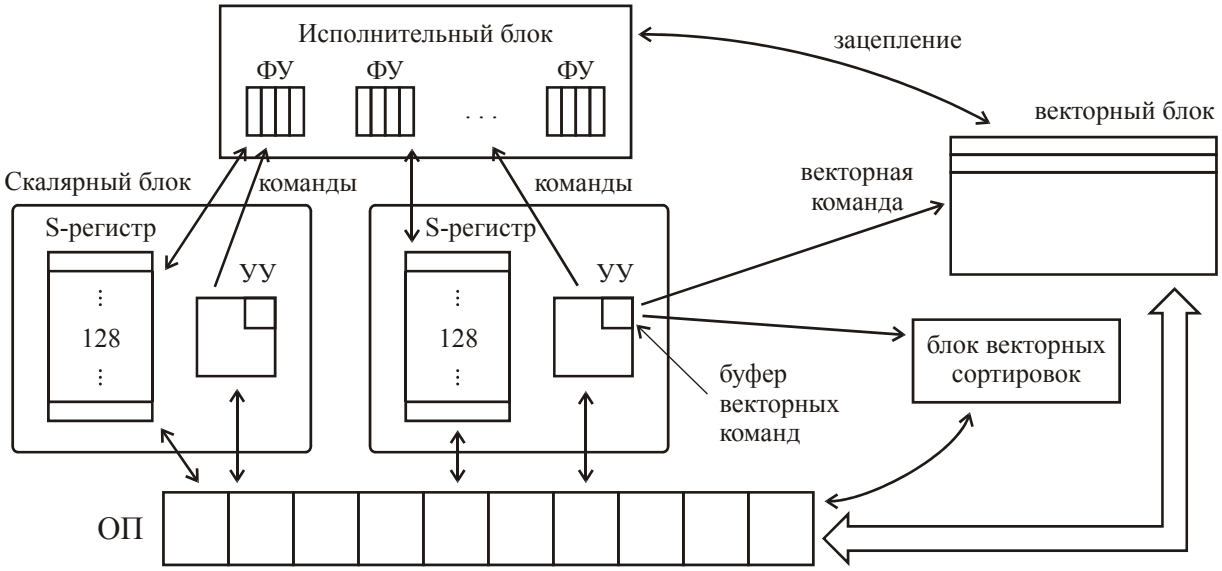


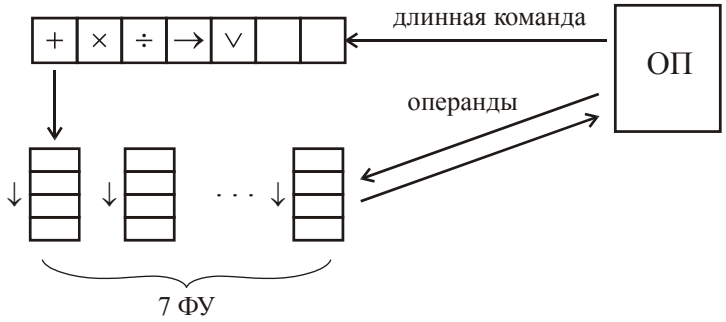
Рис. 4

Итак, машина состояла в основном из 4-х основных блоков, т.е. исполнительного блока, двух скалярных блоков, одного векторного блока. Был ещё очень интересный блок, назывался *блок векторных сортировок*. Т.е. если у вас данные распложены как-то не подряд, то можете их в этом блоке организовать подряд, чтобы потом отсюда они пускались подряд. Естественно, команды сортировок шли сюда. Здесь своя была линия с памятью, никуда дальше не шло, только работа с памятью. Вот так был устроен МКП.

Теперь так. Мысль возбудилась о том, что как же так, УУ выдаёт за один такт только одну команду. И что? Каждый раз можно загрузить делом только одно, а остальные в этот момент простаивают. Их первые ступени будут просто пустыми. Итак, появилась идея *длинного командного слова*, так называемый *very large instruction word* (WLIW).

Разработки начали появляться сразу, но большую коммерческую выгоду не принесли. Но разработки появлялись, и даже у нас была такая разработка под руководством Б.А.Бабаяна. И также точно была загублена в 1990 году из-за отсутствия средств на реализацию идеи. А именно вот что было предложено.

Была предложена машина, называемая Эльбрус 3. В этом Эльбрус 3 предполагалось, что процессор состоит из 7 ФУ — так было задумано. Конвейерный режим, конечно, был. И за такт (усложнение ОП) выбиралась *длинная команда*. В данном случае было 7 команд, и все эти 7 команд одновременно бросались на эти ФУ. Через такт приходит следующая команда — хоп! И так далее. Если нужно работать с вектором — просто циклом. Получится то же самое. Главное что устройства загружены.



Итак, всё загружено, векторных операций нет, циклом будут подаваться вот такие операции. И эта машина была сделана, но в виде стенда.

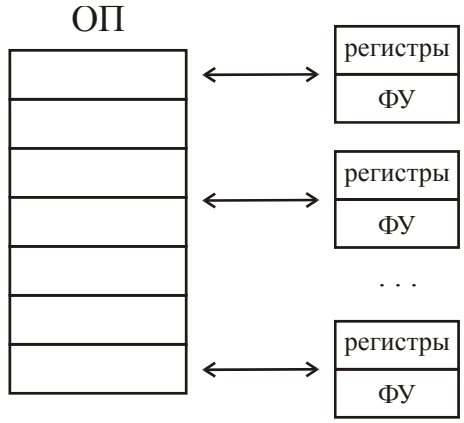
Всё бы хорошо, но в статистике говорится, что в нормальной последовательности команд 2, 3 — максимум, операции в среднем могут параллелиться. Не будет так, чтобы 7 операций, не будут они пригодны для одновременного выполнения. Это же совершенно не ре-

ально, т.е. у вас будет простаивать всё это дело. Ответ Бабаяна: «Параллелизм безграничен. Чудодейственный транслятор найдёт мириады команд (независимых), найдёт и вставит». На самом деле, это вряд ли реально, но вот такая идея длинного командного слова появилась и вот так была реализована.

Дело в том, что поработав над Эльбрусом 1 и Эльбрусом 2, где программное обеспечение целиком делал Бабаян, затем отработав вариант машины Эльбрус 3, затем объединившись с Sun, к 2000 году был разработан Эльбрус 2000 (Э2К).

Как видите, на какой-то момент времени это был самый совершенный процессор с точки зрения параллелизма в мире, и возникли дебаты, которые в американской печати очень основательно проходили. Что писали в этих журналах: «Да, это самый совершенный. Но стоит ли нам покупать эту разработку у Бабаяна? Если мы её купим, то тем самым, мы в определённой степени объединим свои интеллектуальные рабочие ресурсы. Мы сумеем достичь этого? Сумеем, потому что прогресс развивается. Есть опасность, что русские это сделают без нас (возьмут и завалят мир этими процессорами)? Нет такой опасности, потому что для того, чтобы поставить на производство 200 миллионов долларов хватит. Но само производство — это чистые материалы, чистое производство — оно стоит 4 миллиарда. Вот такое они не потянут. А у нас все уже почти есть, поэтому не будем покупать у них разработку». Бабаяновцы на этот счёт обиделись, поехали в Америку, поднялись на гору Мак-Кинли и поставили там флаг Эльбруса. Так всё ничем и кончилось.

Стало возможным при увеличении интерливинга памяти подключать не один процессор со всеми его потребностями (считывание команд, операндов, записи результатов), а уже несколько процессоров. И раз уж мы не полностью отошли от Cray’я, тот же Cray стал создаваться из двух машин, под названиями Cray X-MP и Cray Y-MP. Это были несколько процессорные машины с общей памятью. Т.е. была память, и были процессоры. Внутри каждого из них были регистры и ФУ. Было по 4, по 8 и даже по 16. Здесь довольно эффективно можно решать одну задачу, распараллеливать на несколько ветвей, потому что одна ветвь работает с памятью, другая — с другим участком памяти и т.д. Потом естественно, это одна задача, результат подхватывается другой программой другого процесса и наоборот. Тут уже можно хорошо параллелить решение одной задачи на несколько параллельных ветвей. Или решать их параллельно как независимые задачи.



Вот возникли такие решения, и собственно говоря, все 80-е годы шли на создание этих машин по линии векторно-конвейерных машин Cray, ну и соответственно программное обеспечение и т.д. И, собственно говоря, может быть, даже именно Cray и был ближе к этой конфигурации многопроцессорных комплексов, которая называется SMP (symmetric multiprocessor). Вот, это одно из подмножеств многопроцессорных комплексов, в котором стало очень удобно разбить задачу на параллельные ветви, очень удобно передавать от одной ветви к другой полученные на предыдущем шаге обработки результаты и использовать их дальше. Писались в языках программирования операции синхронизации таких ветвей. Но есть некое затруднение: обращений много к памяти. Оказалось, что всё-таки больше, чем 4, 8, в крайнем случае, 16, подключение такого количества процессоров неэффективно (сразу возникают задержки в работе с оперативной памятью). И вот так на

возникают задержки в работе с оперативной памятью). И вот так на этом эти комплексы SMP стали останавливаться.

Ясное дело, что если у вас имеется необходимость существенно большего параллелизма, для многих задач это требуется, недаром делают комплексы десятки тысяч процессоров, то стала относительно ограниченная, но довольно активно используемая (до сих пор такие комплексы SMP делают).

Был реализован комплекс Эльбрус 2: какое-то количество блоков ОП, до 10 процессоров. Сейчас работает в системе противоракетной обороны Москвы. Процессоры через некий матричный коммутатор имеют доступ к этим многочисленным блокам памяти. И вот этот вот Эльбрус 2 — типичный пример SMP-шного решения. Кстати, этот коммутатор был очень удачный.

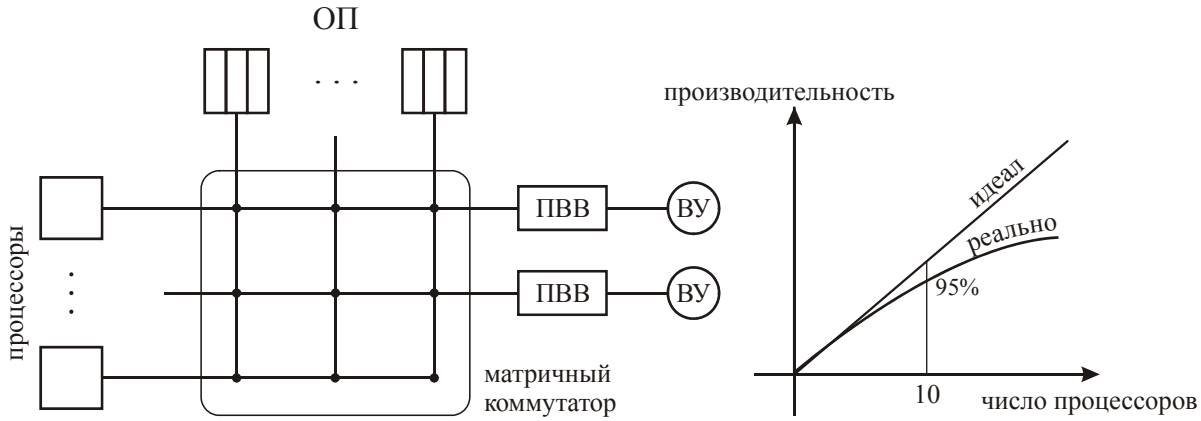


Рис. 5

Идеальной является прямая пропорциональность, но это не достигается. Так вот у Эльбруса 2 при 10 процессорах был достигнут почти линейный рост на некоторых очень важных задачах. Это было достигнуто совсем не просто. Семь месяцев эта задача обрабатывалась системщиками и разработчиками самих алгоритмов счёта, пока им удалось получить 95% при 10 процессорах. И вот Михаил Сергеевич Горбачёв на съезде комсомола поставил главное решение науки и промышленности и наградил молодежь премией за 95%.

Теперь смотрите внимательно: очень интересная вещь, которую можно обсудить. Вот два примера вычислительных комплексов: Cray и Эльбрус 2 — вполне достаточно. Вот смотрите: память общая. Вообще говоря, здесь можно реализовать и многомашинный комплекс. Представить себе, что вот здесь под управлением некоторой операционной системы решается задача Иванова и Петрова. А здесь — Сидорова. Если это объекты с разными операционными системами, то это многомашинный комплекс — это взаимодействие операционных систем между собой через общую память. Т.е. это многомашинный комплекс, поскольку у каждого объекта какие процессоры, несколько процессоров, следующий объект — тоже несколько процессоров (например, 2 по 5 — две многопроцессорных машины по 5 процессоров). Т.е. если есть некая сущность, управляемая собственной операционной системой, объединяется несколько, то это есть многомашинный комплекс. Если одна операционная система, то это одна многопроцессорная машина. А то, что комплекс и Cray, и Эльбрус в любом случае многомашинный, доказывает следующее: к Эльбрусу подсоединяют, так называемый, *процессор ввода-вывода* (ПВВ), для которого были внешние устройства (ВУ), которые обеспечивали работу с внешними устройствами (см. Рис. 5).

Вот на данном оборудовании получился трёхмашинный комплекс: один — многопроцессорный Эльбрус 2, второй — процессор ввода-вывода (ПВВ) как отдельная машина со своей операционной системой (2 штуки).

Действительно, многоэлементный многопроцессорный комплекс не может жить без помощи и поддержки дополнительных машин.

Электроника СС БИС («Красный Крей»)

Машина, созданная параллельно с МКП, назвалась Электроника СС БИС («Красный Крей»). Она представляла собой вариант SMP — Cray SMP, только нашими сделано, поэтому я ничего дополнительно не буду. Ну, там были улучшены устройства выполнения операций чисто логически. Всё было сделано, конечно, на своей элементной базе, в этом весь смысл: она по своей архитектуре относилась к системе Cray. Но что в ней было сделано дополнительно, что потом появилось во многих машинах, в том числе и в Cray'е, не потому что «ах, русские сделали, давайте и мы сделаем». Нет. Естественно возникла потребность — это, так называемая, *массовая память*.

Итак, у вас есть оперативная память (ОП), процессор, и некое устройство обмена (УО). Вот здесь некоторое количество каналов, к которым могут подсоединяться внешняя машина (ВМ), другая внешняя машина. Кстати, как и у Cray'я, так и у Электроники СС БИС отдельно были магнитные диски (МД). Ничего другого, правда, не было, а магнитные диски были. Всё остальное через внешние машины. Был некий коммутатор, и вот здесь была массовая память (МП).

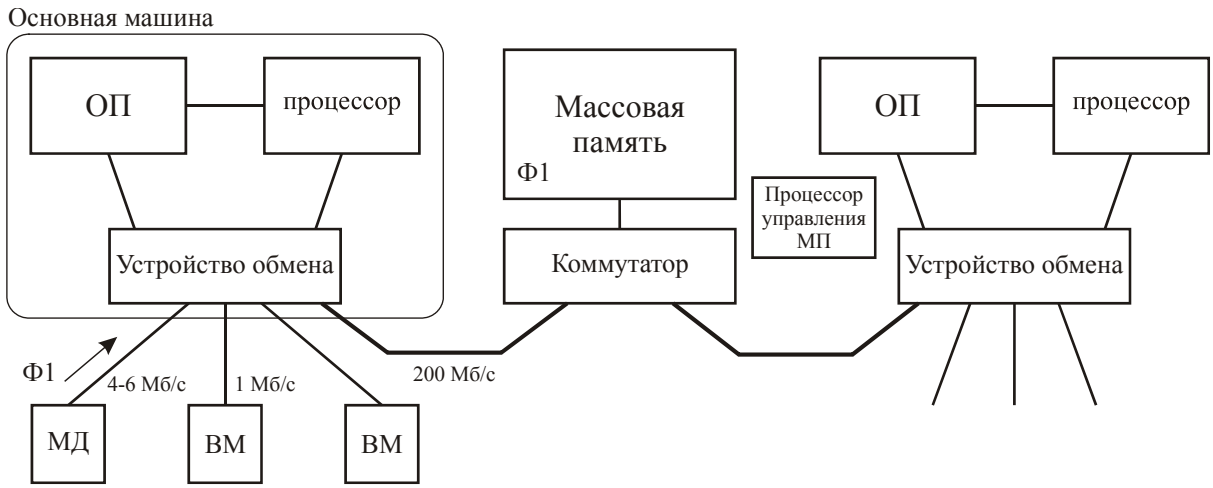
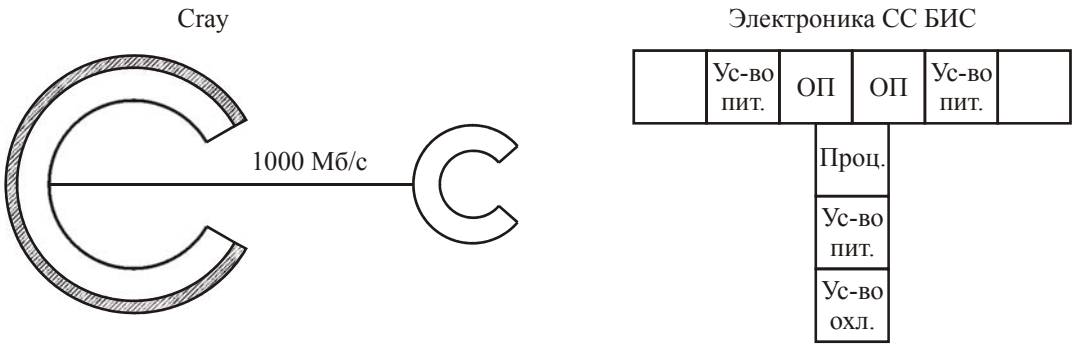


Рис. 6

ОП, процессор и УО называлось *основной машинной*, в отличие от внешней машины (ВМ).

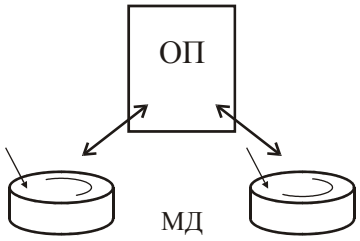
Что же это такой за фокус с массовой памятью? А дело вот в чём, что даже, несмотря на диски высокой пропускной способности, всё равно возникает для ряда задач такая ситуация, что данные не успевают поступать в процессор, и процессор простаивает из-за отсутствия данных (из-за дисбаланса скорости работы машины и диска). Как быть? И вот введён промежуточный уровень памяти, вы берёте активный файл (Ф1) и перекачиваете его через ОП в массовую память (МП) (см. Рис. 6). ОП — память на интегральных схемах, но и МП на интегральных схемах. Но она больше (скажем в 10 раз), в 10 раз медленнее, но в 10 раз быстрее и 10 раз меньше, чем память на магнитном диске. Скорость обмена между УО и коммутатором — 200 Мб/с. Что сделал Крэй? Центральная часть — «подкова» — тут и память и всё прочие (для обслуживания, для всего прочего), снаружи источники питания, внизу зал холодильных установок. У Cray'я это было полукольцо, а у Электроники это было как «самолётик»:



Вот эта подкова (её содержание) то же самое, что центральная часть Электроники. Крэй интегральную панель поставил здесь же и соединил её, только «труба у него была повыше, и дым стал погуще», он стал 1000 Мб/с. И тоже самое несколько таких Cray'евских подков — стало массовой памятью. Т.е. это было сделано совершенно независимо — это реальная потребность в повышении эффективности обработки информации. Вот это массовая память. Крэй свою идею опубликовал к 90-му году, а наши идеи возникли раньше (года на 3-4).

Вот эта память управлялась неким *процессором массовой памяти*. Вот это очень интересная вещь. Этот процессор к многомашинности никакого отношения не имеет.

Кстати, у Крэя был процессор управления магнитными дисками. Что это такое? Это вот что такое: имеется готовый заряд, подготовленный операционной системой основной машины (того же Cray'я), заряд, подготовленный для обмена. Важный момент: у вас имеется несколько дисков, вот у вас массив, который нужно прочитать, как-то так записан на дорожку:



Так вот, какую взять? Второй поближе, но если я возьму, первый прокрутится и потом у меня будет больше. Поэтому нужно смотреть, и повиноваться алгоритмам, разработанным во многих кандидатских и докторских диссертациях. Это очень сильные алгоритмы, чтобы суммарный простой этих каналов был наименьшим. Такой процессор был у Cray. У электроники такого процессора не было, но зато был *процессор управления массовой памятью*. Что в нём сидело? В нём сидели программы реализации методов доступа. Что это такое? Вот смотрите: вот у вас имеется матрица 1000 на 1000. Она сидела где-то на диске, вы её перекачали в массовую память, и вот у вас в массовой памяти матрица 1000 на 1000 (для современных задач вполне естественно — сколько-то мегабайт, подумаешь, мелочи какие). Вот вам нужно взять диагональ — всего 1000 элементов. Нормально, вы её перегоняете в ОП, и тут процессор спокойненько возьмёт эту тысячу. А что, если перегнать только эту тысячу? Т.е. вы задаёте обмен операционной системе по этому каналу. Т.е. фактически эта программа работает в процессе обмена, подсовывая адреса массовой памяти. Итак, вот массовая память.

Итак, мы закончили вопросы организации векторно-конвейерных машин двумя рассмотренными нами уровнями параллелизма: параллелизм работы функциональных устройств и конвейер исполнения операций в каждом из этих устройств. Визитной карточкой этих машин и до сих пор является Cray. Мы рассмотрели следующий этап — это уже работа многопроцессорных машин, когда несколько процессоров со всем содержимым, которое мы рассматривали, подключается к общей памяти. Естественно, память расслоена, но уже она нагружается несколькими обращениями от каждого из процессоров (за командами, за считыванием операндов). Ну и, вообще говоря, обмены с дисками — это так же обращение к памяти. Большая периферия у Cray'я отсутствовала, она подключалась через вспомогательные машины. Напоминаю, что у первого Cray'я, который, кстати, был поставлен не в Америке, а в Великобритании (в метеорологическом центре), сопровождалось

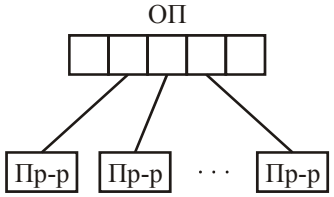
в Великобритании (в метеорологическом центре), сопровождалось такой внешней машиной, организующей передачу заданий, подготовку заданий, хранение массивов данных. Это была малая машина Eclipse. Потом постепенно внешние машины стали существенно более мощными, и уже застабилизировался тот комплекс Cray (естественно, многопроцессорный) на поддержке внешних машин (достаточно производительных) Control Data Corporation серии 7000, где в этой фирме (CDC) остались ученики Крэя. Уже пройденный был этап Cyber'а.

Т.е. вообще говоря, суперЭВМ (машины и комплексы) все имели поддерживающие внешние машины, а то и несколько, которые, по сути, помогали вычислителю, беря на себя front-end заботы — подготовку заданий, окончательную обработку результатов, вывод на устройства вывода. Эти многомашинные комплексы организовались совершенно естественным образом с выделением центрального одно- или многопроцессорного вычислительного звена и с выделением слоя поддерживающих средств.

Рассматривая машину Cray многопроцессорную с общей памятью, мы отметили, что аналогичным образом машинам с общей памятью были организованы машины Эльбрус: Эльбрус 2 — до 10 процессоров работали на общей памяти с подключением с помощью коммутатора. К этому же коммутатору подключались те самые внешние машины, которые в Эльбрус 2 назывались ПВВ (процессоры ввода-вывода), где своя операционная система заведовала работой устройств и передачей данных. Эта операционная система контактировала с операционной системой мультипроцессора. А могла бы контактировать с многими операционными системами, если бы каждый процессор управлялся своей операционной системой, что вполне было бы возможно.

ILLIAC VI

Итак, вот мы рассмотрели два примера SMP (symmetric multiprocessor) — Cray и Эльбрус, да ещё вот увидели, что они поддерживаются внешними машинами, образуя вместе многомашинный комплекс. И к тому же мы отметили, что вот эти SMP (когда у нас на общей памяти, к ней обращаются какое-то количество процессоров с одинаковым временем обращения), и ясно, что, несмотря на довольно большой интерливинг, очень большое количество процессоров подключить не реально.

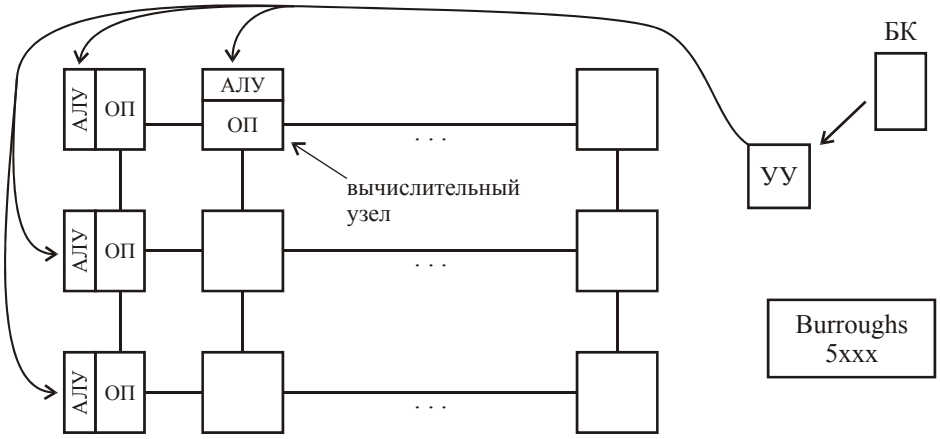


Поэтому такие комплексы и были 4, 8, 16 (в Эльбрусе — 10). А что делать, чтобы мы всё-таки могли распараллелить нашу задачу на большее число ветвей, чем вот. Ясно, для чего параллельно работающие процессоры — чтобы на каждом шла своя ветвь общая, какая-то своя задача. Чтобы либо больше ветвей, либо больше задач решать на этой общей памяти. Надо было что-то делать.

Надо сказать, что уже существенно до появления машины Cray Y-MP, X-MP, Эльбрус ещё в 1971 году (помните, однопроцессорный Эльбрус появился в 1976 году) появился комплекс, который назывался ILLIAC IV. Очень интересная судьба у этого комплекса. Иллинойский университет, вообще говоря. Видите, университеты в Америки активно участвовали в разработки очень многих средств вычислительной техники. И, естественно, создавали программное обеспечение. Люди активно участвовали и в разработке архитектуры таких машин вместе с фирмами. Поэтому роль, конечно, очень большая, и много на это средств выделялось — это очень существенный момент.

В 1971 году на навесных деталях была разработана вот такая система: было разработано 64 *вычислительных узла*. Объединены они были в топологию «решётка». Вычислительный узел представлял собой арифметико-логическое устройство и оперативную память. Как видите, здесь нет одного компонента — полностью организованного нормального процессора (нет устройства управления). Было единственное устройство управления (УУ), которое,

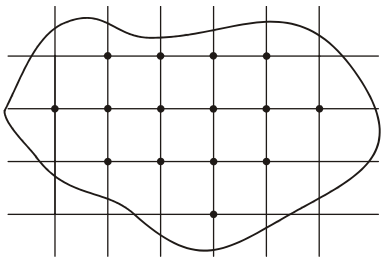
получая из некоторого буфера команд (БК) команду, транслировало её одновременно на все арифметико-логические устройства. Существовала некая маска, которая показывала, на какой узел нужно передать команду, на какой — не надо. Команды все одновременно выполнялись. Что могла обрабатывать такая команда? Например, сложить третью ячейку с пятой и результат переслать в седьмую ячейку своей памяти. И вот все они выполнялись одновременно. Это получились многоалушные узлы, где узлу всегда привязывается сущность памяти (узел всегда имеет память).



Вычислительные узлы можно рассматривать и как машины, поскольку здесь есть устройство управления, есть вычислительное звено — уровень вычислений, и уровень памяти. Откуда ж брались команды? Очень просто: стояла машина Burroughs 5000 с чем-то, и эта машина была той самой внешней машиной вот этого фактически многомашинного комплекса. Но нельзя так называть — «многомашинный»: здесь не было своей операционной системы. Помните, мы говорили, что многомашинный это тогда, когда у каждого элемента комплекса есть своя операционная система. Тем не менее, удивительная машина была — Burroughs, — на которой и готовились задачи, программа передавалась в буфер команд (БК), ну а дальше… Естественно, был канал данных в ОП — забирались данные.

64 было вот таких вычислительных узла — некий квадрат. Хотели сделать 4 квадрата по 64, но сил не хватило. Слава Богу, что сделали это. Машина просуществовала год или 2, была очень дорога в эксплуатации. Машина позволяла развить производительность порядка 300 MFlops (Cray только при 3-х конвейерах приближается к этому), но это регулярных вычислений.

Когда у вас есть какая-то область, которую вы покрываете сеткой (при решении дифференциальных уравнений), то в каждом узле регулярной сетки (я не беру границу) вы выполняете однотипные действия — вот это то самое.

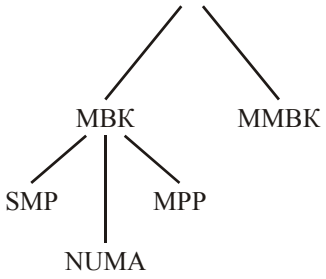


И понятно, что здесь основная вычислительная нагрузка и лежит, и какие-то начальные вычисления, последняя часть, они, конечно, не могли задействовать все узлы. Задействуются только некоторые из них, и естественно, производительность была не большой. А вот когда шло регулярное вычисление — производительность существенно возрастала. Задачи решались с очень высокой производительностью. Какие задачи? И твёрдое тело здесь можно исследовать, и течение воздушное и жидкости и т.д. — самые насущные задачи, которые нужно было решать во все времена.

Так вот, просуществовала эта система не больше года, но в учебниках находится всюду. Есть один феномен, который короткое время просуществовал, большого эффекта на разви-

тие не дал, но вот в любом учебнике эта система есть и долго ещё будет существовать. Куда от неё деться, потому что действительно прародитель такой многоэлементной системы, в данном случае — многоалушных узлов. Вот такая была система.

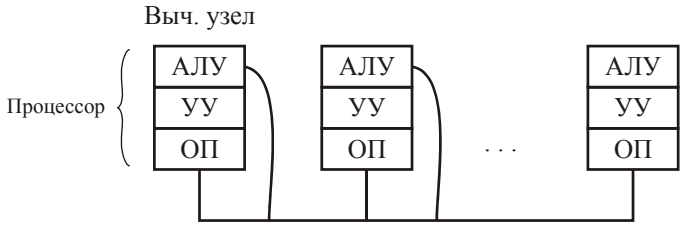
И вот если смотреть на ту классификацию, которую мы с вами говорили, то увидим:

ОКОД	ОКМД	МКОД	МКМД
Cray (если не брать его векторную часть), PC, Work-Station, Mainframe	ILLIAC IV, векторная часть Cray, ПС-2000 (по образцу ILLIAC IV)	∅	

Многопроцессорные вычислительные комплексы

NUMA

Теперь мы нарисуем несколько другой комплекс. Берём вычислительный узел, только теперь он уже полноправный — АЛУ, УУ, ОП. И вот у вас есть некая магистраль, связывающая памяти, да и арифметико-логические устройства сюда тоже могут обращаться.



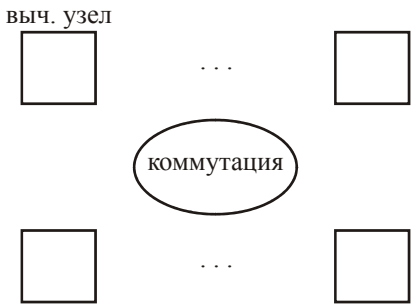
Вот, вы работаете со своей памятью. Если их много на этой магистрали, то обращение к чужой ОП возможно, но несколько дольше. Но вы можете так попросить операционную систему, чтобы наиболее часто используемые данные были расположены поближе (если, конечно, получится). Может выполняться много вычислительных задач, и это может оказаться не так уж просто выполнимым. И всё-таки тогда у вас будет достаточно эффективно выполняться работа. А данные, к которым вы редко будете обращаться, пусть будут где-то в отдалении. У вас будут редкие обращения, это будет дольше, естественно, при числе этих узлов (процессоров) существенно превышающем, могут обращаться ко многим узлам. Но при этом, если удачно расположить данные. Вот такая система носит название NUMA (non uniform memory access). Тогда SMP должен называться UMA.

Такие комплексы создавались. Целый ряд серверов делался и компанией HP и рядом других компаний. Довольно-таки заметное количество (можете посмотреть www.parallel.ru, там как раз все эти вещи). Довольно активно использовалось, но можно сказать, тоже не сверх-сверх широко распространённо, т.е. как какой-то промежуточный вариант.

MPP

Теперь, следующий вариант организации многопроцессорного комплекса. До был многоузловой комплекс, потом появился многопроцессорный комплекс, потом появился многоузловой-многопроцессорный комплекс, и, наконец, появился комплекс MPP (massive parallel

processors), где каждый узел — это тот же вычислительный узел (как в NUMA), и эти узлы уже связываются через некоторую сеть связи, некоторую коммутацию.



Вот получается некая коммутация этих вычислительных узлов многопроцессорных комплексов MPP, если работа идёт в каждом узле только со своей памятью (в смысле командная, т.е. вычислительная ветвь, которая обеспечивает параллельное выполнение, или вычислительная ветвь одной задачи, или многих задач). Здесь уже нужна передача данных, если вычислительные ветви одной задачи (потому что они одной задачи, они должны иметь доступ к данным, которые будут получены на каком-то этапе в результате выполнения какой-то другой ветви для использования в этой ветви). Нужно как-то общение организовывать. Но поскольку каждая команда работает только со своей памятью, это отдельно организуется посредством передачи сообщений через поддержку операционной системы. Для разработки очень много было таких средств. Наиболее известное — библиотека MPI. Естественно всё это делается при трансляции, с учётом указаний пользователя, вот эти передачи данных через коммуникационную среду происходят.

В MPP мы ещё хороших примеров не посмотрели — сейчас посмотрим. Кстати, в вычислительном узле может работать SMP-шная конфигурация. Даже такой термин появился (в литературе всюду, в производящих фирмах) — «SMP кирпич». Например, ОП + 2 процессора, ОП + 4 процессора. Т.е. на самом деле в этих узлах стали появляться вот эти SMP кирпичики. И уже когда мы подойдём к понятию «кластеры», что очень близко к понятию SMP (сущность кластера близка к сущности SMP), там тоже элементом (узлом, машиной) кластера может быть SMP. Что такое кластеры — к этому делу мы сейчас подойдём и все вещи посмотрим.

Конечно, при программировании на MPP, совершенно другая модель программирования — вы уже программируете сознательно по узлам и организуете сознательно связь этих узлов. Например, есть комплекс, который разработал Институт прикладной математики. Там была очень интересная ситуация: допустим, вы делаете свою задачу, у вас там 5 процессов. Вы их считаете находящимися на неких виртуальных узлах с первого по пятый. Вы обращаетесь к операционной системе [шёпотом]: «Ну, пожалуйста, по возможности поближе расположи… Первый — там особенно, а уж пятый — ладно». Но в этом комплексе могла сложиться к этому моменту ситуация, далеко не просто выполнить пожелание этого дела. Прямо писалось: первый виртуальный передаёт сообщение четвёртому, третий — первому и т.д. Это программировал пользователь, используя системы поддержки для этого дела. Вот: там своя модель программирования, здесь — своя.

Была система, созданная в Институте кибернетики академии наук Украины в 70-х годах (в Советском Союзе). Т.е. стремление многоэлементности, мы видим, прорвалось в ILLIAC IV на базе подхода ОКМД. Ну а стремление к многоэлементности прорвалось в разработках Института кибернетики под руководством академика Лужкова Виктора Михайловича — создатель системы, которую они назвали «макроконвейер». Я категорически не согласен с этим названием, возражал против этого названия, и Виктор Михайлович под конец признал. Больше того, в начале эта система называлась «Рекурсивная ЭВМ». И вот в 1974 году (чёрти когда, самые первые подходы к многоэлементности многомашинного комплекса) был доклад в Стокгольме на конгрессе, и Лужков сделал доклад, и в литературе звучало слово «Рекурсивная ЭВМ», хотя совершенно не понятно, откуда это слово взялось. Их было трое: автор — Лужков, Игнатьев, Торгашов. Торгашов занимался многомашинными ком-

плексами для военных целей, пытался сделать что-то. А Игнатьев — тот перестал, сейчас пишет замечательные книжки: «Виртуальный велосипед», «Виртуальные миры» и т.п. вещи.

Среди разработок были машины «Мир», которые были сделаны в Киеве, с системой команд, отражавшей конструкции языка высокого уровня в плане Алгола.

Я хорошо знал Виктора Михайловича. В Москве работала и его дочь, вышла замуж за сына тоже известного человека — сына Титова, человека, который вместе с Криницким писали первые книги, назывались они «Вычислительные машины и программирование», «Как устроены вычислительные центры» (в том числе и военные).

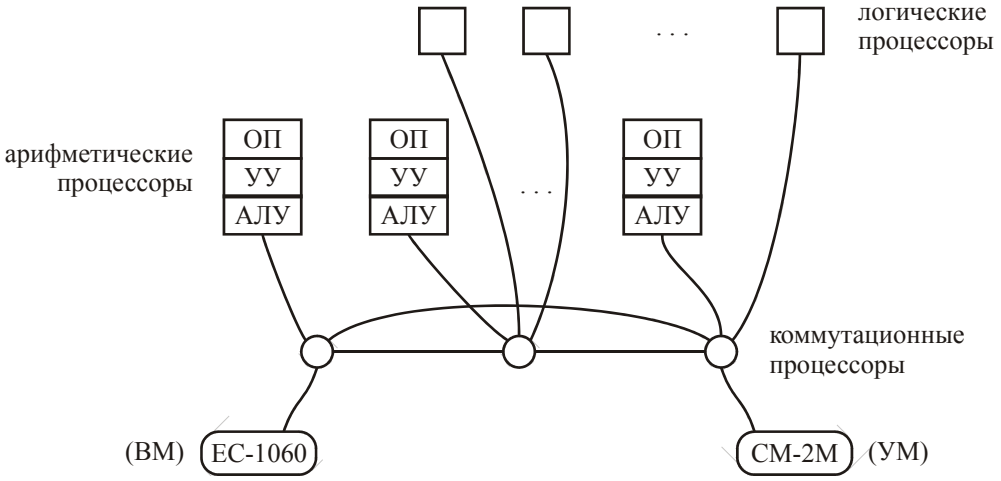
Кстати, Титов, которого я упомянул, — это писатели, которые в своё время писали книжки. Первых книг было мало. И книги Титова, Криницкого, Титова и Криницкого «Вычислительные машины и программирование», конечно, они были не откровением только их, а обобщали то, что было написано. Но была и книжка под руководством коллектива М.Р.Шуры-Буры «Решение математических задач на вычислительных машинах», но это всё первые книги.

Я вам не рассказывал о защите диссертации Криницкого на механико-математическом факультете Московского университета? Он какое-то время был редактором журнала программирования, потом путь его сменился. Это был где-то 60-й год. Вызывает меня сам директор: «Зайдите в кабинет Сергея Алексеевича, там лежит диссертация, нам на неё нужно дать отзыв от имени головной организации». Для каждой диссертации назначается головная организация диссертационным советом — считается наиболее компетентной по этой тематике. Но в этом случае получился прокол именно в этом плане. «Смотрите, она лежит там, на столе, а нам надо на неё написать отзыв». Я прихожу, открываю: введение на меньше, чем полстранички, что-то такое, что важна обработка информации (кто ж возражает), для этого нужны методы. И дальше, без всякого какого-нибудь логичного перехода, пошли 88 теорем. Я даже не могу сказать, что за теоремы. Я читаю первую теорему и ломаюсь напрочь: мне не понятны ни исходные посылки, ни ход доказательства теоремы, ничего. Я сижу и горюю: выпускник мехмата ломается на первой же из 88 теорем — состояние ужасное. Входит Мухин Иван Сергеевич (замдиректора), и Криницкий входит. Тот на меня смотрит (видимо у меня был совершенно размазанный вид) и говорит: «Вы смущены?.. В затруднении?..». Я говорю: «В чрезвычайном». Мухин (замдиректора) смотрит, что я совершенно никакой, и говорит: «Хорошо. Вы сейчас идите, тут есть «рыба», мы потом итоговое наше мнение выскажем, а по конкретным деталям есть предполагаемый отзыв». Не помню, чем точно это дело кончилось. Сейчас будет эта работа защищаться. Действительно, был большой и интересный курьёз. Защищалась она в 1408 (в главном здании — плоская аудитория). Я пришёл и скромненько сел где-то ближе к заднему ряду, и ожидаю, когда начнётся защита диссертации. Постепенно наполняется зал, а впереди сидит куча академиков, и чего-то между собой живо обсуждают, чего-то там гудят. Начинается защита диссертации. Ведущий говорит, что будет защищаться диссертация Криницкого, на такую-то тему. «Давайте предоставим слово кандидату в диссертанты». И в этот момент поднимается кто-то из великих и говорит: «Можно слово по порядку ведения?» И тут выходит чуть ли не сам Павел Сергеевич Александров: «Вы понимаете, какое дело. Мы тут посмотрели эту диссертацию — там 88 теорем, никто из нас ничего не понял. Вы понимаете, какое дело. Мы знаем Криницкого как человека, который создал много вычислительных центров, это наше будущее. Мы готовы принять докторскую диссертацию без защиты, потому что он действительно засуживает того. Но слушать 88 теорем, и при этом там есть отзыв такой некомпетентной организации, как Институт точной механики и вычислительной техники (поделом попало за это дело). Вы нас, пожалуйста, увольте. Правда, мы тут выяснили, что наше желание присудить ему докторскую степень по формальным причинам не может быть выполнено. Тем не менее, мы вносим предложение не слушать доклад диссертанта. Кто "за"?». Все проголосовали «за» — было 22 человека. Но как-то оппоненты критику должны сказать. Павел Сергеевич: «Ну пусть скажут выводы, где там есть недостатки, хотя нас это уже мало волнует». Оппоненты вышли, через

пять минут ушли. Надо голосовать — стали голосовать: 22 : 0. Вот такая оригинальная была штука.

Это всё к чему? «Рекурсивная ЭВМ» — придумают же такое. А что же на самом деле придумали в середине 70-х годов, а потом они стали называть это «макроконвейер», что тоже на, самом деле, неправильно. А придумали следующее. Они решили собрать некоторое количество, как они назвали, «*арифметические процессоры*». На самом деле это был просто обычный комплект, т.е. ОП, УУ, АЛУ и некоторая номенклатура ЕС-ЭВМ. Будем условно говорить модели ЕС-ЭВМ — без всякой внешней периферии, без всего. В таких случаях называли раньше «процессорный комплект». Вот такие арифметические процессоры. Слово не очень удачное, потому что здесь арифметические и логические операции. Но назвали так, в отличие от некоторых «*логических процессоров*». Название не потому, что здесь алгебра логики. Иногда названия не очень удачные, например «рекурсивная ЭВМ». И всё это вместе объединялось некоторыми связями через некоторые шины, которые объединялись «*коммутационными процессорами*». Логические процессоры были придуманы со специальной системой команд, на которой на специальном языке (язык МАЯК) писались управляющие части программного комплекса. А части счётные писались на обычном Fortran'е. Т.е. писалась задача на двух языках: управляющая писалась на МАЯКе, и писались трансляторы, транслирующие в систему команд этих оригинальных логических процессоров, а штатные (массовые) — транслировались штатными трансляторами Fortran'а. Смысл тут какой? Допустим, что всё загружено, и работает система. Естественно, что логические процессоры отслеживают прохождение вычислительных ветвей, и снова передаю по этой коммутационной сети сообщения. Это была типичная MPP-шная система передачи сообщений: заканчивалась работа арифметического процессора, передавалось сообщение другому процессору и т.д. Кроме того, вот что-то сломалось. Естественно эту ситуацию отслеживал логический процессор (сломалось — не выполнялась во время обработка, таймер, всё прочее), передавал эту часть обработки другому арифметическому процессору. Связи позволяли выход из строя логического процессора тоже компенсировать. Т.е. система была нацелена на достижение максимальной живучести. Как они тогда говорили: «Мы такую систему сделали — рекурсивную». Почему «рекурсивную»? Что же дальше получилось на практике, чем закончилось это всё? Закончилось, к сожалению, тоже двумя экземплярами. В данном случае не из-за того, что обслуживать было сложно (может быть и это тоже). Была конкуренция на использование заводских мощностей в Советском Союзе. Им с этой системой не повезло: больше двух экземпляров не сделали.

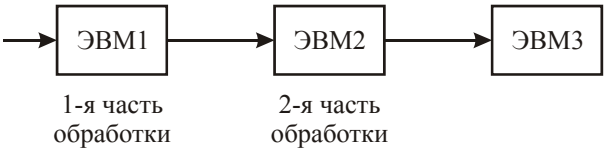
А что же ещё подключалось к коммутационным процессорам? А подключалось следующее: машина ЕС-1060 или 1061. Вот он — front-end компьютер. Всё вместе — многомашинный комплекс. Здесь работала своя операционная система. В каждом из вычислительных узлов (процессоров) работала какая-то часть общей операционной системы. Кроме того, была подсоединена машина СМ-2М, которая была управляющей машиной (УМ). Её операционная система контактировала с распределённой операционной системой многоэлементного комплекса (сказать «многомашинный» нельзя, «многопроцессорный» — ближе).



Я сам поработал на этой системе, было очень интересно. Вот сейчас у нас решается задача с алгоритмами линейной алгебры, что-то связанное с обращением матриц. Мы вам разрешаем поупражняться. Там было 48 процессоров — логических было штук 8, остальное были арифметические и несколько коммутационных. И вот я, передавая операционной системе управляющие воздействия с терминала, говорю: отключить процессоры с 11-го по 18-й и с 30-го по 35-й. А где-то была индикация скорости выполнения задачи — решение задачи замедлилось. Я довёл до двух процессоров — задача продолжала решаться. Потом я всю эту мощность другими кусками снова дал указание на включение. Вот это была очень интересная вещь.

Надо сказать, что многие у нас решения были передовые в то время, хотя, конечно, это требовало огромного зала, всего.

Мы рассматривали конвейер в микро масштабе — одного ФУ. Но мы можем рассматривать, и мы будем это делать, конвейер в макро масштабе.



Какая-то реальная информация (обычно в системах реального времени) попадает на первую часть обработки на ЭВМ1. Вторая часть обработки на ЭВМ2. Для чего так делают? А для того, что во-первых, ЭВМ1 успевает выполнить первую часть обработки за то время, через которое надо уже заниматься первой частью обработки следующей порции информации. Всю обработку она не успевает сделать в реальном времени. Стало быть, надо передать вторую часть обработки второй ступени этого конвейера ЭВМ. А первую часть обработки вести на этой ЭВМ1 параллельно со второй частью обработки первой порции информации. А это будет первая часть обработки второй порции информации. Будет работать конвейер ЭВМ. Это с одной стороны. С другой стороны, у вас может быть такое вычисление, что первую часть обработки выгодно проводить на машине с одной системой команд. Следующую часть обработки выгодно проводить на машине другого типа. Т.е. можно много выиграть, если организовать вот такую конвейерность обработки. Естественно, речь идёт в основном о постоянно наступающей и требующей в определённое время обработки информации. Так что действительно, это конвейер ЭВМ.

Это было одно из решений в MPP организации. Теперь я приведу ещё пару решений.

Есть очень хороший комплекс, на котором очень большие деньги заработала себе фирма IBM. Она реализовала комплекс под названием SP2 вот каким образом:

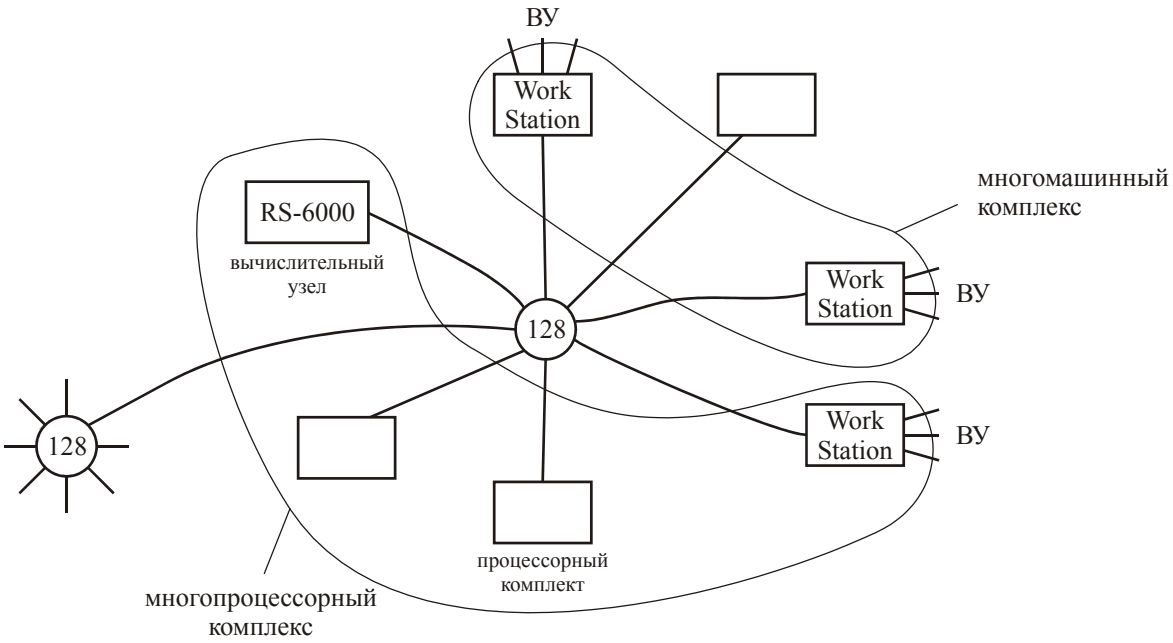


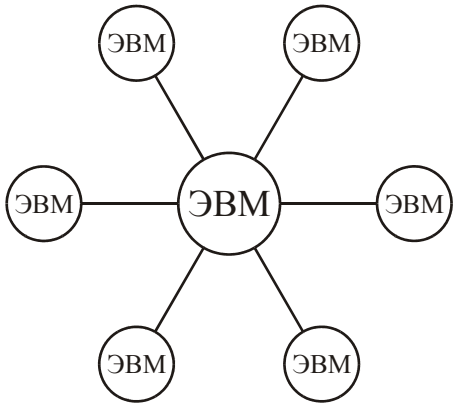
Рис. 7 Комплекс SP2

Вычислительные узлы были на базе RS-6000 (это для конкретизации, какая нам разница). Таких узлов было много. Эти узлы, поскольку это MPP-шная конфигурация (передача сообщений), объединяют через некоторый коммутатор на 128 входов/выходов. Были системы, в которых не хватало одного коммутатора, значит, делали ещё один коммутатор. Больше двух не делали.

Казалось бы, вот такая топология, так в чём же выигрыш? А выигрыш был очень интересный и вот в чём. Они брали реальную рабочую станцию (Work Station) на базе RS-6000 или закупали у производителя, и ставили не полную рабочую станцию, а знаменитый процессорный комплект. Куда-то ставили ещё, а в другое место ставили полностью рабочую станцию со всеми внешними устройствами и т.д. (см. Рис. 7). Тем самым получался чисто многопроцессорный MPP-шный подкомплекс (всегда только 2 плеча связь). И здесь же можно делать многомашинный комплекс. Т.е. на одном и том же оборудовании можно делать многомашинный, многопроцессорный или совместно работающий многопроцессорный и многомашинный комплекс.

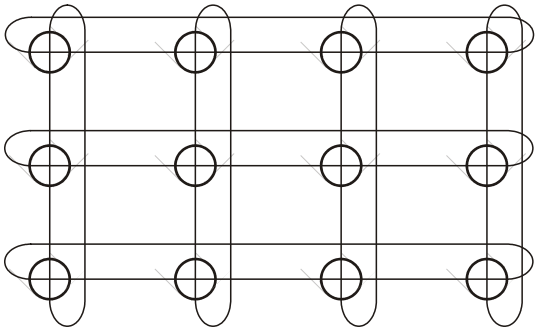
Вот такие комплексы (SP2) являются примером многопроцессорных комплексов с объединением через центральный коммутатор. Кстати, вот это не является тем, что называется «звезда». «Звезда» — это совсем другое дело.

А что такое «звезда»?



Вот такой комплекс: есть центральная ЭВМ, как правило (даже из этого рисунка видно), она более мощная, остальные ЭВМ в неё передают информацию или от неё принимают информацию. Практически, это тоже разбивается на некоторую центральную обработку и обработку front-end. Вот это — многомашинный комплекс типа «звезда».

Вот другой пример с другой топологией. Вот я имею вычислительные узлы, которые объединяются в двойной тор:



Так тор (по вертикали) и так тор (по горизонтали). Отсюда название — «двойной тор». Вот такая система SPP-1200 (2000). Эти узлы на самом деле были SMP-кирпичами. Называли их «гиперузел» из 8 или 16. А уже дальше эти SMP конфигурации обменивались (очень интересный момент) либо сообщениями (тогда это была MPP конфигурация), либо одиночными обращениями к памяти других гиперузлов (тогда это была NUMA конфигурация). И можно было комбинировать и такой и такой вариант работы. Так что это очень интересный гибридный комплекс MPP (NUMA). Вот такая топология связи получилась.

Варианты Cray тоже менялись. И он тоже перешёл на MPP-шную конфигурацию, сохраняя векторно-конвейерный подход в том смысле, что он производил процессоры векторно-конвейерного плана и непрерывно это делал, тем не менее, объединяясь с фирмой Silicon Graphics, он уже переходил на многоэлементные системы, уже не связанные с векторно-конвейерностью. Так вот, Cray T3-E(D) — это был трёхмерный тор. Конечно, система коммутации очень не простая. Раз сейчас делаются разные топологии, в том числе и такие комплексы из сотен и тысяч процессоров, значит это кому-то нужно для решения крупных задач, для параллелизма. Конечно, нужно уметь всё это загрузить, уметь достаточно быстро передавать сообщения по довольно непростой коммутационной системе. Но всё это относится к комплексу MPP, а это вот такой вот симбиоз.

Я хочу сказать, что бывают довольно разные топологии: решётки, кольца, торы (двумерный, трёхмерный).

Что касается связи процессоров, то мы рассмотрим понятия «гиперкуб» и «полный граф». Что касается комплексов MPP-шных, то есть некий прародитель «Макро конвейер», затем мы смотрим SP2, затем SPP-1200, Cray T3-E, и мы рассмотрим наши комплексы МВС 100 и МВС 1000. А дальше мы перейдём к МВС 1000М — это уже кластер.

Итак, мы рассмотрели примеры многопроцессорных комплексов (MPP), в которых процессорный узел соединяется некоторой топологией с другими узлами, при этом прямое обращение возможно только к своей памяти, а передача данных в другую память происходит групповым обменом через посредства операционной системы (т.е. передача сообщений). Мы рассмотрели некоторые топологии, а именно: топологию с центральным коммутатором — SP2, рассмотрели возможность соединения процессоров топологией двойного, тройного тора, при этом рассмотрели даже такой симбиозный вариант SPP-1200, где в топологии двойного тора узлы являются SMP подсистемами (или SMP-кирпичами) и возможна как передача сообщений, так и непосредственное обращение, т.е. вариант существования NUMA.

Мы хотели рассмотреть ещё две структуры, которые работают у нас. МВС 100 или МВС 1000 — это то, что разработано у нас, и то, что меняло друг друга в компьютерном центре. Сейчас этих систем нет, и вместо них работает другая система (работала система МВС 1000М, но сейчас и её нет) — МВС 15000М. Относительно прибавки «М» я расскажу несколько позднее.

Рассмотрим два варианта, которые реально существовали, да вообще-то МВС 1000 существует и сейчас, реально сегодня ещё работают. Вот такой интересный вариант: топология решётки — МВС 100(1000). Чем они отличались? Они отличались только типом процессора

и несколькими другими параметрами, но это сейчас для нас не так существенно. Объединялись они так:

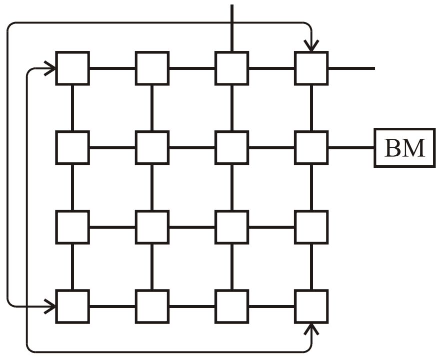


Рис. 8

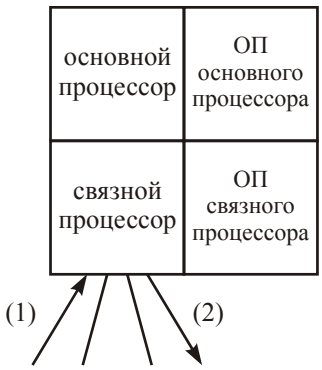
Это модуль, который считался у них модулем из нескольких процессорных узлов, который потом объединялся с другими модулями, и система наращивалась, но мы не будем на это смотреть. Этот модуль на 16-ти узлах, где каждый узел однопроцессорный. А всего такие системы наращивались до 128-256 узлов. Естественно, 2 16-ти узловых модуля объединяются в 32-х узловой, 2 32-х узловых — в 64-х узловой и т.д. Мы рассмотрим 16-ти узловой из определённых соображений. Конечные диагональные узлы соединены между собой. Таким образом, число проходимых связей всегда не больше 3-х. Оптимальный путь всегда есть и он не больше 3-х. Были проверены 2 типа передач данных в таком модуле. Ясно, что есть много свободных выходов: некоторые из них используются для подключения таких же модулей, к некоторым подключаются внешние машины (ВМ). С внешних машин передаются данные, на внешние машины передаются результаты через прохождение по этой топологии.

Как могут проходить данные? Есть масса вариантов. В начале мы чётко не подчеркнули три задачи, которые мы всё время реально отслеживали в нашем курсе: это параллелизм, это конечность/бесконечность (она явно связана с параллелизмом) и компромисс (компромисс между решениями, что сделать аппаратно и что сделать программно). В начале в самые первые годы вычислительной техники всё делалось большей частью аппаратно, но это большая нагрузка на аппаратуру. Потом с развитием самой техники появляются операционные системы, разные системы программирования, поддерживающие трансляцию и библиотеки, и т.д. Появилось «мягкое» обеспечение, его же можно как хочешь, вот на него всё и сваливали, пусть оно всё делает. Вы сами понимаете, если взять какой-нибудь гротеск, то можно сказать так, что на каком-то оборудовании в один разряд процессора и с одной операцией, вы можете решить любую задачу, но — за какое время? И можно вообще всё сбросить на аппаратуру, вообще всё что угодно, — тоже можно: всё спаять и т.д. Преобразовать матрицу — взять обратную — пожалуйста. Указывайте, где лежит матрица исходная, где лежит матрица результат, и вперёд — запускаем операцию. Сложно будет, но выполнится — это второе поле. Поэтому всегда нужен компромисс. Волны передачи функций от аппаратуры к software, и все эти волны они не случайно проявляются, потому что всегда стараются отслеживать или добиваться наилучшего коэффициента производительности (коэффициент стоимость/производительность). Конечно, где-то не жалеют никаких денег, но в большинстве случаев этот не так. И поэтому какие-то структуры годятся для решения класса задач и не нужно более быстрых аппаратных решений, а какие-то — придётся аппаратные решения использовать. Если у вас передачи идут очень частые на дальние расстояния — это один вопрос. Если передач не много и между близкими узлами — это другой вопрос. Можно организовать связь по-другому, чем в первом случае, когда между всеми [illegible].

Кстати, вот у вас работает какая-то система, решаются некоторые задачи. Приходит новая задача, программа рассчитана на 10 логических процессоров (с 1-го по 10-й), на которых должны идти 10 ветвей программы. Ну и, конечно, надо говорить операционной системе, что у меня между 9-м и 10-м — частый обмен, между 2-м и 3-м — частый, между 1-м и 10-м — очень редкий. Ты как-то так расположи, чтобы вот так. Операционная система говорит:

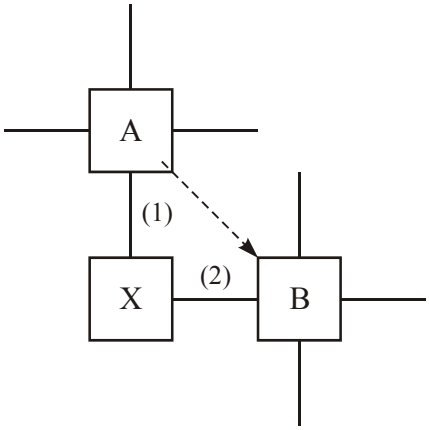
«Конечно, я постараюсь, но твои конкуренты уже заняли много места. Посмотрим». Можно попытаться переместить конкурентов, но это надо всё перегонять.

Теперь, какие же собственно 2 типа взаимосредств? Первый тип был реализован очень интересно. Давайте посмотрим структуру квадратика Рис. 8:



В нём имеется *основной процессор*, который выполняет операции, *оперативная память (ОП) основного процессора*, *связной процессор* и *оперативная память (ОП) связного процессора*. Связной процессор имеет 4 связи или, как говорили раньше, линии. Связной процессор — это процессор с внешними связями, который очень эффективно используется для групповой передачи данных (с малой задержкой и т.д.).

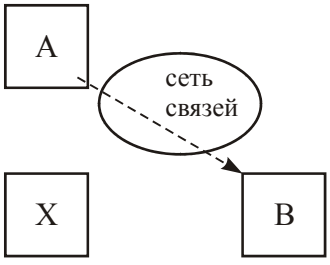
Представьте теперь, что нужно передать данные следующим образом:



По связи (1) приходят данные от модуля (A), и связной процессор помещает их в свою оперативную память. Затем, выдаст их по линии (2) связному процессору другого модуля (B). И уже этот связной процессор модуля (B), принимая данные у себя, перекинет эти данные в оперативную память основного процессора, и процессор узла (B) будет обрабатывать данные, которые ему передаёт узел (A).

Мы рассмотрели функцию промежуточного узла в этом случае. Ну и естественно, когда процессор накопил в своей оперативной памяти результат вычислений и с помощью связного процессора перекидывает их куда-то.

В данном случае мы имеем очень простую коммутацию: никакой коммутации, просто решётка и всё. На коммутацию никаких затрат нет, но мы задерживаем прохождение данных. А могли бы иметь другой вариант:



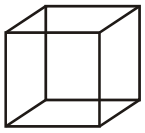
Здесь узел (X) вообще задействоваться не будет, коммутация будет осуществлена и по какому-то направлению топологии (сложной). Но это конечно существенный расход на сеть

связей, на аппаратуру коммутации и т.д. Тем не менее это, конечно, делается. Вспомните SPP-1200, о которой мы говорили, там двумерных тор и надо по всем этим путям, а в Cray-T3E — трёхмерный тор, приходится такую сеть иметь и одиночные и групповые сообщения по ней проводить.

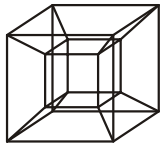
Мы рассмотрели SP-2 — центральный коммутатор, затем мы смотрели SPP-1200(2000) — двойной тор, Cray-T3E(D) — трёхмерный тор.

Теперь мы рассмотрим взаимоотношения двух других топологий: гиперкуб и полный граф соединений. И с топологией связи будет закончено.

Итак, гиперкуб ранга n: число вершин — 2^n, число дуг (связей) — $n \cdot 2^{n-1}$.



гиперкуб ранга 3



гиперкуб ранга 4

Теперь, полный граф соединений (каждый с каждым): число узлов (вершин) возьмём такое же — 2^n, тогда число дуг (связей) — $\frac{2^n(2^n-1)}{2}$.

Соотношение $\frac{\text{гиперкуб}}{\text{полный граф}}$: $\frac{n \cdot 2^{n-1} \cdot 2}{2^n(2^n-1)} = \frac{n}{2^n-1}$. Ясно, что количество связей резко растёт с разной степенью возрастания. Выгодность гиперкуба по числу связей отсюда совершенно ясна. Однако, в полном графе длина пути всегда равна 1, в гиперкубе длина пути растёт с ростом n. Всё зависит от того, как часто вы обмениваетесь, зависит от самого n. Не простые могут быть расчёты коэффициента производительности, но они используются для того, чтобы выбрать систему.

На этом мы заканчиваем многопроцессорность, но давайте тут же сразу поймём, как к этому делу относится понятие *кластер*, что это такое. Вот смотрите: многопроцессорная система. Конечно, сейчас, мы с вами уже отмечали, с активным основательным развитием систем автоматизации проектирования для любого пользователя крупная фирма соберёт, сделает, спроектирует оригинальную структуру в любом варианте (NUMA или MPP система), подготовит соответствующие средства поддержки программирования и т.д. Но в данном случае это зависит от фирмы-производителя. Конечно, это зависит от коэффициента производительность/стоимость, фирмы будут смотреть наиболее взаимовыгодный вариант. Но, обращая особое внимание на коэффициент производительность/стоимость, решили, а что если вместо оригинальной системы или даже серийной системы фирмы А, просто не быть зависимым от фирмы А и от Б тоже, взять готовые средства, которые есть в массовом выборе на рынке, и эти средства соединить между собой. Получится что-то похожее на MPP. Купить готовые изделия (дешёвые), купить имеющиеся средства связи (тоже готовые, а значит, дешёвые), и стали появляться системы, которые с одной стороны явно многопроцессорные, с другой стороны — каждый из них (вы уже видели это в SP2) — это отдельная машина со своей операционной системой, со всем хозяйством. Некоторые из них могут иметь полный набор внешних устройств, некоторые вполне ограничиваются наличием только жёсткого диска. Такие объединения стали называть *кластером*. Фактически, это нечто среднее между многомашинными комплексами и многопроцессорными комплексами. Хотя у каждого из них своя операционная система со своей памятью, но и в многопроцессорном комплексе тоже в каждом узле MPP-шном есть какие-то средства операционной системы, хотя бы для передачи данных. В каких-то узлах более основательный набор средств операционной системы (на SP2 мы всё это с вами рассмотрели). Таким образом появление кластеров стало носить массовый характер. Каждый узел может быть с несколькими процессорами. Сейчас я уже пропущу МВС-1000М, построенный по кластерной технологии, правда, узлы делались самим НИИ «Квант», а уже в МВС-15000М — купленные у IBM'а, двухпроцессорный на общей памяти SMP-кирпич. В принципе, это подход «купи и сделай сам».

Вспомним некие наши заметки, которые мы делали, рассматривая тот же Cray. Помните, в устройстве управления (УУ) появляется команда, должны быть готовы ресурсы (коды заранее переданы на векторные регистры или скалярные регистры). Если она пришла на выход УУ, и нет готового материала (т.е. векторные регистры или скалярные не пищат, что мы готовы), то ждать придётся, пока это всё будет подготовлено. Значит, желательно, чтобы это было готово, т.е. оптимизация, распределение регистров — это дело транслятора или вручную, если вы пишите на автокоде. Мы говорили с вами о перестановке команд, которую выполняет транслятор — шедъюринг. Помните, если у вас имеется

$$\begin{matrix} + & a & b \to c \\ \times & c & d \to e \\ \times & k & l \to p \end{matrix}$$

конечно, надо поменять местами последние 2. Если написано в алгоритме так, то всё равно надо поменять, потому что здесь явная зависимость. Какие-то оптимизационные действия — либо вы делаете сами, либо это делает транслятор (своими машинозависимыми действиями). Т.е. вы управляете наиболее эффективным прохождением команд. Эффективным в том смысле, что вы стараетесь параллельно запустить исполнение действий на реально имеющихся параллельных устройствах. Т.е. это программное управление параллелизмом. То же самое, когда мы говорили о зацеплении конвейера: нужно как-то сделать, чтобы результат пошёл не в регистр, а на вход следующего функционального конвейерного устройства, как-то соответствующие команды транслятором подготовить. Т.е. использование параллелизма аппаратуры во многом зависит от программирования (либо в самой программе, либо от транслятора). Вспомним слово «суперскалярный», когда (вы богатые) хороший микропроцессор содержит ещё и некоторые функциональные устройства, которые можно запустить в принципе параллельно. Во многих случаях, даже в литературе, есть такой момент. Кроме вопросов, связанных с функциональными устройствами, считается, что есть следующие 3 устройства, которые могут работать параллельно:

- устройство выполнения целочисленных операций;
- устройство выполнения вещественных операций;
- устройство загрузки/записи.

Рассуждения ведутся следующие. Да, они будут работать параллельно, но если у вас имеется задача (нить вычислений), где вычисления выполняются с целыми числами, то, вообще говоря, 1-е будет использоваться, а 2-е — простаивать. Если у вас идёт вычисление серьёзной задачи, где вы работаете с вещественными данными, то 2-е будет работать, а 1- е — простаивать. Как сделать, чтобы они по возможности работали параллельно? Это один вариант, а второй вариант, который я только что рассказал, когда суперскалярный процессор, имеется сложитель, умножитель, делитель. Как сделать, чтобы они работали параллельно?

Только что я говорил, что есть вариант программистский. А есть вариант, когда всю оптимизацию берёт на себя аппаратура. Она берёт на себя эту оптимизацию вот каким способом. Процессор имеет 2 счётчика. Т.е. аппаратно, согласно этим счётчикам команд, процессор сам ведёт не одну нить вычислений, а несколько. Называется эта техника *Hyper-Threading*. Т.е. устройство управления процессора само анализирует команды, смотрит, какие оно может выполнить из какой ветви так, чтобы было максимально загружено. Оно ищет команды из двух или нескольких таких потоков, которые могли бы быть выполнены параллельно. Т.е. мы имеем на одном физическом процессоре (микропроцессоре) несколько логических процессоров. И эти логические процессоры должна учитывать операционная система. Операционная система, ориентируясь на этот процессор, ориентирует несколько нитей одной задачи или разных задач, т.е. выполняются одновременно. Если есть устройства, которые могут работать одновременно, то по возможности они загружаются тем, что есть нити. И вот этот подход Hyper-Threading много значил. Иногда бывало так, что его делают, а для определённого класса задач выключают, потому что, например, две целочисленных ветви параллельно выполнялись хуже, чем последовательно. Вот эти системы в многих чипа реализовывались.

Отсюда несколько, может быть, противоположная — это *многоядерность*. Она не совсем противоположная.

Несколько в сторону отойду, вспомню одну историю. Виктор Александрович Поспелов — это ведущий специалист в области искусственного интеллекта. Это мой товарищ, мой однокурсник, больше того, из некоей более близкой небольшой группы. Он в области искусственного интеллекта, в области задач искусственного интеллекта, он конечно, один из крупнейших специалистов. К сожалению, последние 7 лет ему очень тяжело: он упал, пробил голову на конференции (со второго этажа на первый получилось падение), несколько месяцев комы, продолжает сейчас лечение. Он не говорит, но на это надежда ещё остаётся. 7 лет не говорит, научился хорошо говорить слово «да». Но прекрасно всё понимает. Когда он вышел из комы, то «верхнее сознание» его возродилось мгновенно: память, понимание той области деятельности, в которой он. При этом в начале он потерял возможность читать и считать, т.е. он даже не мог складывать однозначные числа. Его учили. Наконец, всё это преодолено. С ним непрерывно занимаются, идут упражнения на голову, упражнения для физики, всё это делается, но вот пока воз не движется. Дело в том, что мне удалось сподвигнуть его на один шаг вперёд. А было это так. Я к нему пришёл, мы часто к нему ходим (вдвоём, втроём и т.д.), он был в разных местах, в разных клиниках, у нас здесь в Москве клиника по восстановлению речи после черепно-мозговых травм. И вот как-то раз я к нему пришёл и один был. Как беседа проходила: я рассказываю что к чему, он вытаскивает фотографии. Закончилось всё, я выдохся. Он на меня посмотрел: «Ну!». Мы всегда, когда входим, пишем ему в тетрадке, кто когда пришёл. Я попросил книжку. «А!» Хорошо, взял книжку и начал делать надпись одной рукой (другая не действовала). Делает надпись и посматривает в эту тетрадку. И тут до меня дошло: он не может писать из головы. До этого всяческие упражнения, например, перечень существительных и надо разделить на 3 колонки (автомобили, фрукты, овощи), он всё блестяще делает, но он списывает начертания. И я понял, что пора действовать. Я подошёл, демонстративно закрыл, и сказал: «Ты же мне хочешь что-то сказать? Ты же мне только что написал послание. Напиши!» Видели бы вы, что с ним сделалось: он весь покрылся потом, вращался на каталке со страшной силой, на меня злобно сверкал, рычал буквально. Потом подъехал, взял карандаш, рука у него вся дрожала, весь напрягся и вывел большую букву «В». И опять на меня: «Гм!» Давай дальше, я опять не понимаю. Он опять промучился, но уже побыстрее и написал вторую букву «И», очень аккуратно. Я всё понял: «Извини, пожалуйста. Я очень спешил, я не успел». Он всё понял: «А…» Он хотел написать слово «вино». Мы когда приходили, мы притаскивали грамм 100 когорчику. Конечно, он ждал, а я пришёл «пустой».

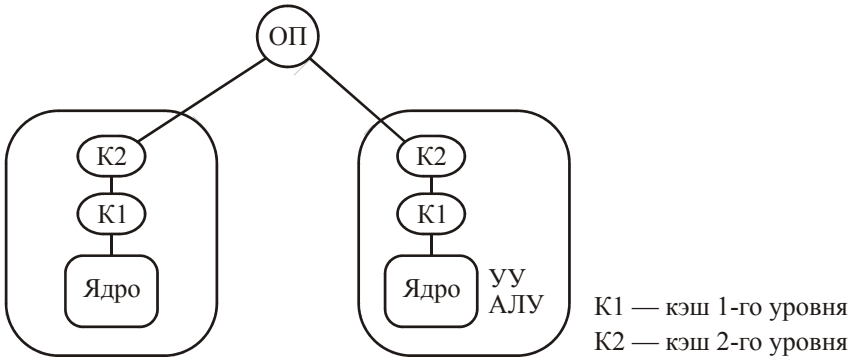
Так вот, о чём речь, здесь идёт о его докторской диссертации. Он в своё время защитил блестяще кандидатскую диссертацию, а речь тогда шла о новых вещах: о явно параллельных алгоритмах, т.е. о распределении вычислений на разные машины. Очень интересно он отвечал на вопросы. Ему задавались вопросы, как обычно, когда доклад делают. Он говорил: «Суть данного вопроса говорит о том, что задавший вопрос не понимает…»

Проходит 3 года, и он обращается ко мне: «Вот я подготовил докторскую диссертацию. Хорошо бы, чтобы ваш Институт точной механики и вычислительной техники дал бы мне отзыв как головная организация». Я посмотрел, и мне показалось, что он продолжает эксплуатировать эту же идею, не внеся чего-то такого, что соответствовало новому принципиальному решению в этом направлении. Как-то вот побольше всего, более основательно продвинута тема, но ничего такого. Что ж мне делать? Я пошёл к Лебедеву и сказал: «Сергей Алексеевич, вот тут докторская работа моего друга. Я хотел бы, что бы вы посмотрели и сказали». Взял работу, говорит: «Через недельку зайди ко мне». Я зашёл. «Знаешь, я посмотрел эту диссертацию. Ведь что он хочет? Он хочет одну задачу распараллелить на многие процессоры. А мы что хотим сделать? Собрать многие задачи в одну машину. У нас разные направления, и мы вряд ли можем быть таким сверхкомпетентным местом для оценки его результатов». Я передал, что мы вряд ли дадим отзыв, он на меня дулся года 3, но защитил диссертацию, правда, не без чёрных шаров.

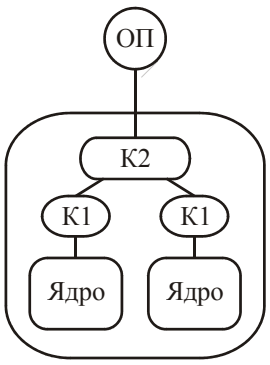
Вот тут мы, как бы, в один процессор организуем hyper-threading, устройство управления во всю старается подобрать на имеющихся ресурсах, которые могут быть параллельно использованы, соответствующие действия. Очень интересный и плодотворный подход.

И вот появляется многоядерность: в каждом ядре свой hyper-treading. Что такое многоядерность? В этом ничего принципиально нового нет.

Реально многопроцессорность:



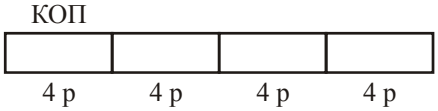
Совсем уж загрубляя, можно сказать так, что это 3 разные интегральные схемы, они все находятся на плате. В случае многоядерности происходит следующее:



Кэш второго уровня уже общий. Т.е. у вас интегральная схема двухядерного процессора. Связей стало меньше — это, во-первых, меньше по месту, с энергетикой стало получше. Т.е. целый ряд аспектов чисто технического характера приводит как к определённому ускорению, так и к уменьшению связей. А что касается нашего разговора о hyper-threading'е, то он здесь точно так же может идти как отдельный логический процессор в каждом ядре. Т.е. сама многоядерность — вещь скорее техническая, которая должна привести к ускорению, удешевлению, где удастся.

А делить физический процессор на несколько логических — это принципиальная вещь.

Теперь осталось нам немножко с RISC'ом. Это тоже ведь стремление повысить производительность, с производительностью стоимость связана. Было замечено, что для многих задач вычислительного характера используется не очень много команд. И решили сделать процессор, который в своих основных схемах имеет возможность выполнять только ограниченный набор команд (схем поменьше, а значит, и быстрее выполнять эти команды). Команда выполнялась за такт, была возможность конвейерной реализации или векторной. Все команды выполнялись за одинаковое время, данные для выполнения команд всегда находились на регистрах, заранее туда занесённые, результаты тоже помещались на регистры. Чтобы в памяти команды всегда располагались одинаково. Вот у вас команда:



Значит всего 16 операций (4 разряда под КОП). Допустим 16 регистров, тогда 4 разряда для адресации регистра. И так каждая команда, т.е. одно и тоже, всё одинаково. Отсюда можно построить молотилку более выгодную. Рабочие станции стали делать, где много вы-

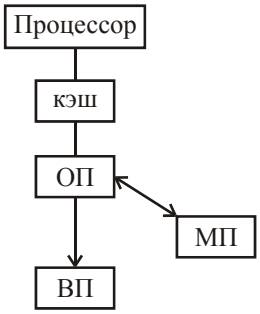
числений производится, с так называемой, RISC'овой системой команд (reduce instruction set command — сокращённая система команд). Если вам очень нужно какую-то другую операцию, вы к имеющейся аппаратуре добавляете другую аппаратуру, и тогда нужно выделить специальный один код, который укажет, что дальше будет нестандартная. Это уникальный случай для особых применений.

Естественно, что машины, которые были до этого, стали называться CISC (completed instruction set command). До этого и не подумали их так называть. Полный набор команд — это и есть полный набор команд, где есть всё, включая и такие фокусные команды, как команда косвенной адресации. Возьмём БЭСМовский пример ради интереса:

Как это всё будет работать? А очень просто: берётся исполнительный адрес первой команды ($\langle 2\rangle + A_2$) и берётся содержимое этого (в данном случае 15 последних разрядов), т.е. $\langle\langle 2\rangle + A_2\rangle$. Тогда $A_{ИСП} = \langle\langle 2\rangle + A_2\rangle + \langle 1\rangle + A_1$. Ясное дело, что здесь ещё может быть косвенная адресация, ещё и ещё. Надо найти ячейку, где находится адрес ячейки, где находится адрес ячейки, где находится адрес рабочего операнда. Как это понравилось трансляторщикам… Они, во-первых, в трансляторах стали использовать, а во-вторых, из под пера трансляторов стали выходить вот такие команды. Весь конвейер команд порвётся. Он останется, но задержки будут ужасными. Но зато всё красиво и хорошо. «Не надо делать победу науки над здравым смыслом», мы уже об этом говорили. Конечно, это тоже надо, и всё разумно в своих режимах.

Таким образом, мы с вами закончили весь первый самый большой раздел, дальше пойдёт проще.

Иерархия памяти



Имеется кэш (может быть иерархия этих кэшей), затем иеется оперативная память (ОП), затем имеется внешняя память (ВП) — магнитные диски (МД), магнитные ленты (МЛ), магнитные катушки (МК). На уровне машин типа Cray, имеется массовая память (МП), т.е. память на интегральных схемах такая же, как оперативная, но она более ёмкая, чем оперативная память (на порядок), и на порядок менее быстрая. И условно внешняя память на порядок больше, чем массовая память, и на порядок менее быстрая память.

Мы, конечно, рассмотрим работу кэшей (достаточно одного уровня). Собственно, мы с вами уже видели полностью ассоциативный кэш (кэш команд). Останется рассмотреть нам ещё 2 типа: кэш с прямой адресацией (direct mapping) и частично ассоциативный кэш. Ну и вопросы, связанные с использованием кэш при записи, чтении.

По оперативной памяти будем рассматривать вопросы логической организации — разные варианты виртуальной памяти. Два понятия «виртуальность» для памяти: 1) когда места в памяти не хватает, помещается во внешнюю память, оттуда подкачивается, а вы об этом ничего не знаете; 2) просто не на своём месте — ваши данные находятся в памяти, ваша вир-

туальная 5-я страница находится совершенно в другой. Все эти аспекты мы должны рассмотреть.

Кэш (cache)

Мы решили перейти к вопросам организации памяти и рассмотрели общую иерархию уровней памяти. Теперь осталось только рассмотреть отдельные уровни. Начнём с кэша. Собственно, мы с ним познакомились, когда рассматривали структурную схему машины БЭСМ-6, рассматривали буферные регистры слов (кэш команд) и рассматривали буферные регистры записанных результатов (кэш данных). Мы также отметили, что эти кэши организованы по одному из трёх принципов организации кэш с точки зрения организации доступа к кэш, а именно как полностью ассоциативный кэш.

Итак, имеется три способа организации доступа к кэшу, естественно, и помещения данных туда и получения данных из кэш, это

1) полностью ассоциативный кэш;
2) частично ассоциативный кэш;
3) кэш с прямой адресацией (direct mapping).

Вот три типа. На счёт полностью ассоциативного кэша: имеется кэш, который разделён на некоторое количество блоков. Все блоки в кэш имеют одинаковую длину. Как правило, количество блоков равно степени 2. С каждым блоком связан регистр, на который помещается адрес того участка памяти, данные из которого переписаны в этот блок (А_ОП), и так у каждого блока. Эти блоки могут быть больше или меньше по объёму. Те 2 кеша, которые мы рассматривали, там каждый блок был одна ячейка (одна полноразрядная ячейка). И когда у вас возникает адрес (из команды), он сравнивается со всеми этими указателями. Они могут по-разному называться, могут называться тегами.

Вообще говоря, любая память может быть двух типов:

1) адресная память, когда у нас точно указано, какое место в памяти, куда нужно обратиться (записать или считать информацию), и
2) ассоциативная память, когда вы ищете то, что вам нужно, чтобы взять или туда записать, задавая некоторый признак. По этому признаку вы ищете, где это находится. Естественно, если вам приходится долго вести такой поиск, то тратится время.

Вот, если как-то организовать так, чтобы это делать достаточно быстро, на это и направлены все усилия разработчиков аппаратуры.

Вот мы сравниваем. Если совпало, то тогда мы работаем с этим блоком кэш — записываем в него или считываем из него. Если не совпал, то тогда нам нужно взять из памяти данные. Допустим, адрес памяти (будем в восьмеричной системе счисления) — с 200_8 по 237_8, а вам нужно ячейку 215_8. А блок кэш имеет объём 40_8 (32 ячейки). Вы берёте этот адрес (215), сравниваете, и ни где нет 200-ки. А должна быть именно 200-ка, потому что вы обращаетесь в середину куда-то по адресу оперативной памяти, а блок весь должен находиться где-то здесь. Ну, или не находиться. Если находится, то всё в порядке. О стратегии записи будем говорить, считывание — естественная проблема. А что касается того, когда нет, тогда вы должны из памяти этот блок данных переместить в какой-то из блоков кэш. Какой из блоков кэш? В середине процесса жизни машины все адреса уже заполнены, и это проблема — какой выбрать. Вообще говоря, лучше выбрать тот блок, хотя тоже не факт, который был выбран из памяти и который не менялся. Т.е. если вы хотите в какой-то блок поместить блок с 200-й ячейки по 237-ю, и если этот блок изменялся, то естественно, этот блок нужно записать в ОП прежде, чем на его место записать нужный нам участок памяти. Это, конечно, накладные расходы. А если этот блок не менялся, то есть соблазн записать в него и всё. Когерентность соблюдена (начиная с этого момента, рассматривая уровни памяти, мы с вами будем употреблять этот термин — *когерентность*). Но вопрос такой: да, здесь ничего не менялось, но этот блок требовался всё время. Это рассуждение приводит к тому, что может быть, мы всё-таки не будем этот блок затирать, потому что снова он понадобиться. Тут решения могут быть разные, и алгоритмы (всё делается аппаратно) могут быть разные.

Конечно, есть где-то *бит модификации* — вначале он стоит в нуле, а если он менялся то ставится этот бит в «1», что даёт основание вызывать запись в память перед затиранием. Вообще-то это относится к любой организации кэша. Просто на примере полностью ассоциативного вот это дело рассмотрели.

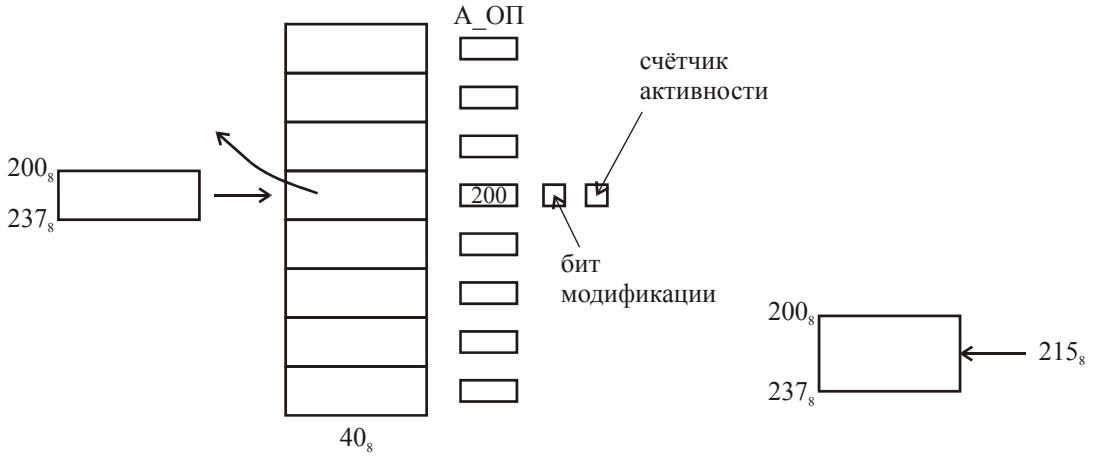


Рис. 9 Полностью ассоциативный кэш

Понятно, что здесь адреса могут быть только кратные 40_8: 200, 240, 300, 340, 400 и т.д.

Теперь рассмотрим противоположную организацию. Мы рассмотрели полостью ассоциативный кэш с точки зрения выбора блока, помещения туда данных, взятие данных оттуда. Прямая адресация:

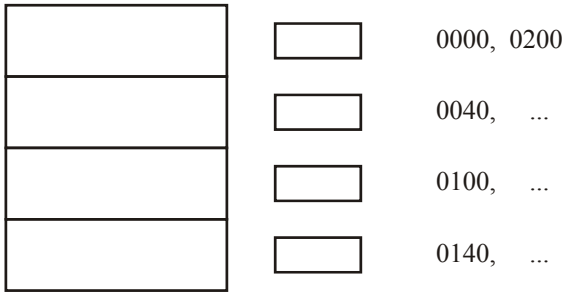


Рис. 10 Прямая адресация

Здесь достаточно жёстко. Допустим, что у нас та же длина блока: 40_8. Естественно, что в этом случае существенно проще организовано, существенно проще поиск, но здесь могут возникнуть ситуации, когда вы чаще будете менять.

В первый блок могут попасть данные, только начиная с адреса 0000. Во второй могут попасть данные, начиная с 0040, в третий — 0100, в четвёртый — 0140. Следующим претендентом на помещение в первый блок является адрес 0200. Следующий 0400, 0600, 1000 и т.д. Соответственно и дальше. Т.е. вы вычисляете блок очень просто: берёте адрес (например, 215), куда он может попасть? Только в первый. Если у вас возник адрес 215, то вы сразу приходите к адресу 200 (аппаратура знает длину блоков кэша). Если в сопровождающем регистре 200 — прекрасно, возьмём и запишем. А если нет 200? А вот тогда уж никакого другого выхода нет, кроме как вытолкнуть в память (если что-то менялось), либо не выталкивать в память, а просто сюда записать и только сюда. Вот такие будут автоматические действия в аппаратуре. Т.е. поиск очень быстрый: либо попали (совпало 200-200), либо придётся сюда 200 блок перетащить. Вот такая работа с кешом прямой адресации.

Частично ассоциативный кэш. Вот у нас 4 блока:

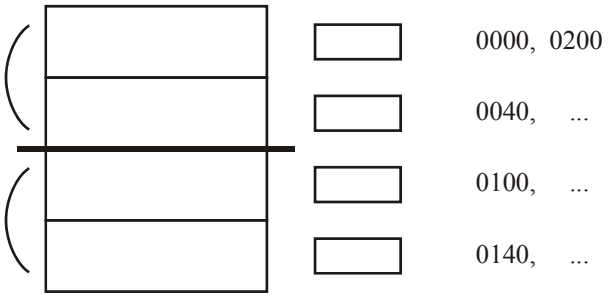


Рис. 11 Частично ассоциативный кэш

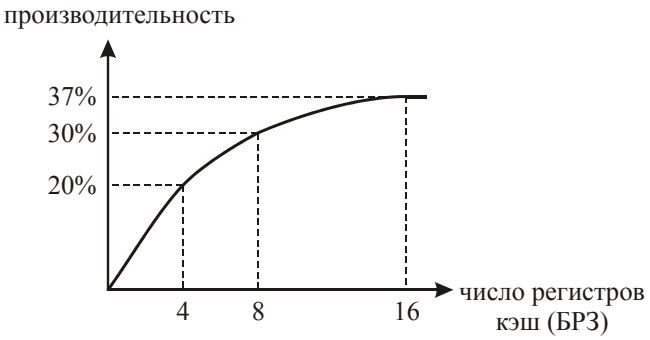
Частично ассоциативный кэш всегда делится на группы блоков. Как мы говорили, количество блоков равно степени двойки, делится на группы блоков степени тоже двойки, но меньше. Например, у вас кэш 8 блоков можно разделить на 2 группы по 4 блока, или на 4 группа по 2 блока. Мы взяли кэш из 4-х блоков, у нас тут только единственный вариант: делим на 2 группы по 2 блока. Частичная ассоциативность заключается в том, что вы методом прямой адресации выбираете группу блоков, а уже в группе блоков используете подход полностью ассоциативной. Т.е. что это значит? Что в первый блок кроме 0000, 0200 и т.д., может быть ещё и 0040, 0240 и т.д. И то же самое может быть и во второй группе. Когда у вас возникает адрес 215, вы это 200 сравниваете только с первыми двумя, больше они нигде не могут быть. Но может быть 200 и в первом, и во втором. Т.е. вы выбираете группу, в которой должно находится, и ваш адрес сравниваете со всеми в группе. Вот такая идея частичной ассоциативности (симбиозное решение). Как бы, здесь больше вероятность найти совпадения.

Мы с вами говорили о бите модификации, ну и ещё можно говорить, конечно, о *счётчике активности*. Вот это интересный момент: можно иметь ещё и счётчик активности. Когда мы с вами говорили о том, какой взять блок кэш для замещения в нём информации на нужную нам информацию. Это по-разному делают. Можно делать так. Счётчик активности находится в состоянии 0. Если у вас за какой-то период времени туда идут частые обращения, то прибавляем. Ну, это так, на предмет поиска жертвы: какой блок выкинуть.

Итак, вот это у нас 3 способа организации кэш. И даже мы обсудили вопрос замены в кэш информации какого-то блока на нужную нам информацию. Итак, время от времени из кэш данные заносятся в память. Как видите, идеальным для некоторых машин является ситуация, когда у вас данные сидят в кэш, и вы только их и используете (вы в память не обращаетесь, оставляя возможность памяти работать над приёмом данных из внешних устройств, передачи туда и т.п.). Прекрасно! Это самый лучший вариант. Это, конечно, хорошо. И существует 2 подхода реализации записи в кэш.

Прежде, чем я это скажу, ещё разочек: из наших рассуждений вы поняли, что данные кэш могут быть сосчитаны и в случае, если требуется нам сосчитать данные из некоего блока памяти, и если вы записываете, то же самое — вы считываете блок из памяти, а потом туда записываете. Если вы вспомните машину БЭСМ-6, то там так не было. Там данные попадали в кэш только при записи из арифметико-логического устройства, а затем уже брались и при считывании. А когда была следующая запись и оказывалось, что от предыдущей записи данные ещё там находятся, это же место определялось и для следующей записи. Больше того, вы помните, там происходило омоложение.

Вообще говоря, здесь тоже можно делать омоложение. В каком-то смысле вот этот счётчик активности так и направлен в эту сторону (этот же эффект). А там, вы помните, всякое обращение делало этот блок кэш (т.е. один регистр) самым молодым будь то по записи, будь то по считыванию. Можно и здесь эту схему, но она очень будет большая, если эту схему «футбольной таблицы» применить. Кстати, вы помните, моделирование показывало, что



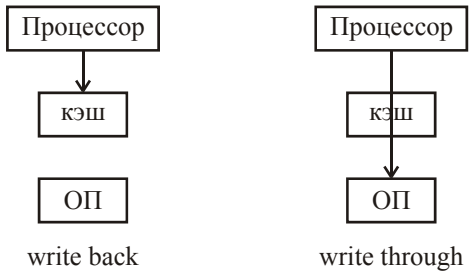
Если у вас 4 регистра, то 20% выигрыша, за счёт активного решения задач (или кусков формульных задач), где число используемых переменных небольшое. При 8-ми получалось 30%, а вот при 16-ти получалось 37%. 7% — это 70 тыс. операций в секунду — практически максимально быстродействующая машина до БЭСМ-6. Но каков был объём оборудования? Особенно, учитывая схему омоложения. Оно равно в кубе, и в те времена никак нельзя было это оборудование включить, и этими 7% пожертвовали, оставили 30%.

Поэтому возможны разные схемы, связанные с фиксацией и удержанием активно используемых данных в кэш.

Так вот 2 схемы (стратегии — страшное слово) записи:

1) только в кэш — ну и когда приспичит, когда надо уже будет, то пойдёт в память — всё, как мы рассмотрели, здесь ничего для нас нового нет, а вот
2) запись в кэш и в ОП одновременно.

Получается, что на основе такого принципа, который считается совершенно великолепным и идеальным (только что рассуждения на эту тему у нас были), при этом полностью значение кэш не исчезает. Вы потом, когда будете читать, то данные в кэш, значит, будете брать быстро. Помните, у нас даже была такая договорённость: мы считаем время работы регистров равным нулю. У вас будет читаться мгновенно, ну а в памяти — очень хорошо — там будет тоже, что в кэш, т.е. сразу устанавливается когерентность (одинаковость данных). Когда это нужно? Оказывается, это нужно для многопроцессорных систем с общей памятью (SMP). Кстати, она так и называется — *write through* (запись через).



Первая схема записи в литературе называется *write back*. Я когда это первый раз прочёл, и с тех пор я сегодня решительно возражаю. Для них (за бугром) это естественно, потому что у них другая сфера ощущений в делах своих и т.д. Для них это звучит, видимо, как-то естественно. Я просто переведу: запись обратно. А куда «обратно»? У меня протест возник сразу, как только познакомился. Почему у них протест не возникает?

Рисуем схему SMP:

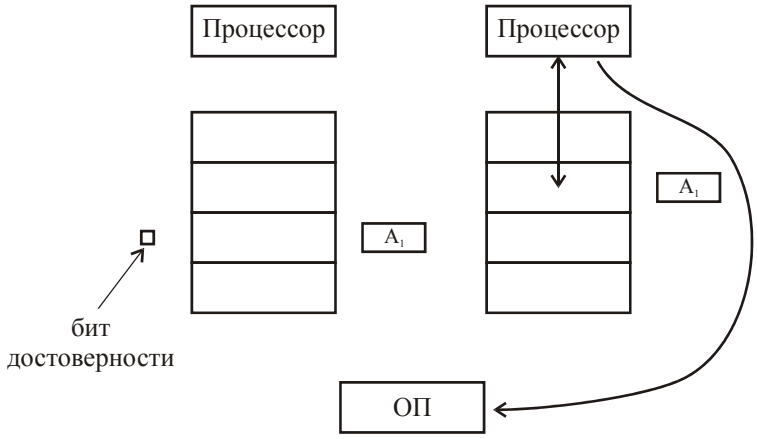
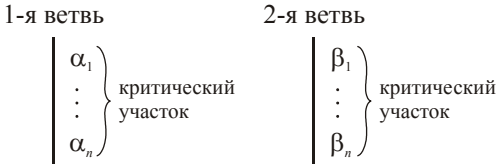


Рис. 12

2 процессора и 2 кэша. Не важно, при какой это организации: ассоциативная, прямая — всё равно. Как бы, всё хорошо, но у вас где-то в обоих кэшах есть блоки помеченные A_1, т.е. начало участка по ОП одно и то же. Почему? Потому что на этих процессорах идут разные ветви одного алгоритма одной задачи и очень может быть, что там используются общие данные. Например, для считывания. Если для считывания, то ради Бога, — здесь никаких проблем нет. А вот если записывать… Чего тут делать? Тут надо несколько порассуждать.

В литературе вы частенько можете увидеть всякие сложные схемы, чтобы последнее записанное было бы увидено и т.д. Когда это надо?

Вот у вас выполняется одна ветвь (1-я ветвь), и параллельно работает вторая:



1-я ветвь работает со своими данными ($\alpha_1, \dots, \alpha_n$), а 2-я — со своими ($\beta_1, \dots, \beta_n$). Ну и прекрасно — будет себе работать. В какой-то момент, либо 1-му процессу понадобиться работать с $\beta_1, \dots, \beta_n$, либо 2-му — с $\alpha_1, \dots, \alpha_n$. Ну и естественно, здесь возникает момент, когда все эти проблемы очень острые. Т.е. когда один процесс записал в память, тогда можно разрешить другому процессу пользовать это и наоборот. Т.е. получаем *критические участки* — там, где формируются данные, которые могут потребоваться другим процессам. Где-то на пересменке, когда уже разрешается на критическом участке (как делается защита критического участка? — методом семафоров, разными средствами программной и аппаратной синхронизации), но это может случиться в любой момент. Не то, что вы работаете (допустим, команда выполняется микросекунду) и в каждую микросекунду будете хватать данные. Да что это за жизнь? Такой жизни не бывает. Она возникнет на пересменке. Итак, всё таки такое может возникнуть, и нужно с этим делом как-то аккуратно сделать.

Что можно сделать? Вот вы записываете сюда, и тогда аппаратно проверяется (в соответствии со способом организации кэш, учитывая адрес), а нет ли его здесь (во втором кэше). Т.е. это очень сложное дело: надо проверить в первом кэше, а потом еще и во втором. И вообще говоря, на это время нужно заблокировать обращения к кэш. Если не нашли, то нет проблем: записали туда, записали в память. Когда другому процессору понадобятся эти данные, он их здесь не найдёт, возьмёт из память, а там они есть. А если всё-таки они здесь есть, то есть 2 варианта дальнейшей жизни:

1) либо запись с обновлением, т.е. то, что записали сюда, пишется и сюда, и после этого разрешается пользоваться — это технически разрешает,
2) либо запись «с обнулением». Здесь появляется новый бит (помните, у нас был бит модификации, бит и даже большее количество разрядов, связанных с активностью данного блока кэш) — *бит достоверности* (см. Рис. 12). Это означает, что вы ничего сюда писать не будете, но объявляете его недействительным. И если затем процессор

хочет что-то найти, то, учитывая этот бит, просто здесь сравнение не будет происходить, и из памяти (сюда или ещё куда-то) вытащится то, что записано здесь.

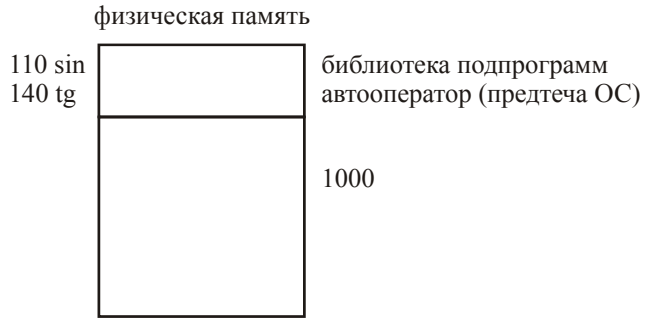
Вот есть два таких подхода: запись с обновлением и запись с обнулением. Казалось бы запись с обновлением — вроде бы всё сделано, чего ещё делать, зачем делать недостоверным данный блок кэш. Это уже зависит от разных аспектов, связанных с количеством процессоров в SMP конфигурации, естественно, с количеством кэшей. Помните, мы с вами говорили, что SMP конфигурация не очень большая — это мы аргументировали, что у нас интерливинг памяти не бесконечен, но и здесь тоже свои сложности. Поэтому на самом деле существует вот здесь 2 подхода. Запись называется «с обнулением» — блок становится недействительным, а когда вам понадобится — пожалуйста, из памяти доставайте сюда или ещё куда-нибудь. Вот стратегии записи.

И вообще говоря, мы взяли упро́щенный момент, чтобы показать при этом проблематику, которая возникает. Могут быть ещё более сложные схемы — попытки выжать максимум эффективности. Но нам хватит для обозначения проблем, которые связаны с организацией доступа к кэш и организации записи в кэш. Вот, собственно говоря, это — всё.

Я уже видел, как задавали задачки. Ну, вот у нас есть такие адреса: 1115, 241, 716 (больше, чем 7-ка нет — это восьмеричные). Вот у вас кэш с частичной адресацией, частично ассоциативный кэш. Покажите, в каких блоках могут быть эти адреса. Такая игра преподавателя на экзамене. Играть очень приятно, пустяковая, и все довольны.

Структурная организация оперативной памяти

Перейдём к структурной организации памяти (структурированная память). Вначале, естественно, память имела физические адреса, и все работали по этим адресам. Появлялись какие-то программы, которые занимали конкретные адреса (библиотеки подпрограмм).



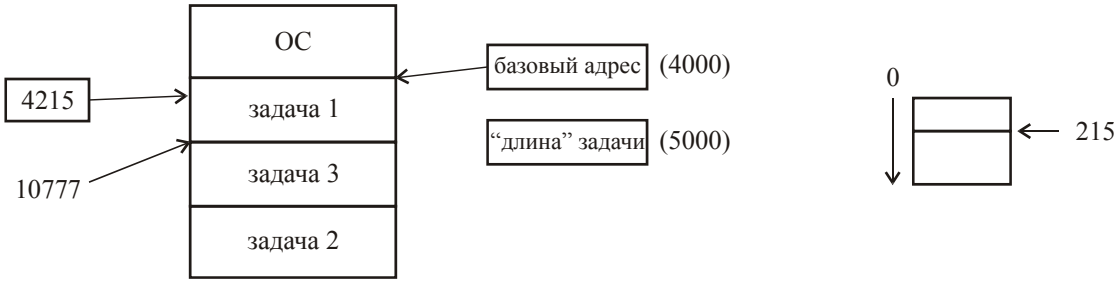
Автооператор, который в ряде случаев помогал менять задачи — некая предтеча операционной системы. А если это был человек (запускал сам программу), то он мог затереть эту память — никакой защиты не было. И здесь, с адреса 1000 пиши свои программы. Ты знаешь, что на адресе 110 находится sin, на адресе 140 находится tg и т.д. и т.д. Можешь обращаться, на сумматоре будет x, получишь $\sin x$, продолжишь дальше вычисления. Если ты, конечно, не затёр это дело. Т.е. машина находилась полностью в ведение человека, который за ней находился, т.е. все первые машины были машины персональные на тот момент, когда там работал человек.

Конечно, это было не удобно. Человек в пределах своего времени сидел, ничего не получалось, он сидел, думал — машина стояла. Потом запускал заново: перебивал карты, запускал, у него опять ничего не получалось и т.д. Конечно, это было не дело, и решили уметь помещать задачи в любое место памяти, и обеспечить тем самым для автооператора возможность быстрой смены задач.

А дальше было сделано так: вот помещаем сюда задачу 1, в некоторых случаях говорили «задание». Но «задание» нехорошо, потому что есть такой IBM'овский классический подход, что есть задание, которое вводится в машину, помещается на внешнюю память машины, задание разбивается на шаги задания, для выполнения каждого шага задания формируется задача, и при смене их выполняется задание пользователя. При этом все задачи пи-

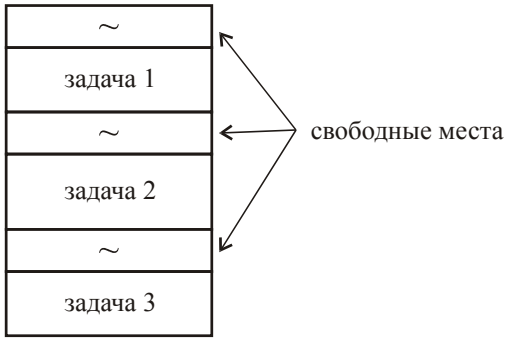
шутся с 0 до какого-то адреса. Это всё под управлением автооператора, заранее через оператора отдали перфокарты. Как раньше пользовались вычислительным центром? Имеется окошечко вырезанное, где находятся полочки. Вы на эту полочку кладёте свою колоду перфокарт и уходите. Через сутки (иногда меньше) получаете распечаточку.

Итак, задача могла помещаться на любой свободный участок памяти. И вот здесь находился регистр, в котором указывался *базовый адрес*. Естественно, он указывал на начало этого участка. И ещё один был регистр — «*длина*» *задачи*. Т.е. когда у вас возникал какой-то адрес (215) — в данном случае будем уже называть его *виртуальным*, — то он складывался с базовым адресом (4000) и получался адрес (4215), к которому и происходило обращение. Была известна длина задачи, если в результате ошибки генерировался адрес больший, чем 10777 (= 4000 + 5000 - 1), то всё — прерывание, надо выбрасывать задачу, ошибка и т.д. При этом происходит защита других областей памяти от вашей задачи.



Ясное дело, что поскольку задачи имеют разную длину, хотя не превышающую какого-то. Одна задача может выполнять какой-то обмен с внешними устройствами, в этот момент, не теряя времени, может пойти другая задача, и этим занимаются операционные системы, пусть даже на уровне автооператора. Так или иначе, реализуется мультипрограммный (или многопрограммный) режим. В данном случае, многопрограммный режим — это говорит о задачах, имеющих собственный виртуальный адрес. Когда мы будем говорить о многих процессах в одной задаче на одном виртуальном адресном пространстве — это многонитевый (multythread’овый) — много легковесных процессов.

Итак, вот таким способом (за счёт введения дополнительного оборудования) задача, идущая на процессоре, в момент её прохождения на базовый адрес заносился адрес начала информации задачи в памяти (физический), и её длина ограничивала возможность работать только в пределах информации этой задачи (вся остальная информация в памяти защищалась от этой). Ясное дело, сразу какой здесь возникал недостаток — это *фрагментация*, которая затем стала называться *внешней фрагментацией*.

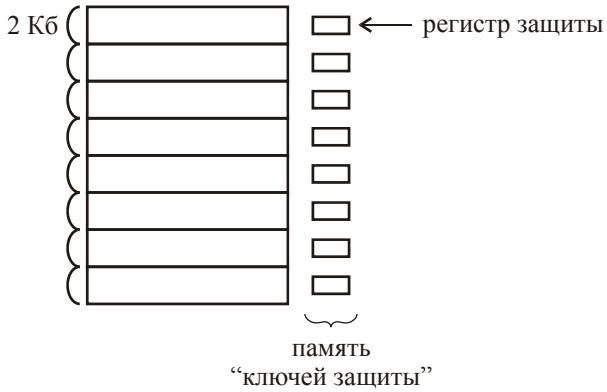


Свободные места получились исторически так. Ясное дело, что при этом хорошо защищаются и вот эти вот вещи. Возникают вот эти «дырки», т.е. фрагментация. Вам нужно 4-ю задачу запустить. В сумме этих 3-х свободных мест хватит, чтобы ввести материал 4-й задачи, но каждого из них не достаточно. Поэтому приходится всё это подвигать — целая большая работа. И потом, понимаете, какую-то задачу не закончили, вы можете её удалить на внешний носитель (допустим, всю целиком), потом её ввести, но опять нужен свободный участок памяти или опять всё подвигать. Т.е. конечно, такая организация мультипрограммного режима вызывает очень большие накладные расходы. Тем не менее, достигнут мультипрограммный режим, и появилась реализация виртуальной памяти.

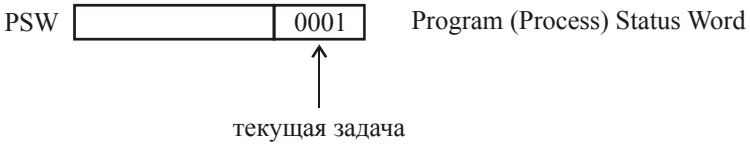
Всё это вместе называется *односегментное отображение*. Когда мы будем говорить о сегментации памяти, это совершенно другая ипостаси. В данном случае, это неструктурированная организация памяти: ваши адреса не имеют структуры, просто у вас один неструктурированный массив с 0 до 4000).

Как в пределах односегментного отображения попытались выйти из положения? Довольно интересный вариант, который был реализован в IBM 360, а затем у нас в машинах серии ЕС. Что они сделали? Повторяю, что слово «сегментация», которое у нас появится через несколько минут — это совершенно другая. Сегментная и страничная организация памяти, а также сегментно-страничная — это всё 3 вида структурированной организации памяти. А это просто историческое название — «сегментное отображение».

Вот смотрите, что было: они взяли и разделили память на участки одинаковой длины (2 Кб). И вообще говоря, этим самым избавились (или почти избавились) от фрагментации. Что именно? Они вот так поступили: они определили режим мультипрограммирования для количества задач не более 16, т.е. от 1 до 15. Нулевая — операционная система. И сделано было вот что: у каждого такого участка памяти существовал *регистр защиты*. Причём это была нормальная память, а регистр был на более быстрой памяти («на тонких плёнках» — как-то так они называли).



И этот регистрик имел всего 4 разряда, в который помещался номер задачи (допустим, у нас с вами первая задача — 0001). Первая задача попросила у операционной системы: «дай память» — даже такое обращение называлось get_mem (дай память). Она говорила: «вот тебе, пожалуйста, память», и отдавала ей физический адрес. И задача работала в физических адресах, не беспокоясь, что кого-то затрёт, потому что она получала эти адреса и в них работала. Допустим, что вот этот участок памяти был отдан какой-нибудь третьей задаче. Если задача работает в пределах своего участка памяти, она «знает» — получила адрес 4000 — что может работать на расстояние 2 Кб. Если она сгенерировала адрес физический (что возможно) не свой, то всякое обращение к адресу вызывает сравнение: сейчас какая задача работает? — первая. В некотором регистре, называемом PSW (во многих машинах существовал этот регистр), а в некоторых ещё даже называли его LPSW (long PSW). А что такое PSW? А как хотите, так и переводите: Program Status Word, Process Status Word — тот процесс (задача), которая идёт сейчас на процессоре.



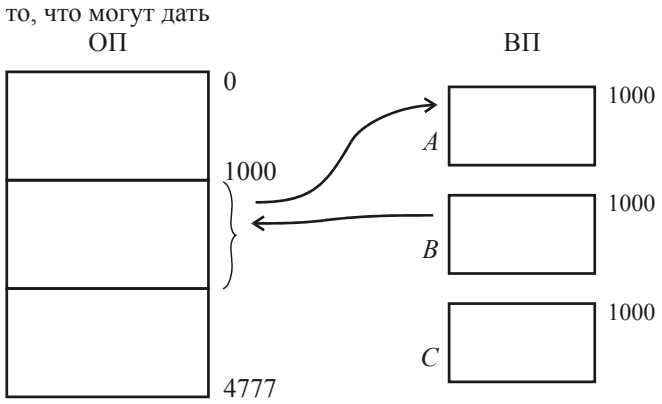
Сейчас идёт первая задача, вот здесь и находится единичка (в этом PSW). Задача работает, обращается к каким-то своим ячейкам по считыванию, по записи, и всякое обращение сравнивается со значением регистра защиты. Как только появился адрес троечка — о, нет, это нельзя. Защита. Поэтому это даже называлось *память «ключей защиты»*.

Вот она попросила — ей дали 2 Кб. Она говорит: «Дай ещё!» — «Пожалуйста, вот кусок какой-то свободный, начиная с какого-то адреса». Таким образом, информация задачи уже попадала в память не обязательно подряд (была раскидана по памяти). Она работает в

физических адресах, нет виртуальных. Т.е. исчезли как бы виртуальные адреса. Защита всех остальных участков памяти полностью организована, и вроде бы как мы достигли нового качества: мы исключили фрагментацию. Ей нужно может быть не 2 Кб, а поменьше, но это уже второй вопрос — будет, так называемая, *внутренняя фрагментация*. И после этого появилось слово «внешняя фрагментация». Но внешняя фрагментация при этом полностью исчезла. Но исчезли и виртуальные адреса. Отдельные части программы этими адресами связаны, обмениваются и прочее, и будет работать только на них. Если убрать задачу всю целиком, или даже какую-то часть её на внешнюю память (приостановить выполнение задачи) — можно, но обратно сюда же, а не куда-нибудь ещё. В односегментном отображении — пожалуйста, вы задачу можете убрать всю целиком и поместить на какое-то другое место (всю целиком). Здесь вы можете убрать кусочек и поместить его на это же место, а не куда-нибудь ещё. Это конечно, резко ограничивало. Этот момент — динамика со своей стороны подкачала. Становится неудовлетворительной, когда вам нужно действительно какую-то задачу приостановить, вытащить другую, потом эту. Здесь резко ухудшилось состояние динамики.

Это тоже относится к односегментному отображению. У вас нет структуризации. Конечно все эти затруднения и этого варианта и другого, привели к структуризации оперативной памяти, и сейчас мы её рассмотрим.

Итак, мы говорим, что нам выделили под задачу 5000 ячеек. Представим, что задача у нас существенно больше — 100000 ячеек. Нам сказали: «Прости, у нас вся память 20 тыс. Тебе только 5, дай и соседу поработать». Что же делать? Как решать такую задачу? Что, раньше не решались такие задачи? Конечно, решались. Делали это даже очень просто. Вот у вас есть адреса с 0 до 4777. Вы где-то у себя программируете модуль *A*, начиная с 1000. И модуль *B* вы программируете тоже, начиная с 1000. И когда модуль *A* отработал, то вы его переписываете во внешнюю память (ВП), и сюда подкачиваете модуль *B*. Вы занимаетесь вот этим качанием, вы выкручиваетесь и на меньшем объёме виртуальной памяти реализуете работу своей большой задачи, программируя на одни и те же виртуальные адреса. Такие модули, имеющие одни и те же адреса, стали называться *оверлеями* (*overlay*), а процесс называли свопингом, оверлеенгом и т.д. Т.е. это делал сам программист, он не мог избавиться от таких оверлеев. Даже в той модели, где он запрашивал «дай ещё, дай ещё, дай ещё», память всё равно была (физическая) небольшая, а у него задача — большая. И поэтому всё равно в любом случае приходится заниматься вот этими оверлеями.



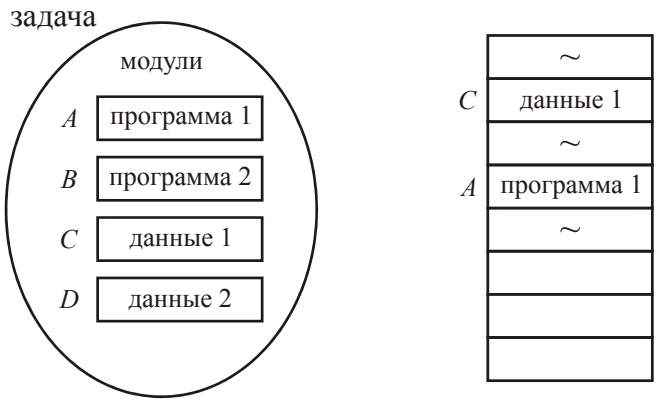
Так вот структуризация памяти позволила полностью избавиться от оверлеев. Эта структуризация памяти привела почти параллельно к двум вариантам: сегментной и страничной.

Сегментная организация памяти

Итак, сегментная и страничная организация. А поскольку имеется сегментно-страничная, то отсюда ясно, что и сегментная, и страничная организация много дали чего. Во-первых, полностью исключили оверлей. Возникла виртуальная память на полный объём

вашей памяти, т.е. вы считаете, что у вас машина на такую длину памяти, а машина на память в 10 раз меньше. И без вашего участия, с помощью поддержки операционной системы, ваша виртуальная память существенно большего размера, чем физическая, реализуется на этой памяти. При этом возникали недостатки как сегментной, так и страничной организации памяти. И вот, объединив достоинства и ликвидировав недостатки, они дали сим метод построения, естественно, несколько более сложный, чем каждый из этих. Но не на много, и большинство машин обладают этим методом организации.

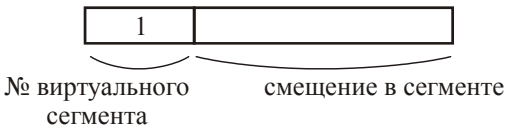
Итак, что такое *сегментная организация*? У вас имеется некое количество модулей:



И в память помещаются по мере необходимости эти данные, т.е. это ваши сегменты. Понадобилось — вы куда-то поместили модуль *A* (программа 1). Понадобилось — куда-то поместили *C* (данные 1). Т.е. куда угодно помещаются, причём они разной длины. Важно, что сегменты разной длины, такие, какие требуются: сделали программу на 10 ячеек — значит, 10, сделали на 3000 — значит, 3000. Если модуль данных 143 ячейки, значит 143. Я говорю «ячейки», в IBM'овских были байты.

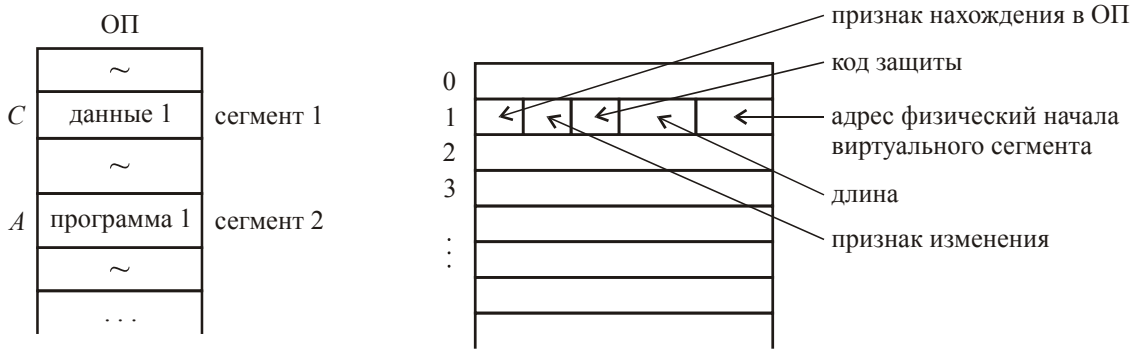
Сегменты разной длины. И отсюда, вы понимаете, они помещаются в память на любое место. Если там что-то есть, надо убрать (если это вам надо) — это дело операционной системы, это не ваше дело. А *D* я ещё не написал, может ещё их нет. По мере необходимости они проявляются. Если нет места, чего-то убрали, кого-то поставили, а потом кого-то, кого вы убрали, понадобилось снова, пришлось убрать ещё кого-то — начинается *замещение сегментов*, в простонародье — *подкачка* — подкачка информации сегментного характера.

Что же такое *сегмент*? Сразу появилась внешняя фрагментация (они разной длины). Так зачем же мы так сделали? А вот для чего. Чтобы иметь возможность уникально защитить каждый из этих модулей. Что это значит «уникально» защитить? Пожалуйста. У вас программа, значит, писать в неё нельзя. А это данные, в них писать можно. А вот в эти данные, а вот в них нельзя. Вот в эти данные вы можете писать, а вам не разрешают. Эту программу вы можете использовать (и вы и вы), а вам не разрешается и т.д. Т.е. у вас возможность есть уникальной защиты объектов программы. Уникальная защита объектов программы реализуется очень хорошо. Таким образом, существует некая таблица, находящаяся в ОП, где каждая строчка этой таблицы соответствует номеру сегмента. Все сегменты виртуальные, они нумеруются от нуля и т.д., и когда возникает ваш адрес, он структурирован: некоторая часть разрядов указывает номер виртуального сегмента, а другая часть — смещение в сегменте.



По номеру виртуального сегмента (например, 1) со смещением 1 берётся строчка таблицы, находится адрес физический первого виртуального сегмента, где расположен первый виртуальный сегмент. И здесь находится *код защиты*. Таким образом, в этой строчке кроме адреса физического (и вы, таким образом, легко преобразуете виртуальный в физический, а именно берётся адрес физический и прибавляется смещение, и можно залезать в память, если код защиты это разрешает), есть *признак нахождения в памяти*, код защиты показывает

можно работать. Происходит преобразование виртуальный адрес → адрес физический с помощью вот такой таблицы:



Естественно, таблица сегментов находилась в памяти. Вы в виртуальном адресе выделяете номер сегмента, находите информацию об этом сегменте и, проверяя наличие в памяти, разрешённость данного типа обращения, всё. Вот так организована сегментная память.

Вроде бы здесь всё очень просто, операционная система заботится и об этом и об этом, в место может быть любое помещено, но при этом есть проблема внешней фрагментации. А также есть уникальная защита, всё идеально, кроме внешней фрагментации. И ещё: прежде чем обратиться в память по существу, вам пришлось обратиться в память служебным образом, т.е. у вас получились накладные расходы 100%. Как выйти из этого положения? А выйти из этого положения очень просто. Есть табличка на регистрах (регистров было 4), у которой 2 части: номер виртуального сегмента (например, 1), и всё остальное — признак нахождения в памяти, код защиты, признак изменения, длина и адрес физический начала виртуального сегмента.

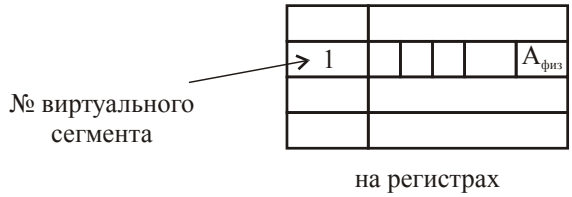


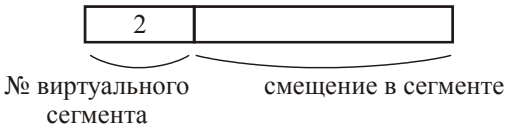
Рис. 13 Таблица адресной трансляции

Что происходит? У вас какой сегмент — первый. Вы что делаете? Ассоциативно ищите и находите. А дальше всё то же самое. Всё точно так же. Получилась на регистрах ассоциативная память, т.е. кэш — самый настоящий кэш. Почему здесь достаточно бывает небольшого количества регистров? Из-за свойства локальности программы. Например, перемножение матриц: один объект, второй объект, результат и сама программа — всего 4 сегмента. Вам больше и не надо. А вот если не совпало, то можно лезть в основную таблицу. Больше того, всё это можно поручить операционной системе.

Дальше мы увидим организацию страничной памяти, где будут абсолютно похожие вещи, т.е. именно будет таблица страниц. Вместо таблицы страниц, для того, чтобы не залезать в память, появится такая же таблица аппаратной трансляции. Ну и будет один недостаток — это пропадание уникальной защиты, зато от внешней фрагментации мы избавимся полностью. Попытка избавиться от двух неприятностей (внешней фрагментации и оставить уникальную защиту) привела к сегментно-страничной организации, которую мы тоже рассмотрим.

Мы с вами рассмотрев организацию кэшей и, перейдя к организации виртуальной памяти, рассмотрели односегментную организацию виртуальной памяти. Затем, правда, рассмотрели вариант одноуровневой памяти (без структуризации) — это то, что было сделано для IBM — деление на разделы (участки) памяти длиной 2 Кб, где каждому такому участку был приписан ключ защиты (специальная память ключей защиты). И на регистре процессора, где находился PSW — слово, характеризующее проходящий на процессоре процесс, там фиксировался номер данной задачи, которая сейчас решается. А у всех выделенных ей час-

тей в памяти ключей защиты было то же самое. Поэтому, если вы обращались по всем своим участкам, получали их физические данные (тут был отход от виртуальной памяти, но в пределах общего односегментного отображения), всё было в порядке. Если только вы пытались по ошибке попасть в какой-либо участок памяти, то там был другой номер (ключ защиты, соответствующий этому участку содержал номер другой), он не совпадал (а проверка всегда происходила) с номером задачи в PSW, и конечно, было прерывание и не разрешалось. Мы рассмотрели сегментную, потом частично-сегментную память, когда у нас с вами адрес делился на 2 части: виртуальный номер сегмента и смещение в сегменте.

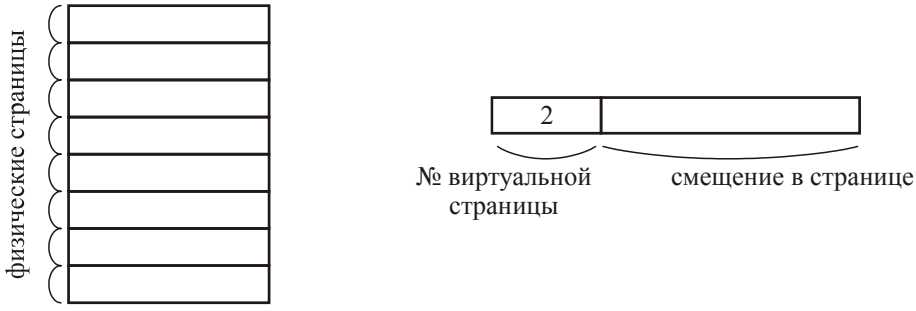


Такой подход был, при этом сегменты были разной длины, в них помещались законченные объекты программы (будь то программный модуль или модуль данных) и тем самым появлялась возможность для уникальной защиты отдельных сегментов (можно было указать, кому чего можно делать). Тем самым вы используете структурно организованную память и получаете такие важные возможности защиты отдельных объектов программы. Но при этом, поскольку модули разной длины, возникала фрагментация. Эта фрагментация возникала и при односегментной организации памяти между сегментами, принадлежащими разным задачам. В IBM пытались уйти от этого, так и ушли, но там появились другие неприятности.

Существует таблица всех сегментов в ОП, аппаратура знает начало этой таблицы. А в принципе, можно иметь *регистр-префикса*, т.е. какой-то регистр, который указывает, где находится эта таблица, а операционная система, делая таблицу, помещает на этот регистр начало таблицы, либо она закрепляется для аппаратуры на всю жизнь (тогда не нужен регистр). Так или иначе, номер (допустим, второй) ищется, находится строчка этой таблицы, соответствующая номеру сегмента. В этой строчке находится всё, что нужно. Во-первых, адрес физический начала сегмента, длина этого сегмента, чтобы вы могли к адресу физическому прибавлять смещение, никогда не выйти за пределы. Что ещё? Признак нахождения сегмента в ОП (сегмент может находиться во внешней памяти), код защиты. Всё просто, но служебное обращение является накладным расходом. Чтобы избавиться от этого (мы это рассматривали), была введена *таблица адресной трансляции* (см. Рис. 13). Из-за свойства локальности программы достаточно небольшого количества регистров. Фактически, это служебный кэш — ассоциативная память. Тогда у вас исчезает обращение к таблице сегментов, только очень редко. Это можно делать аппаратно, а можно поручить операционной системе взять и сюда (в таблицу адресной трансляции) поместить. А вот что здесь будет выталкиваться — это вопрос интересный — это тоже кандидатские и докторские диссертации.

Страничная организация памяти

Как и в случае, когда мы делили на участки одинаковой длины (2 Кб), здесь тоже появляется деление на одинаковые участки. Но в этом случае, адрес структурирован так же, в нём находится *номер виртуальной страницы* и *смещение в странице*.



Здесь всё то же самое, т.е. есть *таблица страниц* данной задачи (для другой — своя таблица страниц), и здесь всё точно так же, только для страниц:

Что здесь? Номер физической страницы или адрес начала, что то же самое, поскольку вся память разделена на страницы одинаковой длины (в широком пределе от 0,5 Кб до 4 Кслов, в разных машинах по-разному). В БЭСМ-6, которую мы рассматривали, там страничная была организация и там длина таблицы была 1 Кслово (= 6 Кб). Признак нахождения страницы в памяти. Длина известна, поэтому переползая на другую страницу — пожалуйста, нет проблем. Признак изменения. Всё. Что мы теряем? Теряем код защиты, потому что в физическую страницу может быть помещено несколько объектов программы (несколько модулей программы, несколько модулей данных). Так что мы потеряли это дело, за счёт того, что мы полностью избавились от внешней фрагментации. И даже называется такая память со страничной организацией в литературе — динамически распределяемая память.

И здесь, конечно, нужно как-то замещать. Какой? Вот тут начинаются вопросы. Имеются на этот счёт и книги всевозможные, есть стратегии замещения. Стратегии — это в смысле либо вы ждёте, когда вас «ударит» (вы не найдёте здесь, увидите, что страницы нет в ОП, она на внешней памяти, тогда надо её подкачивать) — это называется «замещение страниц» по научному или по жаргонному — «подкачка страницы» по запросу. Есть с упреждением. А тактики — самые разные. Например, FIFO (первый пришёл и первый, к сожалению, на выход) либо по давности использования (Last Recently Use), опять же надо отмечать активность этой страницы, это вполне может сделать и аппаратура, счётчик активности ввести в этой же строчке этой же таблицы. Это тоже делается, и выталкивается самая плохо посещаемая страница. Есть много других, есть оптимальные алгоритмы, и графики на этот счёт приведены. Это много расписывается в литературе, здесь много людей постаралось.

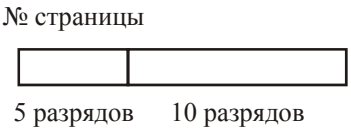
В своё время вышла так называемая «Энциклопедия информатики». Я не помню сейчас, может быть, лет 10 назад, может 8-7. В институт к нам попала рукопись (оригинал текста) этой энциклопедии по информатики. Мы взглянули на эту энциклопедию, и у нас возникло очень беспокойное чувство, мы так хотели предпринять какие-то действия, но было уже поздно. Пока мы просмотрели, сказали, что вот эти главы написаны просто блестяще. Эти главы студентам можно давать, какие там школьники. Они написаны так хорошо, что и школьники поймут, и студенты. Всё будет замечательно. В частности и вот эта глава про замещение страниц. Картинка очень понравилась: комната, там сидят за столом игроки и играют в карты, есть две двери, в одну дверь встречают с улыбками, а в другую выкидывают страницу из памяти. Это замечательная совершенно карикатура, очень образная. А другие главы — просто дикие ошибки (больше программистские, с рекурсиями и т.д.). Выяснилось, что кое-кто решили подзаработать, поскольку тема современная, родители своих детей постараются обеспечить. Они устроили «пирамиду» через каких-то знакомых, те своих знакомых, и раздали кому-то написать. Кто-то написал блестяще, а кто-то похалтурил. Всё это вместе сложили и мгновенно издали. Издать сейчас как угодно можно — пойдите за угол и тут же у вас тираж. Поздно уже — всё разошлось. Вот такие вещи бывают.

На счёт поздно и у нас бывает. Вот книга Королёва последняя, которая вышла в издательстве Вычислительного центра, по структурам ЭВМ, была очень нужная. Потом мы начали (он меня попросил), советуясь, её чистить — просто в написании есть дефекты, и эти дефекты могли приводить к двусмысленному пониманию, вообще случайно неправильно написанные фразы, не просмотренные и неверные. К сожалению, многое осталось. А это уже

всё — у него план, а мы не укладывались во время его плана. Так что вот такая получилась картина.

Иногда говорят, что здесь недостаток, кроме того, что мы теряем уникальную защиту объектов программы, может быть в том, что у вас внутренняя фрагментация. Конечно, память выделяется страницами под задачу. Если у вас задача помещается всего в одну страницу и содержит, в случае БЭСМ-6, 30, 40, ну 100 ячеек и всё, то да — попадает. Но, как правило, большие задачи и пропадание части последней страницы имеет мало значения. Момент внутренней фрагментации имеет место, но не столь важный.

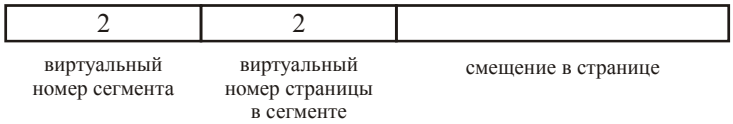
Кстати, с БЭСМ-6 получилось очень интересно. Долго не было памяти достаточного объёма, потом она появилась, но машины не были на неё рассчитаны. А виртуальный адрес всего 15 разрядов, и в нём



В 10-ти разрядах можно указать любую из 1024 ячеек. А раз адрес 15-ти разрядный и страниц виртуальных таким образом появляется 32, то в машине существовало 32 регистра 15-ти разрядных (т.е. вот эта табличка страниц). Так вот такая табличка существовала сразу на все страницы данной задачи, т.е. фактически такая таблица страниц помещалась в аппаратуру полностью. Если у вас очень много виртуальных страниц, то понятно — здесь у вас 4, 8. Здесь было 32, причём левая часть была не нужна, потому что всегда нулевой виртуальной страницей был нулевой регистр.

Сегментно-страничная организация памяти

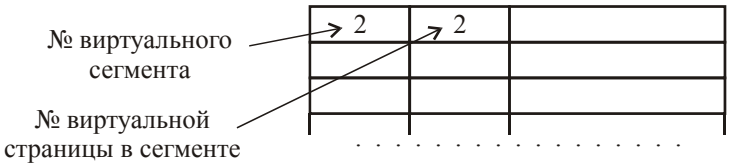
Всё то же самое, как вы понимаете, только что попытка ликвидировать оба эти недостатка. А именно, за счёт страничности обеспечить динамическое выделение памяти, а за счёт сегментности обеспечить защиту объекта, который в этом случае располагается в нескольких виртуальных страницах. Таким образом, виртуальный адрес сегментно-страничной организации у нас будет: виртуальный номер сегмента, виртуальный номер страницы в сегменте и смещение в странице.



Тут надо сделать важное замечание: мы с вами рассматриваем некую каноническую структуру. В реальных машинах (зарубежных) есть очень сложные организации. А мы с вами эту общую идею представляем в некотором общем варианте, которого в чистом виде нет, но зато понятно. У вас есть виртуальный сегмент, который содержит какое-то количество виртуальных страниц. В сегменте они нумеруются, начиная с нуля.

Дальше происходит следующее. У нас имеется таблица сегментов. Здесь теряется длина (некое количество страниц просто), и адреса начала сегмента памяти тоже не существует. Начала сегмента нет, поэтому здесь находится *адрес таблицы страниц*.

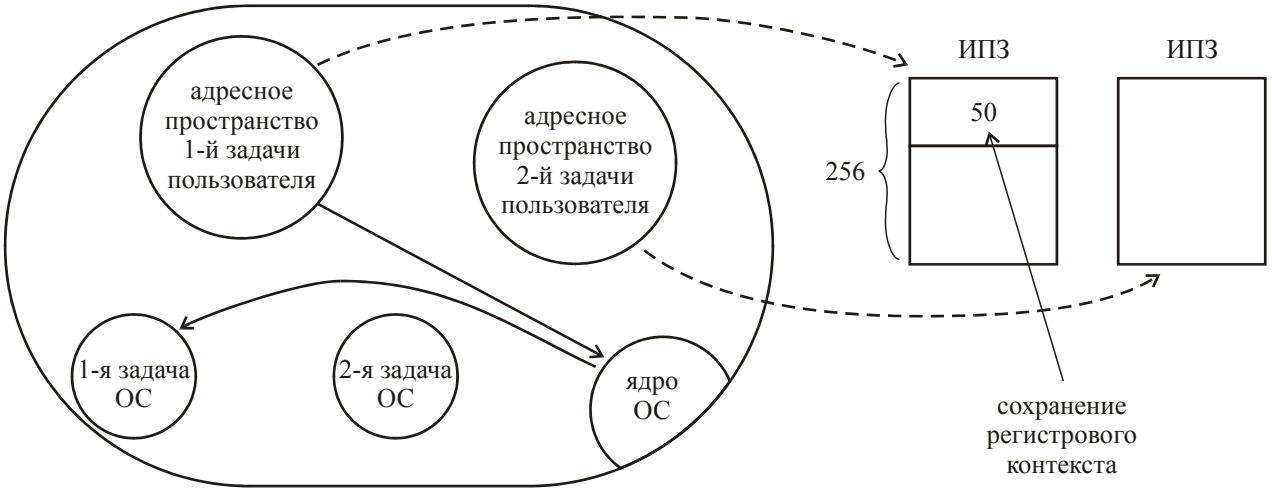
Получается (n + 1) таблица страниц, где n — количество сегментов. Двойное приходится делать обращение в ОП — 200% накладных расходов. В таблице адресной трансляции существует всё то же самое, только первое поле делится на 2 части:



В третью колонку из таблицы сегментов приходит код защиты, а из таблицы страниц — всё, что там есть: номер физической страницы, признак нахождения в памяти и модификация.

Был ли у нас разговор о том, как организационно реализуется (в виде каких организационных структур) операционная система в памяти? Понятно, что задача есть задача, естественно, есть возможность защиты. А вот как операционная система сделана организационная? Наверно стоит об этом сказать. Здесь, как бы, кстати, об этом идёт речь, потому что сейчас мы рассмотрим такую интересную вещь, как определение физического адреса при нахождении таблицы сегментов и страниц в виртуальной памяти.

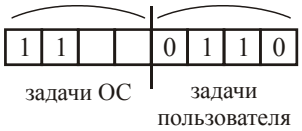
Вот у нас есть задачи:



Естественно, задачи защищены друг от друга. Есть какая-то информация о каждой из них, которая находится в специальном поле, которое называется по-разному. Может называться *описателем задачи*, может называться *блоком состояния задачи*, или, что в наших операционных системах (ОС) часто использовалось, *информационное поле задачи* (ИПЗ). Что здесь может находиться? Ведь задача может временно приостановиться по какой-то причине. Причины могут быть разные, например, выполняется обмен какой-то страницы с внешней памятью, задача продолжает работать параллельно с обменом и налетает на страницу, с которой обмен ещё не закончен, надо подождать, когда он закончится. Значит, задача приостанавливается, пойдёт другая задача. Либо вам какой-то квант даётся (всем даются одинаковые кванты). Ваш квант кончился — задача приостановилась. На момент приостановки задачи, чтобы её потом можно было продолжить, надо сохранить регистровый контекст (например, индексные регистры, которые были на БЭСМ-6, в IBM были регистры общего назначения (РОН) — их тоже надо сохранить, в Cray — тоже надо сохранить), естественно, адрес возврата. Возьмём, например, БЭСМ-6. Регистров там не много. Сколько там индексных регистров? 15. Какие-то ещё там служебные и т.д. В целом набегает порядка 50. А дальше-то что? Всего — 256 ячеек. А всякие сведения об управлении этой задачи, об ожидаемых событиях, о местах, на которые нужно передать управление в случае, если эти ожидаемые события произойдут, о ресурсах внешней памяти и ещё Бог знает о чём. Всякие эти вещи, действительно надо много места, и вот такое информационное поле задачи (ИПЗ).

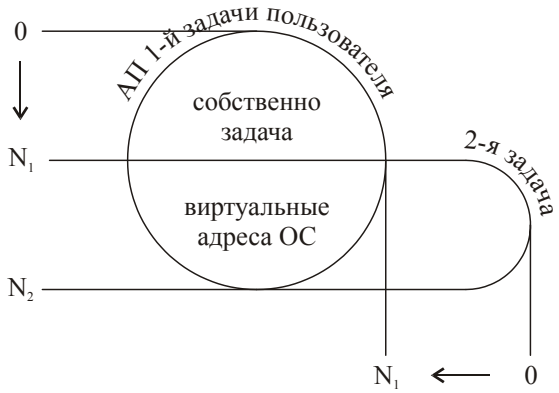
Как же с операционной системой? У операционной системы есть какие-то функции. Функция один реализуется как такая же точно с такими же информационными полями зада-

ча ОС (1-я задача ОС). Есть вторая задача ОС. Т.е. они имеют так же своё адресное пространство (у каждой своё адресное пространство) и работают точно так же, как задачи пользователя. Их можно так же отлаживать. Только они при выполнении имеют больший приоритет. Естественно, они возникли из-за потребностей задач пользователя. Даже можно себе представить вот такую *шкалу готовности задач к решению*:



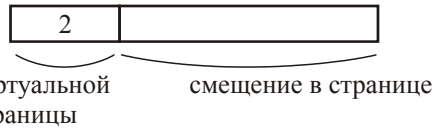
Эти все разряды указывают готовность задач (ОС и пользователя). И вдруг возникло обстоятельство: закончился обмен или сама задача подала обращение к ОС, значит должна выполниться одна из задач ОС. Может быть, в процессе она активизирует ещё какую-то задачу. По этой шкале всегда можно смотреть, какую задачу (приоритет слева направо). Что-то не входит в такую организацию, а именно, ядро ОС.

Есть второй способ организации, а именно, вот какой. Вот у вас есть адресное пространство (АП) первой задачи пользователя.



Какая-то часть адресов — это собственно задача (с 0 до N_1), а с N_1 до N_2 — это виртуальные адреса ОС. Т.е. операционная система (ОС) входит в эту задачу. При этом она должна от всяких безобразий и ошибок программы не пострадать. Т.е. она входит как защищённая подсистема. Итак, это защищённая подсистема в общем адресном пространстве задачи ОС. Операционная система вошла в задачу пользователя, осталось только ядро. Теперь возьмём вторую задачу. Т.е. у неё тоже имеется свой диапазон виртуальных адресов и какой-то диапазон адресов ОС она тоже в себя включает. Вот так организационно может располагаться информация операционной системы. Эти задачи так построены, что не используются в них команды, которые являются запрещёнными для пользовательских задач (команды управления аппаратурой, непосредственное помещение на управляющие регистры здесь нет, а всё это находится в ядре ОС).

Так вот в этом случае что у нас возникает. Пусть у нас страничная организация и возникает адрес



Естественно, мы ищем в таблице адресной трансляции. Не нашли (нету здесь двойки), тогда мы должны (как договаривались) идти в таблицу страниц. А таблица страниц находится на некоей физической странице, а у этой таблицы она виртуальная и её может не быть в ОП, значит заменять просто так нельзя. Таким образом нужно по номеру виртуальной страницы обратится в некую таблицу (в ядре ОС) виртуальных страниц ОС, где указано про нашу страницу всё (её статус, есть, нет и на какой физической странице она находится). Если нет, то надо подкачать. Т.е. у нас 3 обращения в ОП. Какое может получиться максимально? Нужно подкачать эту страницу, если она откачена. Но тогда может быть придётся откачать что-то, т.е. 2 обращения к диску. Затем, таблица появилась, может выясниться, что страница

задачи откачена, и её надо подкачать. Таким образом, порядка 4-х обращений, всё остальное — о-малое от этого времени. 4 обращения к диску — это максимально возможное (минимально — нуль). Кстати, в машинах VAX фирмы DEC вот так вот это дело организовано.

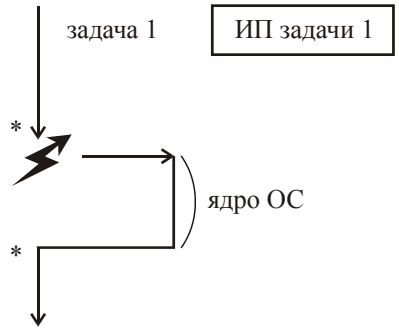
Средства поддержки операционной системы

Теперь, мы как бы рассмотрели вопросы организации памяти. Давайте поговорим о средствах поддержки ОС. Некоторые из них мы рассмотрим, связанные с переключением задач (процессов) и с организацией защиты динамических знакомств модулей. Итак, первое — это будет у нас поддержка переключения задач, второе — многоуровневая защита и третья — динамическое знакомство программ. Это некоторые из средств, включаются в аппаратуру и помогают осуществлению целого ряда функций операционной системы, в том числе вот этих вот.

Переключение процессов

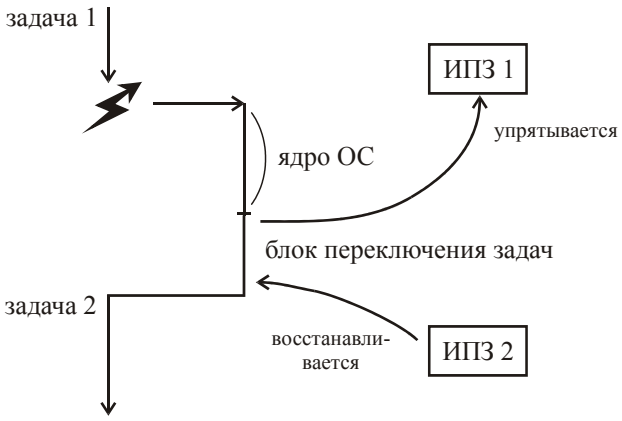
Сразу нужно здесь посмотреть, как происходит переключение процессов, какие могут быть действия. А действия могут быть следующие.

Вот у нас идёт какая-то задача — задача 1. У неё будет прерывание, либо произошло обращение к задаче ОС. И то и другое вызывает переход к ядру ОС. Куда переходим в это ядро — несколько позже, пока — общее. Это ядро ОС выясняет, что за необходимость, при которой к нему произошло обращение. Давайте простую вещь рассмотрим, например, это прерывание по времени: аппаратурой определилось, что прошла единица времени (например, 1/250 секунды в машине БЭСМ-6), берётся и прибавляется единица к счётчику, и смотрится: кто-нибудь ждёт этого момента? Никому ничего не надо — надо выходить обратно и продолжить, естественно, с этого самого места.



Для выполнения ядра ОС что-то сохраняется аппаратно, потом всё восстанавливается. Будем условно говорить, что в этом месте мы вообще не прерываемся. Это первый вариант.

Второй вариант: снова идёт задача, снова происходит прерывание, снова работает ядро ОС, и вместо возврата на задачу принимается решение о переключении на другую задачу. Например, если это было как раз обращение к запрещённой области памяти, то нужно приостановить задачу, а значит перейти на какую-то другую из готовых к исполнению. Либо это какое-то дано задание ОС, которое оформлено как задача. Тогда при переходе к *блоку переключения задач*, после которого вы переходите на задачу 2 (первая задача прекращается). Ясно, что всё информационное поле первой задачи будет упрятано, а второй — восстановлено.

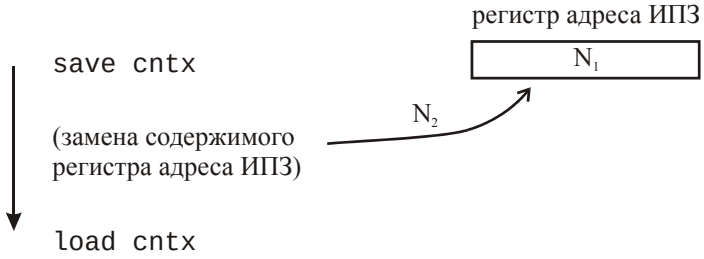


Вот всё что требуется для продолжения задачи, которая когда-то была приостановлена. Вот так происходит второй вариант.

Всё вместе взятое называется *стержень*. Он определяет программный процесс, который может быть переключён на другой программный процесс. Некая программа, которая процессом не является. Она обеспечивает переключение процессов, сама процессом не является и называется программным стержнем.

Какие же поддержки могут быть для работы блока переключения задач?

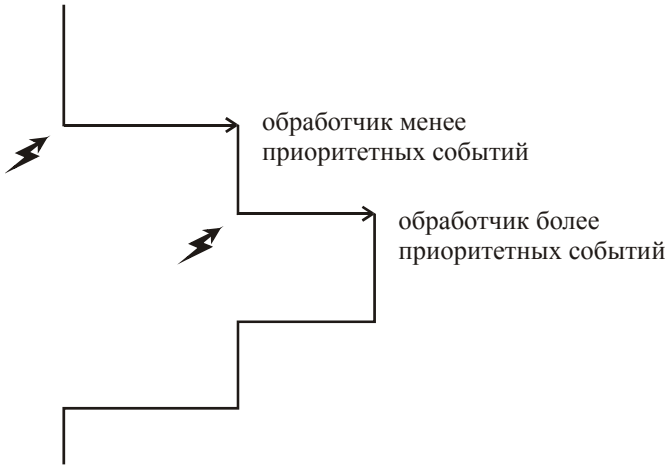
Возьмём машину фирмы DEC — VAX. Там были такие поддерживающие команды: `save cntx` (сохранить контекст). А вот куда? Это просто команда. Так вот, на всё время работы первой задачи имеется регистр — *регистр адреса ИПЗ*. Таким образом, команда `save cntx` грузит состояние регистров вот сюда. Затем произойдёт выбор новой задачи, и сюда будет помещено (после команды `save cntx`) N_2. И дальше пойдёт команда `load cntx`, которая и произведёт вот эти действия.



Наличие команд `save` и `load cntx` и регистра адреса ИПЗ можно сделать в самой операционной системе, что и сделано в БЭСМ-6.

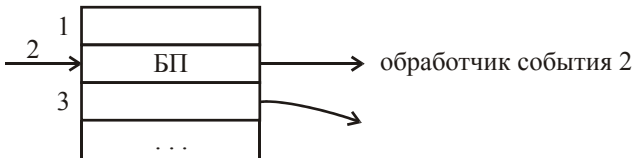
Если в процессе обработки прерываний, на самом деле действий по обработки информации, связанной с появлением события. Что такое прерывание? Прерывание — это переход от выполнения одного процесса к выполнению какой-то другой программы в ядре ОС при возникновении некоего события. Как могут быть зафиксированы эти события — это чуть позже. А сейчас — можно ли переключаться во время обработки прерывания или нет?

В IBM существовали *обработчики прерываний*. Они обрабатывали группу прерываний. Было несколько обработчиков прерываний, и каждый из них был фактически как задача. Было так: имелось какое-то количество событий, в результате выполнения события появлялось прерывание, прерывалось выполнение программы, и фиксировался номер этого события. Были приоритетные, менее приоритетные и ещё менее приоритетные. Было 2 обработчика или 4.



А если это событие было этого же ранга или менее, то мы ждали до конца, после чего перешли на обработчик. Так что эту часть тоже можно организовывать с прерываниями.

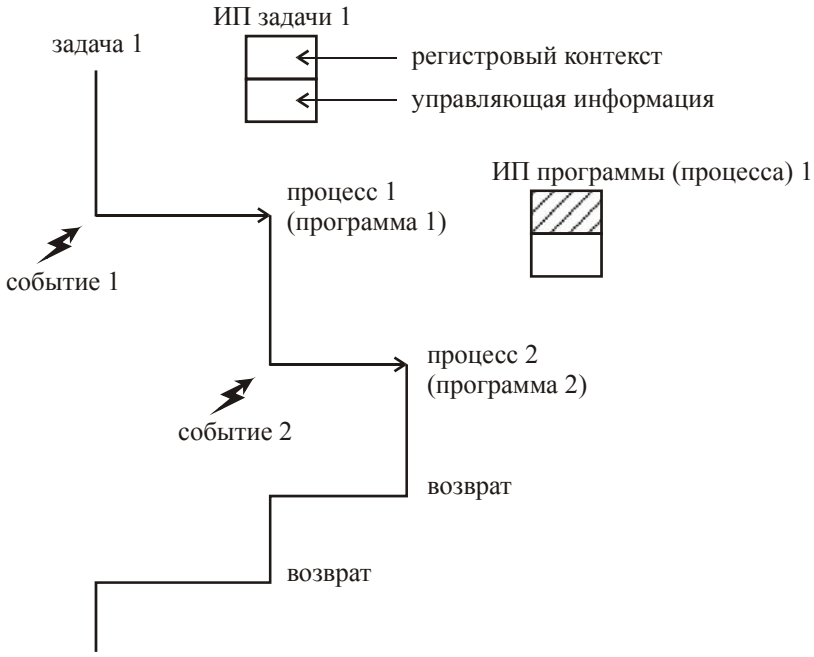
Теперь, как же собственно реализуется само прерывание, как выходим на эту программу, на этот обработчик. Каким-то образом фиксируется номер события. И у нас возможны разные варианты. У нас имеется некая таблица (в памяти или на регистрах) по номерам событий — *таблица номеров событий* (или иногда говорят «прерываний номеров событий»). Пусть нулевого события не может быть. Вот у нас определилось второе событие. Тогда происходит переход на вторую строчку, а здесь переход безусловный (БП) на обработчик события 2:



Может быть, как в случае БЭСМ-6 сама операционная система по некоторой информации о событиях определяет номер события (аппаратно этого не делалось) и передаёт управление на функцию какой-то программы. Можно сказать, что это дешифратор реакций на события или дешифратор прерываний. Были ещё некоторые названия этой таблицы — *вектор прерываний* или таблица векторов прерываний (что не верно). Что такое вектор — это массив данных. В данном случае — это массив переходов на какие-то адреса, можно назвать вектор этих переходов (каждый переход есть компонент этого вектора). Опять же, употребляя жаргонное слово «прерывание» — вектор прерываний. Ещё называют вектора. Здесь стрелка, здесь стрелка. Этих стрелок много, значит вектора.

Ну ладно, пока что мы по нашей тематике рассмотрели только аппаратные средства, поддерживающие переключения. Аппаратуру прерываний, аппаратуру фиксации информации о событиях мы рассмотрим позднее, и перейдём к вопросам многоуровневой защиты и вопросам знакомства программ.

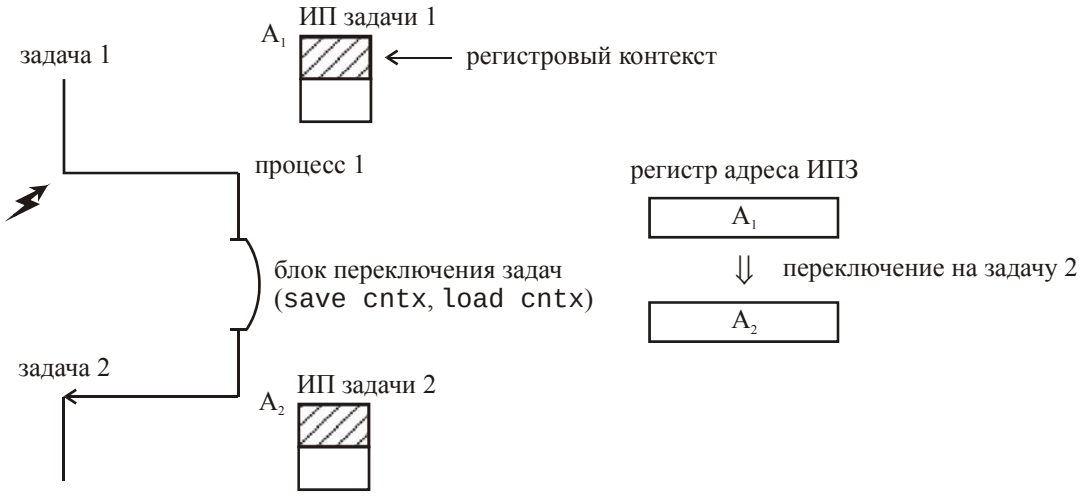
Мы зафиксировали аспекты прерываний прохождения какого-либо процесса (переход на процесс операционной системы по обработке информации связанного события, в результате которого произошло прерывание). При этом мы отмечали, что у нас возможность после обработки данных, связанных с событием, т.е. после появления прерывания в программе действий по обработке события, если все действия в ней завершаются, мы можем выйти на продолжение. Поэтому есть варианты: эта программа не прерывается совсем, какие бы события не случились, или она прерывается, мы переходим на ещё какой-нибудь процесс операционной системы обработки данных, связанных с другим событием, не закончив.



В любом случае необходимо иметь некое поле для *упрятывания регистрового контекста*, хотя бы для этого, относящееся к задаче 1 (помните, ИПЗ — информационное поле задачи), где должен упрятываться регистровый контекст, ибо он будет использоваться в этих процессах (по обработке прерывания). Вообще это поле большое, и там находится всякая управляющая информация, которую содержит операционная система о задаче (ожидаемые события, о реакции на эти события, сведения о ресурсах, всё, что угодно). Нас интересует сейчас вот это вот, когда произошло некоторое событие, возникло прерывание, мы перешли на некую программу операционной системы или процесс. Здесь ещё одно информационное поле (ИП) программы (процесса). Сюда мы тоже упрячем информацию на этот момент: регистровый контекст — состояние регистров на этот (вообще говоря, случайный) момент. Ну а затем командой возврата.

Мы отмечали, что в IBM в знаменитых IBM'овских сериях существовали вот такие обработчики прерываний (называются не программы, не процессы, а обработчики прерываний). Их было несколько, они обрабатывали группы прерываний. И если возникло прерывание более приоритетной группы, то тогда мы переходили на другой обработчик более приоритетных событий. Если нет, то этот запоминался, и когда дообрабатывалось событие, по поводу которого произошло прерывание, а затем этот обработчик уже обрабатывал следующее событие, которое за время обработки первого события появилось, но на него среагировали в конце, по очереди их обрабатывая.

Мы с вами отмечали, что естественно, эти упрятывания регистрового контекста могут происходить аппаратно, может происходить и даже программно. Например, в машине БЭСМ-6 прятался только адрес возврата, а вот здесь уже происходили все моменты, такие небольшие циклы упрятывания. Но это один вариант жизни в машине самой задачи, которая выполняет действие, связанное с возникшим событием (каким угодно: внешним, внутренним событием). Внешние: окончание обмена, окончание приёма данных по каналу или выдача. Какие внутренние события: неприятные какие-нибудь события, связанные с переполнением, деление на нуль, взятие операнда, который оказался плохой, испорченный. Так что разные прерывания. Именно вот в этом случае могут быть разные действия, когда у вас плохой операнд: либо задача должна быть выброшена, либо передано её управление по указанному ей адресу, будет работать программа по обработке нештатной ситуации в задаче. Такое тоже возможно. Но в любом случае, другая последовательность такая. Давайте попроще — без прерывания в прерывании:



Помните, какие команды здесь у нас были: была команда `save cntx` — аппаратные средства для поддержки переключения (упрятывается всё), затем выбирается задача другая, и тогда уже наоборот — из информационного поля другой задачи `load cntx`. А вот вместе с выбором меняется внешний *регистр адреса ИПЗ*. В нём находился адрес ИПЗ до самого переключения. Потом мы приняли решение переходить на другую задачу (например, пришло прерывание по времени, и квант, который выделялся на эту задачу, закончился, и теперь будем отдавать квант процессорного времени другой задачи, или просто закроем задачу, если было обращение к области памяти, с которой осуществляется обмен с внешним устройством, и пока он не закончится, работать с ней нельзя, задачу надо приостановить). Так что здесь по-разному может быть ситуация, важно, что мы решили выбирать вторую задачу и меняем на A_2. Существует, естественно, такое поле A_2 — ИП второй задачи. Команды `save cntx` и `load cntx` ориентируется на этот регистр адреса ИПЗ. Поскольку там было A_1, то по команде `save cntx` всё упрятывается в ИП первой задачи. А на момент `load cntx` там находится A_2, и мы попадаем либо на начало задачи 2, либо на продолжение задачи 2, если она была приостановлена. Понятно, что вместо этого может действовать сама операционная система, не используя никаких средств аппаратной поддержки. Всё зависит от возможностей аппаратуры.

Это всё мы с вами отмечали, и ещё отметим, что переход происходит из-за того, что произошло событие 1, и имеется некий дешифратор переходов на программы обработки данных, связанные с событиями. И вот здесь, допустим это будет какой-то адрес N_1, значит здесь стоит переход безусловный (БП) на N_1. Если событие 2, то стоит БП на N_2 и т.д.

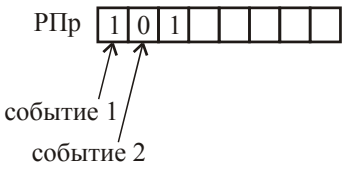
Как только обнаруживаем, что произошло событие 1 (это может быть аппаратно или программно), так или иначе, происходит переход сюда, а отсюда уже сюда. Почему нужен такой дешифратор? Вы здесь можете менять: программу реакции на событие 1 можете поместить в другое место. Поэтому очень удобно: аппаратно или программно попадает на первую ячейку (если событие 1), а отсюда переходит сюда. Вот такая вот история. И т.д. Это всё называется *вектором прерывания* или *набором векторов прерываний*, имея в виду просто стрелки. Что такое стрелка? Это вектор. Или это всё вместе вектор, а это компоненты. Но это всё игра слов, используемая в литературе.

Вот вопрос: как же мы переходим вот сюда? Что за аппаратура (у нас это есть, вот мы с вами посмотрели «Специальные регистры и команды процессора для поддержки обработки прерываний и переключения процессора» — это мы рассмотрели — 14-й вопрос, а теперь переходим к 12-му, потому что теперь будет понятно, для чего она нужна эта аппаратура).

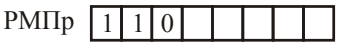
Аппаратура прерываний

Обычно они (события) как-то зафиксированы в аппаратуре. Они могут одновременно появиться (физически одновременно появится в пределах каких-то долей микросекунды). Один из вариантов — это *регистровая аппаратура фиксации прерываний*.

Нарисуем некоторый *регистр прерываний* (РПр):

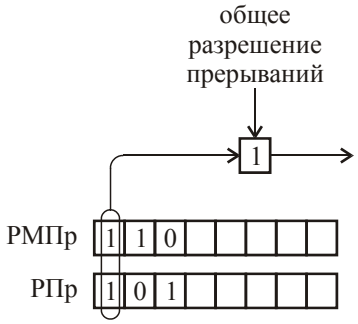


Всякое событие, если оно возникло (внешнее, внутреннее), отображается появлением единички в этом регистре. Наше любимое событие 1 возникло, но и одновременно возникло событие 3. Далее существует регистр (как видите, этот регистр управляется аппаратурой, не операционной системой; в принципе, конечно, может управляться операционной системой, но это для отладки — вы можете имитировать события, даже если они не появляются). А вот это вот *регистр маски* (РМПр):



Что это означает? Он точно такой же, как и регистр прерываний. В нём проставляется единичка. Это означает, что мы ожидаем и разрешаем (мы — это значит операционная система) реакцию на событие 1, разрешаем реакцию на событие 2 и не разрешаем реакцию на событие 3. В нашем случае, появилось событие 1, событие 3, реально произойдёт реакция на событие 1.

Теперь, кроме того, имеется ещё и *общее разрешение прерываний*:



Появляются вот эти ситуации, идёт как раз определение того, что это событие 1, и дальше переход на ту самую строчку, о которой мы с вами говорили. Если бы определилось событие 2, соответственно, определился бы номер 2.

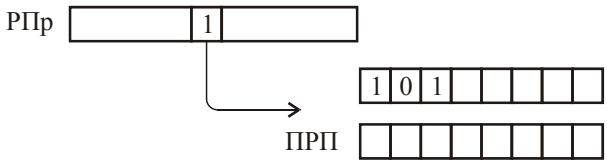
Когда эта штука может быть в нуле? Она может быть в нуле, только в момент переключения задач. Т.е. это тот самый стержень, который есть программа, процессом не являющаяся. То, что является процессом, может быть переключено на другой процесс. Эта программа процессом не является и не может быть переключена ни на что в процессе своей работы.

Либо мы пожёстче сделаем и попроще для операционной системы. Можно, чтобы программы обработки прерываний были бы короткими, и во время них не происходило бы прерывание. Для машины БЭСМ-6 так оно и было: эти программы всегда были короче. Потом, когда будем смотреть работу с внешними устройствами, ещё раз будет понятно почему. В пределах 100 команд (100 мкс) разрешалось без прерываний, а потом либо надо было переходить обратно, либо нужно было переходить на другую задачу пользователя, если задача приостанавливалась и не могла продолжаться, или возникла необходимость выполнить более приоритетную задачу.

В чём хороша и в чём плоха такая аппаратура? Хороша она в том, что можно так расположить здесь фиксацию событий, чтобы события, на которые надо реагировать самым

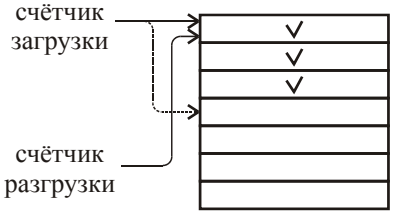
срочным образом, находились бы в голове. Операционная система что делает? Она считывает вот этот регистр, делает логическое умножение (т.е. конъюнкцию) вот этих двух величин (из РПр и РМПр), получает результат и выполняет команду выдачи номера старшей единицы. Такие команды вполне существуют в машинах: выдача старшей, младшей, количества единиц в коде — в универсальных машинах всё это есть. И вот, пожалуйста.

В чём плохо? В машине БЭСМ-6 существовал такой регистр на 48 разрядов. Хорошо, у вас возникает 49-е событие и что? А делается буквально следующее. Выдели здесь разряд, на который происходила сборка всех тех событий, которые появлялись и фиксировались в *периферийный регистр прерываний* (ПРП), тогда этот был *главный регистр прерываний*. Здесь, естественно, своя маска. И когда здесь появлялось хотя бы одно, а может несколько, событий, все они приходили на этот регистр.



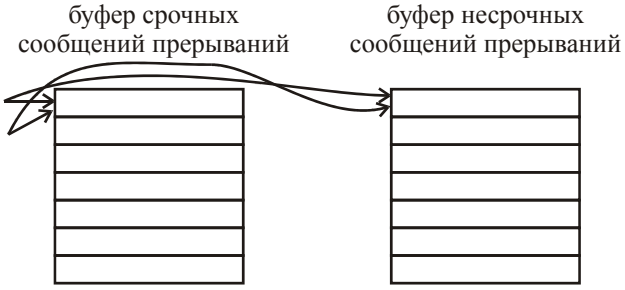
Вот такая регистровая система имеющая преимущества и недостатки.

Теперь второй способ. Имеется *буфер «сообщений — прерываний»* в ОП. Как вы сами понимаете, отсюда преимущество — можно иметь сколь угодной длины. Его начало зафиксировано в аппаратуре, или как всегда на любую аппаратную таблицу или структуру данных, которая может исследоваться по своим отдельным строкам, может всегда указывать адрес в каком-нибудь регистре, который обычно называют *регистр-префикс*. Сюда поступает информация о событиях. Во-первых, вы сразу видите, если она поступает сюда целой строкой (сколько-то разрядов, довольно много), то конечно, существенно отличается от одного разряда. Иногда, правда, информации о событии так много, что она ещё куда-то в дополнительный регистр попадает или ещё где-то в памяти объявляется массивом данных, но это второй вопрос. У вас естественно есть *счётчик загрузки* и *счётчик разгрузки*, т.е. какие-то регистры. Возникло одновременно допустим 3 события. Они поместились согласно счётчику загрузки. И счётчик загрузки стал указывать на другое место.



Раз разошлись (счётчики загрузки и разгрузки), значит, возникает прерывание — появились события, на них надо реагировать. А как? Вот их 3 здесь, что делать? Поручается операционной системе реагировать на события с учётом приоритета.

Это некоторые логические решения, которые где-то употреблялись, где-то не употреблялись, по-разному. На самом деле, это было актуально, и есть машины, где было 2 буфера:

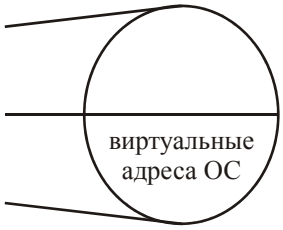


Сначала исследовалась ситуация расхождения счётчиков в буфере срочных прерываний, а уже потом — в буфере несрочных. Как видите, имеются всякие преимущества и сочетаются с некоторыми недостатками. Например, наличие сообщения в буфере сообщений прерываний можно отобразить единичкой в разряде какого-то регистра и получить некий

симбиоз систем фиксаций событий. Т.е. каким-то образом фиксируется событие, и затем на них аппаратно или программно происходит реакция.

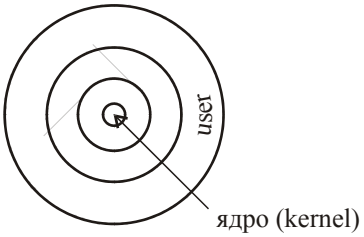
Защищённая подсистема. Многоуровневая защита

Вот вопросы защищённых подсистем, многоуровневая защита. Время от времени такие машины появляются. И хотя бы даже 2 уровня, когда у вас, помните, вариант жизни операционной системы, когда в виртуальной памяти задачи находится операционная система (ОС):



Виртуальные адреса ОС как продолжение виртуальных адресов задачи. Т.е. задача может, задав виртуальный адрес, взять и попытаться записать что-то сюда и испортить. Этого не будет. Эта система защищена. Посмотрим, как такая защита осуществляется. Т.е. должны быть защищённые подсистемы.

Ну и проще всего это представляется как кольцевая защита, где этих уровней, вообще говоря, очень много. И действительно, существовали системы, в которых имелось 8 уровней (причём 4 уровня для пользовательской задачи и 4 уровня для ОС) или 16 уровней защиты. Скажем, ваше ядро работает (центральная часть) и ваше работает. А вы только начинаете новую вещь. Вы ставитесь на тот уровень, что если вы чего-то попытаетесь испортить, это не удастся, можете только у себя. Таким образом, нарушить функционирование вы не сможете до поры до времени. Т.е. это для развития программного комплекса (причём как одна задача), там могут защищать одни части от других. Туда же идут все операционные системы — достраивание операционной системы, развитие — тоже могут быть защищены отдельные части. Скажем, у машины фирмы Digital Equipment Corporation (DEC) существовала такая хитрая вещь: там было 4 уровня, было ядро (kernel), затем шёл supervisor или superuser (или наоборот), а дальше шёл user.



Возникает необходимость иметь защищённые подсистемы. Как это реализуется? Это реализуется следующим образом. Что значит «необходимо иметь защиту», обращаться-то всё равно надо к программе и попросить выполнить функцию, которая находится вот здесь. Это должно осуществляться. Обратится просто так или по считыванию — это будет запрещено. А на определённые зафиксированные точки входа, вы должны иметь возможность. Таким образом, у вас существует уровень защиты (1, 2, 3, 4).

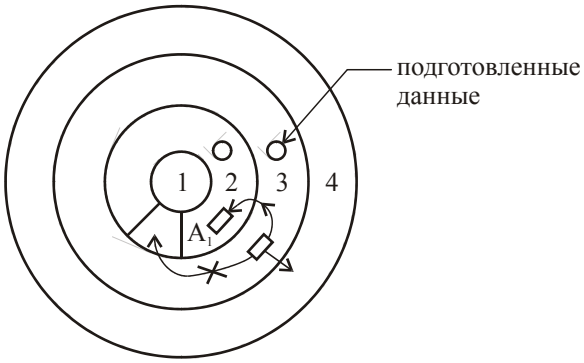
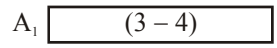
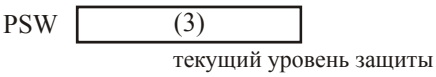


Рис. 14

Когда вы обращаетесь куда-то, у вас имеются некоторые *описатели входа* (находятся на регистрах или ещё где-то, пока не будем говорить). И вот здесь (в описателе входа) будет указан диапазон уровней, из которых возможно обращение.

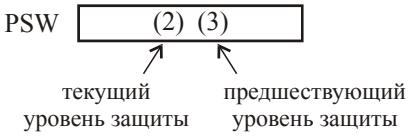


Скажем, здесь будет 3-4. Значит, и отсюда и отсюда можно обращаться. А как знать где вы находитесь (на каком уровне)? Это уж совсем просто. Имеется вот этот самый регистр PSW (мы говорили о нём тогда, когда исследовали решение с разделением памяти на участки постоянной длины и наличие специальной памяти ключей защиты). А сейчас будем читать *текущий уровень защиты* (на нашем рисунке он 3):



Теперь, поскольку память мы посмотрели, нам особенно хорошо понятно, почему это можно посчитать. Мы обращаемся в какую-то ячейку (допустим, что это страничная организация), и в описателе этой страницы есть сведения про уровень защиты. Какой у нас? 2. Если я вдруг, работая здесь, взял и полез сюда: 2, ага, а ты кто? — ах, ты 3-й! (см. Рис. 14) И естественно, возникнет событие. Нельзя работать, возникнет прерывание и все действия, которые с этим могут быть связаны. Т.е. это всё очень просто решается вопрос, и уже вам никуда нельзя будет обратиться. Конечно, при этой системе на внешний уровень вроде как можно. Но можно усложнить систему и это тоже не разрешить (только свой уровень). А можно так, что эти уровни могут обращаться сюда, например, уровни ОС могут обращаться к уровню пользователя и что-то там изменить, потому что ОС работает для пользователя, и если пользователю что-то понадобилось, то это можно сделать с изменением чего-то у пользователя, и это должна уметь делать операционная система. Можно сделать и такой и такой подход, что можно менять только у себя или можно менять на более низких уровнях, на более высоких — будет всё запрещено, а обращения на разрешённые точки входа будут.

Ещё один момент: вот мы вошли в этот модуль A_1 и начинаем в нём работать. Всякое обращение — указываются какие-то параметры по значению или по ссылке. Допустим по ссылке, значит вы указываете, где находятся параметры, которые этот модуль будет обрабатывать. Допустим, что вы эти данные подготовили вот здесь (3-й уровень — см. Рис. 14). А если вы ошиблись и указываете данные вот здесь (2-й уровень). Вы уже работаете в этом модуле, и у вас уже стоит 2, и когда вы обращаетесь к указанному месту, где находится параметр, и здесь будет возможно. А на самом деле — это ошибка. Значит, тут будет *предшествующий уровень защиты*:

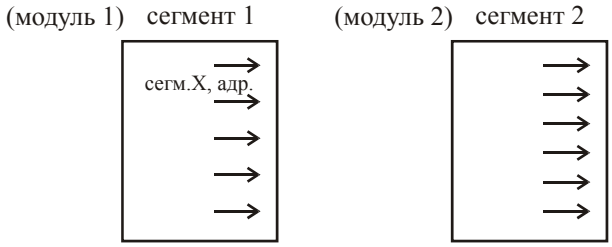


И здесь будет работать специальная команда — обращение к предшествующему уровню защиты. И тогда эта команда прочтёт: здесь будет 2-ка, а предшествующий уровень — 3-ка — нельзя. Вот такие вещи. С помощью таких решений обеспечивается создание

ка — нельзя. Вот такие вещи. С помощью таких решений обеспечивается создание защищённых систем и их использование в жизни задачи и работы операционной системы.

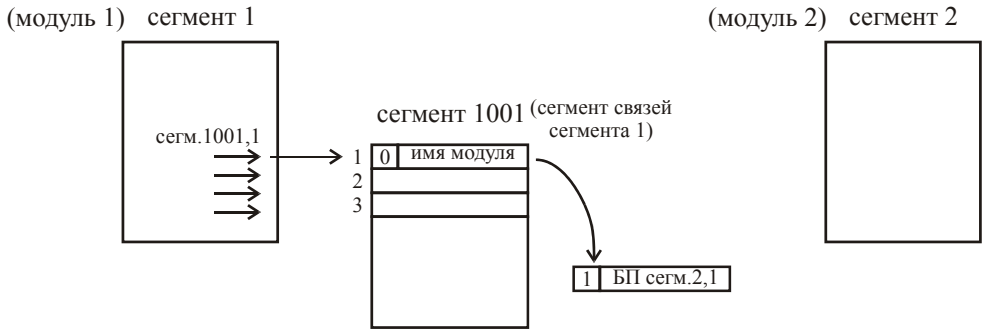
Аппаратная поддержка взаимодействия программных модулей

Аппаратная поддержка взаимодействия программных модулей — вот что это такое? Вот у вас имеется программный модуль. Естественно, что в программных модулях есть какие-то обращения к другим модулям. Если модулей тысячи и в каждом из них очень много обращений, то, вообще говоря, возьмём некий предшествующий период, вы редактируете внешние связи. Допустим, помните нашу сегментную организацию? Этот модуль находится в сегменте 1, этот модуль находится в сегменте 2:



Другие модули в своих сегментах. И всё это редактирование внешних связей будет произведено, и здесь будет написано: сегмент такой-то, адрес в сегменте и т.д. Значит, вы проведёте огромное количество редактирования внешних связей. Ясное дело, что если у вас тысяча модулей, то они все у вас будут сидеть на внешней памяти, когда он понадобится, вы его будете оттуда подкачивать, там уже указаны связи с другими модулями. Вы обратитесь, подкачаете следующий модуль, пусть это делает операционная система. Вроде как всё хорошо, всё нормально.

Вы отредактировали десятки тысяч внешних связей. Вы запустили вашу программу — никаких этих десятков тысяч не произошло, потому что такие входные данные. Проявилось второе глобальное свойство программы. Первое — это свойство локальности. Об этом мы с вами говорили, когда про кэши говорили, про таблицы адресной трансляции. Здесь второе свойство — программа зависит от исходных данных. Программа может работать 2 часа, и в ней сработают десятки тысяч связей, а может вылететь через несколько миллисекунд, и сработает одна-две связи. Спрашивается: зачем тогда мучаться? Естественно, что просто так ничего не даётся, поэтому делается это так. Вот берётся модуль и рядом помещается *сегмент связей* (пусть сегмент 1001 — сегмент связей сегмента 1). Здесь будет некое имя модуля, к которому будет обращение.

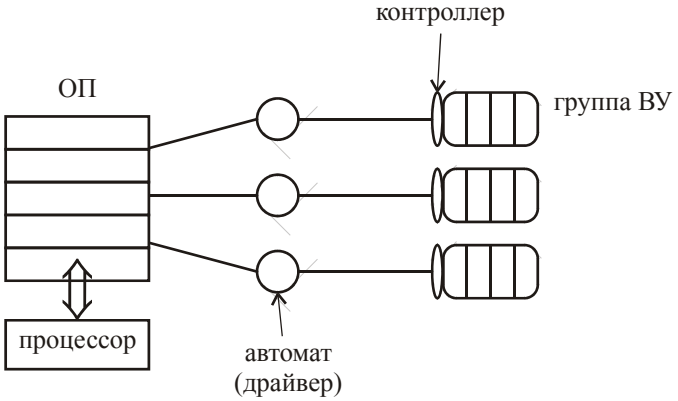


Усилиями транслятора и операционной системы созданы вот эти 2 сегмента: сегмент 1 вот с такими обращениями в сегмент 1001, и сегмент 1001 — сегмент связей модуля 1. Аппаратура учитывает специально диапазон сегментов с 1000. Запаяно в аппаратуру, что если сегмент начинается с 1000, то надо проверить старший разряд. Например, в нём «0». О, работать надо: возникает прерывание — с этим сегментом (с этим модулем) не было связи ещё,

поскольку там «0». Всё, прервались. Обратились к операционной системе, операционная система выкачивает сегмент 2, создаёт ему, естественно, сегмент связей (1002), и заменяет переход безусловный (БП) на точку входа, а здесь ставит «1». Если в следующий раз произойдёт обращение, то тут «1» — прекрасно, переход сюда (по БП). А сюда не произошло обращение — ничего и не трогалось. А понадобится — всё точно так же будет. Т.е. за счёт некоторого усложнения (а именно сегментов связей для каждого смыслового сегмента, и создание соответствующей аппаратуры ощупывания разряда той ячейки сегмента связи, на который происходит передача управления из смыслового сегмента), то значит здесь произошло знакомство (аппаратная поддержка взаимодействий). Вот так. Т.е. конечно, если у вас действительно будет происходить 10000 связей, то это у вас будет слишком много дополнительных (накладных) расходов. А если у вас есть разные варианты, то этой аппаратурой можно воспользоваться, если она в машине предусмотрена. Сейчас в некоторых есть, в некоторых нет. Вот такая вот ситуация. Всякая вещь несёт в себе всякий «плюс», вызывает ещё и какой-то «минус».

Типы устройств внешней памяти и ввода-вывода

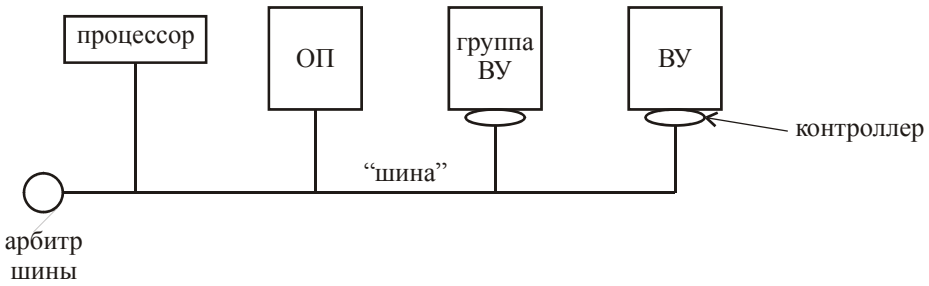
Конечно, нужно представлять себе, как работает терминал, какие-то такие моменты нужно себе представлять. Интересное понятие старт-стопности работы устройств. Довольно небольшой будет обзорчик этого дела. В основном-то вы всё это знаёте. Ну а затем, практически есть 2 подхода: один подход фирмы IBM, которая на длительное время предложила, осуществила, и многие это используют — это идеология каналов (селекторных и мультиплексных), где управление ведётся централизованно.



Процессор работает с ОП туда-сюда (принимает команды, операнды, записывает результаты — выполняет решение задачи), и независимо от этого с внешними устройствами (ВУ), как-то объединёнными в группы, осуществляется обмен данными параллельно друг другу и параллельно с выполнением программы на процессоре. За счёт распараллеливания памяти всё это происходит. Так вот, если здесь находится некоторое устройство, которое является *автоматом* управления обмена с этими устройствами, естественно, под управлением данных, которые доступны этому автомату (которые подготовила ОС). Операционная система (драйвер) подготовит информацию для этого автомата, который дальше будет выполнять автоматически загрузку, централизовано управляя работой либо одного выбранного устройства (*селекторный канал*), либо по небольшой порции информации группы устройств (*мультиплексный канал*). Именно этими терминами (селекторный и мультиплексный канал) именуются эти автоматы. Конечно, сразу возникает некий протест против названия. Позвольте, канал-то — вот он. Он идёт туда, по семи комнатам — это есть канал. А это что такое? Это какая-то промежуточная вещь, которая управляет созданием, т.е. *интерфейс*. Вот в IBM всё наоборот: это называется каналом, а то, что протянуто на 30 метров, называется интерфейсом! Что хочешь с ними, то и делай.

Ладно, Бог с ними. Это вот подход фирмы IBM, где у нас централизованное управление.

Теперь другой подход — фирмы DEC. Второй подход такой. Вот есть процессор, вот оперативная память, вот группа внешних устройств (ВУ), или одиночное внешнее устройство (ВУ). Здесь всё не так. Здесь имеется некая общая «*шина*», где все имеют доступ к этой общей шине.



Кто-то захватывает эту шину (устройство в этом случае называется master), а тот, кому оно передаёт информацию, называется slave (т.е. «слуга»). Ну а если все вместе хотят, то есть *арбитр шины*, который как-то осуществляет — сначала одной паре даёт, потом другой. Так что здесь передача идёт вот по этой «шине», и информация такая же, как в устройстве автоматов, здесь попадает в каждый из контроллеров внешних устройств. Там они этой информации не имеют, т.к. централизованное управление, а здесь — децентрализованное управление передачи данных, где каждый рвётся использовать «шину» по той информации, которая у него находится, подготовленная, естественно, операционной системой. Может быть несколько этих «шин», это не имеет значения. Но это мы рассмотрим позже. К этому ещё раз вернёмся.

Я вот тут снова обращаю ваше внимание (снова на http://parallel.ru), всего лишь несколько дней назад (14 ноября 2006 года) в новостях сообщается, была опубликована 28-я редакция списка Top500 (мощных компьютеров мира). Я начинаю с нижней грани:

«…Последняя, 500-ая система в новой редакции списка была бы полгода назад на 359-м месте, а для того чтобы попасть в текущий список, потребовалась производительность на Linpack (не пиковая, а реальная производительность на задачах линейной алгебры) 2.737 Tflop/s против 2.026 Tflop/s в июне (от июня до ноября вот какой произошёл колоссальный скачёк). Суммарная производительность систем в списке выросла за полгода с 2.79 Pflop/s до 3.54 Pflop/s.

…С 70-го на 99-е место в новой редакции списка опустился российский суперкомпьютер MVS-15000BM производства IBM, установленный в Межведомственном суперкомпьютерном центре РАН, с производительностью на тесте Linpack 6.646 Tflop/s.»

Буквально на 30 мест слетел вниз за счёт появившихся всё новых более мощных. Он не уменьшил свою мощность, а даже несколько увеличил, но получилась вот такая интересная история. Т.е. очень интенсивно сейчас появляются машины всё новые.

«…На первом месте в 28-й редакции списка остался прототип будущего суперкомпьютера IBM BlueGene/L с производительностью на Linpack 280.6 Tflop/s.

На второе место в новом списке поднялся суперкомпьютер Cray Red Storm, ставший только второй системой, преодолевшей рубеж производительности 100 Tflop/s - 101.4 Tflop/s на тесте Linpack. В предыдущем списке Red Storm занимал только 9-е место.

На 5-м месте в списке оказался новый самый мощный суперкомпьютер Европы IBM JS21, установленный в Barcelona Supercomputing Center, с производительностью на Linpack 62.63 Tflop/s.

…Также подрос в производительности самый мощный суперкомпьютер Японии производства NEC и Sun, установленный в Tokyo Institute of Technology, - с 38.18 Tflop/s до 47.38, но и он при этом опустился в списке с 7-го на 9-е место.

…Суперкомпьютер Earth Simulator, долгое время возглавлявший список, сейчас находится уже на 14-м месте.

…Количество кластерных систем в списке примерно стабилизировалось и составляет 361 против 365 систем в июне. Из коммуникационных технологий наиболее популярными остаются Gigabit Ethernet - 211 систем (в прошлом списке - 255), Myrinet - 79 систем (в прошлом списке - 87) и InfiniBand - 78 систем (в прошлом списке - 36).

Количество систем в списке, построенных на процессорах Intel, уменьшилось с 301 до 261. Процессоры AMD Opteron становятся все более популярны, на них построено уже 113 систем (в июне - 81). Процессоры IBM Power используются в 93 системах. Всё больше в списке становится систем, по-

строенных на двухъядерных процессорах - 75 систем на двухъядерных Opteron и 31 система на двухъядерных Woodcrest.

По количеству установленных систем, вошедших в список, IBM (239) продолжает доминировать, на втором месте по-прежнему Hewlett-Packard (156).

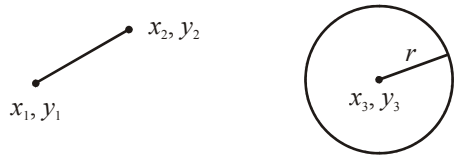
…По географической принадлежности продолжает доминировать США - 306 систем, Европа укрепляет свои позиции - 95 систем против 83 полгода назад, а Азия несколько уменьшает своё присутствие - с 93 до 79 систем.

…На 407-м месте в списке вновь появился суперкомпьютер HP SuperDome, установленный в Сбербанке, с производительностью на Linpack 3.059 Tflop/s…»

Ладно, приступим к двум темам, которые нам остались. У нас с вами рассмотрение подключения внешних устройств с точки зрения параллелизма работы связей с этими устройствами, и как это организовывается по двум технологиям (или парадигмам): парадигма (технология) IBM'а и парадигма (технология) DEC.

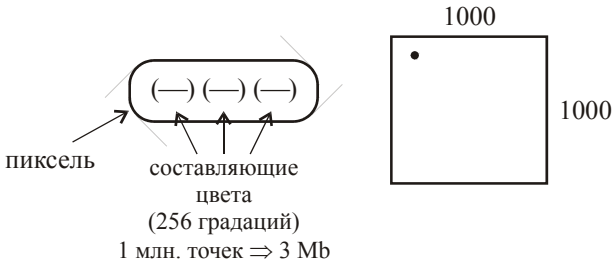
Дисплеи

Конечно, мы должны представлять, что такое собой представляют современные устройства. Например, устройство отображения — *дисплей*. Исторически довольно было преимущественным образом, дисплеи — векторные. Потом они постепенно вытиснились дисплеями растровыми или дисплеями телевизионного типа. Дисплей векторный — это означает, что вы где-то в памяти указываете (ну список передайте в некую память на устройстве управления дисплея) координаты начала и конца (x_1, y_1, x_2, y_2). Если у вас круг, то вы указываете центр (x_3, y_3) и радиус (r). Если у вас *АБВ*… (последовательность буквенно-цифровая), значит, вы указываете x_4, y_4 (с какого места начинается) и адрес этой последовательности в памяти. Как видите, для такой картинки информации надо не много, она помещалась в память специального устройства — контроллера, но зато приходилось иметь специальные схемы: специальные схемы прочерчивания прямой (называлось *генератор векторов*), *генератор окружностей*, *генератор буквенно-цифровой информации* всех символов, т.е. уже сама аппаратура помещала соответствующие точки, а потом ещё, ещё и ещё раз, чтобы вы видели комфортно на экране. Но самой информации для этого в памяти — всего ничего.



Но когда таких элементов (векторов, окружностей, символов) становилось очень много, то уже работа «всемогущей электроники» стала замедляться, просто потому что очень много приходилось выполнять всяких действий, а потом это повторять сколько-то раз в секунду, чтобы было комфортное ощущение. И вот, когда таких элементов стали тысячи на картинке (больше — десятки тысяч), векторные дисплеи уступили полностью место растровым.

Мы с вами говорили о создании для машины БЭСМ-1 в своё время в 1958-м году практически первого дисплея растрово-телевизионного типа. Это очень просто: у вас где-то в памяти (называемый *видеобуфер*) имеется информация о том, как засветить конкретную точку на экране. Допустим 1000 на 1000 точек (как правило, бывает поменьше, но скажем так). Если просто светло-темно, то достаточно одного бита (что и было для самого первого дисплея, где видеобуфер был на магнитном барабане). Если же вы хотите иметь цветовое решение в этой точке да ещё с интенсивностью, то вам надо указать составляющие цвета (синий, красный, зелёный) и указать ещё градацию этого цвета (например, 256 градаций). Такие есть системы. Зато у вас уже байт на указание градации одного составляющего цвета, байт на указание второго, т.е. вам 3 байта нужно на указание как засветить эту точку. Это и есть, так называемый, *пиксель*, т.е. информация о точке. Таким образом, если у вас здесь будет миллион точек, то вам нужно 3 Мб для этого буфера.

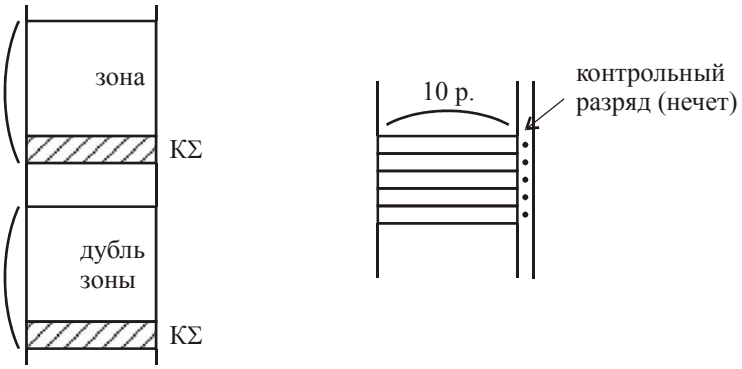


Естественно, мы должны в памяти это всё подготовить, а затем уже просто автоматом происходит. Тут уже вам генераторов не нужно, вам нужно просто луч направить. Вот это дисплеи.

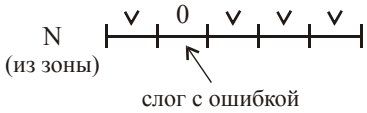
Диски

Что касается устройства дисков, то вы понимаете, что на диске у вас имеется какая-то на дорожке информация и, конечно, есть какие-то служебные данные. Может быть много информации: информация, контролирующая то, что там находится. Фактически, когда вы считываете, а когда вы записывали, то вы туда поместили информацию, контрольные суммы (например, по сложению всех слов или байт — уникальная вещь, или ещё каким-то способом контроля) здесь это помещается. Ну и служебные данные могут включать другие данные о том, кто хозяин этого дела, когда это было и разную прочую информацию об этой информации — *метаданные*.

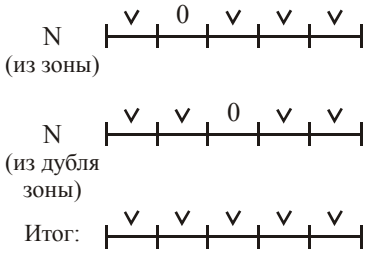
Это, конечно, так, в своё время самые разные решения здесь были, на магнитной ленте тем более контроль очень существенный. Я вот могу только привести очень интересный пример, как в своё время из очень плохой ленты (а ленты были всегда плохие, но не сами ленты, а их покрытия вместе с их прилеганием к головкам) — ленты были очень ненадёжным устройством, считывание происходило долго. Вот записывался некий массив данных (скажем, страница памяти БЭСМ — 1 килослово = 6 кБайт), записывалось сюда, и делалась зона и дубли зоны. Что собой представляла зона? Зона представляла следующее: вот слово 50 разрядов (48 — информационных, 2 — контрольных) разбивалось на 5 слогов — 6 слогов по 10 разрядов. Каждому слогу приписывался один контрольный разряд (т.е. нечет), при записи он сюда помещался.



Как это происходило? Вы считывали зону, при этом каждый слог аппаратно контролировался в связи с этим битом контроля. Контролировалась правильность считывания с той возможностью, которую предоставляет этот контроль. Т.е. если у вас появлялась одна ошибка, что означает — нарушилось на самой ленте или неправильно считалось при считывании, то возникал сигнал ошибки. Если при считывании возникал сигнал ошибки, то тогда делалось следующее. Да, во-первых, при считывании в память слова целиком, информация поступал следующим образом: если слог прочитывался, и аппаратура определяла, что правильно прочиталось (если 2 пропало или 2 появилось — это не ловится, дальше поймаем), тогда слог этот записывался. Если какой-то слог определился, что он не верный, то записывался сюда нулями (это называется блокировкой записи неверных слогов). Вот в некую ячейку памяти N появилась такая информация:

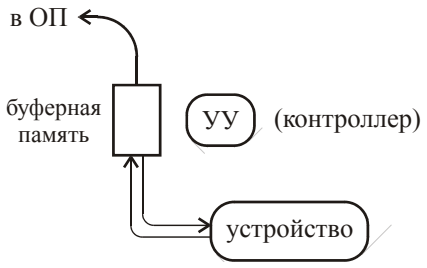


Но поскольку был общий сигнал ошибки (если хотя бы один слог одного слова неверен — появился общий сигнал ошибки), тогда считывался дубль зоны, но считывался он в эту же страницу памяти на эту же ячейку оперативной памяти тоже с блокировкой записи неверных слогов и с наложением. Вот теперь давайте посмотрим, как N из дубля зоны:

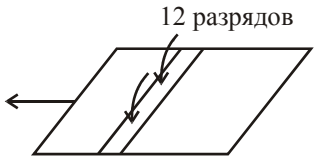


Предположим, что в этот раз на дубле зоны этот правильно прочитался, а не верно прочитался, скажем, следующий. Тогда что произойдёт? Этот (будем считать верный) наложится на верный, этот верный наложится на нули — мы тоже получим верный, а этот нулями наложится на верный — получится верный. И таким образом из двух одноимённых слов, где разные слоги неверные (это самый плохой случай), получится один верный. Но естественно, что при этом, когда всё это произойдёт, после этого надо проверить *контрольную сумму* (КΣ), а она, конечно, здесь, как и в случае диска, в специально отведённой зоне, где тоже есть сведения об имени ленты или информации на ленте, о времени работы и т.д. Так что эта контрольная сумма поступает в специальные ячейки памяти, как и в случае диска, и тогда всё вместе производится контрольное суммирование и сравнение на предмет, а вдруг двойные ошибки были. Если всё совпало, значит всё в порядке, можно работать. Если нет, разворачиваем и на натуральный накладываем ещё раз, не получилось — натурально считываем, накладываем, накладываем, пока не получится. И опять ничего не получается. Как вы думаете, что надо делать? Подходите, открываете шкафчик, где магнитофон, и пальчиком оттягиваете эту ленту — эть! — проскочила, и так и стоите. Не получается — снимаете ленту с этого магнитофона и ставите на другой. Естественно, операционная система обнаруживает, что с одного сняли, на другой поставили, и продолжает эту работу. Вот это интересно, как из двух массивов плохой информации можно получить один с хорошей информацией. И разные прочие интересные ухищрения были, во многом существуют сейчас. Сейчас уже более мощная аппаратура имеется для повышения надёжности работы устройств внешней памяти.

Что можно сказать? Про устройства можно сказать только последнее. *Старт-стопные* и *не старт-стопные* устройства. Что это такое? А это вот что такое. Если у вас имеется устройство и имеется устройство управления (УУ) — контроллер. А что такое контроллер? Это тот, который контролирует. Такая функция у него есть, но это не единственная функция (контролирования). От слова «control» — «управление». Вот если у него имеется внутри некий буфер (некоторая память), которой достаточно для того, чтобы запустив работу устройства из этой памяти при его непрерывной работе, которую уже нельзя по какой-то причине остановить, информация бы вся передавалась, как только мы запускаем некий акт работы с устройством, и то же самое в обратную сторону, если это устройство для считывания.



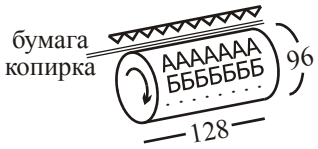
Ну, например, вот у вас есть перфокарта — её подали и она летит мимо светодиодов, и её уже не остановить. Поэтому снять с неё информацию можно автоматически, поместив её вот в такой промежуточный буфер. В этом случае там 80 колонок по 12 разрядов — всего ничего. Если вы из этого буфера не успели взять информацию, то ножик следующую перфокарту не толкнёт. А когда оттуда заберёте информацию уже в оперативную память (ОП), то уже следующую карту подаст. Тогда это устройство называется *старт-стопное*. А не старт-стопное, это когда вот эта перфокарта движется под фотодиодами, и вот пришли 12 разрядов, и вам нужно сразу их записать в память, потому что через какое-то время (через 200 мкс, может через 300 мкс) уже придёт новая информация. Естественно, она поступает в некий регистр, но из него надо забирать. 200-300 мкс на нём будет сидеть ещё вот это, а потом затрутся. Поэтому надо успевать забирать, забирать, забирать. Вот такие устройства называются *не старт-стопными* устройствами.



Печать

Кстати, такое же не старт-стопное устройство были раньше печать (ЦПУ). Очень интересно. Существовал барабан с печатью (сейчас нигде не найдёшь показать), и на нём выгравированы русские, латинские, цифры — полный алгольный набор — 96 символов, 96 образующих. Вот он крутится. А над этим находятся молоточки, а тут бумага и копирка.

Есть какая-то реперная точка. И когда это поворачивается, подходит под эти молоточки вот эта образующая цилиндра с выгравированными «А», вы должны иметь готовыми 128 таких вот нуликов и единичек. Это подавалось на 8 регистров по 16, и тогда подачу, имеющуюся готовую к этому моменту, это означает, где единички, там пробьётся «А». Проходит 1500 мкс и снова «караул» — надо бить «Б». А не успеете на 100 мкс — ничего. На 600 мкс — будет половина от одного символа, половина от другого. А опоздаете на все 1500 мкс — у вас вместо «2» появится «3». Так что это вот не старт-стопное.

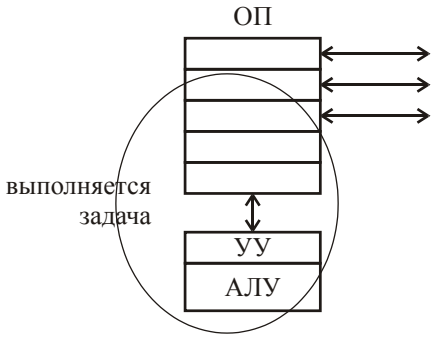


Так что мы должны иметь готовыми 96 × 128 бит, которые составляют какую-то матрицу. И это делала операционная система. На уровне вот этой матрицы было доступно, а внутри уже недоступно — надо было всё успевать. И это делала операционная система, обслуживая вот такие старт-стопные устройства на БЭСМ-6 по каждой такой единице информации, пробивая при подаче прерывания каждый раз один символ. Немного вперёд забегая, могу лишь сказать, что на БЭСМ-6 мультиплексного канала не было (о нём мы ещё с вами поговорим), в качестве мультиплексного канала работала операционная система. На БЭСМ-6, чтобы обслуживать печать, 15% времени на это уходило. В среднем же тратилось на операционную систему не только на это, но и на все действия по управлению задачами, использование ресурсов, — не больше, чем 5-7%. Ничего в этом страшного нет, можно так поступать. Неудача не старт-стопных в том, что ничего нового не подсоединишь — это серьёзная проблема. Потом появились устройства управления для печати, и уже эта штука превращается в старт-стопную. Тогда никаких временных проблем у вас вообще не возникает.

Кстати, на этом деле очень хитрый мужичок по фамилии Ганголия — индус, вот что значит, совершенно ничего не понимает в вещах, связанных с маркетингом. Машину БЭСМ-6 поставили в Индию. Была конкуренция: то ли ставить американскую машину, то ли машину БЭСМ-6, и выиграла машина БЭСМ-6, но я думаю не только по своим качествам высокой

производительности, достаточной надёжности и т.д. (там приблизительно одинаково было). А он сразу понял, что нужно делать. Он приехал в Дубну (поскольку ядерный центр), приехал, посмотрел. «А сколько времени тратится процессором на обслуживание одной печати?» Ему всё рассказали — 15%. «Ой, как это нехорошо… Это очень много… У нас много печати будет! Давайте мы поставим сотивитную машину. У вас же есть машины, так называемые, фортранные станции ввода-вывода?.. Давайте мы купим машину БЭСМ-6, но не одну, а с машиной ТРА венгерского производства». Откуда он знал, что такое маркетинг? А контракт был составлен так: Советский Союз поставляет машину, как только машина поставляется, и она запускается и начинает там работать, за неё сразу платят. Пока купят там машины ТРА, Советский Союз бесплатно поставляет обслуживание, запасные части и т.д. Машину поставили, машина хорошо заработала, все довольны, деньги за машину получили, а венгры не сделали: «Мы сейчас не успели, мы сейчас не сделали, но в следующий раз мы сделаем в 2 раза больше!» Ну да, 2 раза нуль — чего будет? Короче говоря, они на этом здорово выиграли. Года 2 их бесплатно обслуживали, пока венгры всё это делали. Видимо соображал с самого начала, что обязательно будет задержка. Ничего страшного там с печатью не случилось, не так часто они там печатали, вовсе не по этому. Вот такая интересная история была, очень интересная была картина всего этого дела.

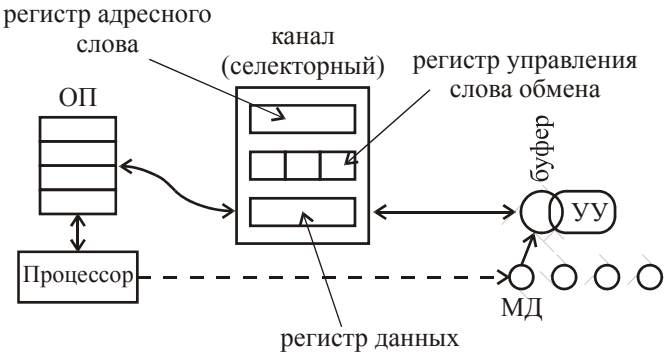
Вот вам нужно в оперативную память помещать данные параллельно с работой АЛУ. И желательно по нескольким направлениям обеспечивать обмен параллельно с выполнением задачи.



Значит, как-то внешние устройства должны уметь подключаться и обслуживаться параллельно. И вот было предложено 2 идеологии (стратегии, парадигмы, как любят выражаться) — это парадигма IBM, так называемая, каналов, и парадигма DEC — общей шины. Вот это общая идея.

Нарисуем здесь подход каналов.

Итак, оперативная память. Дальше имеется связь с неким устройством канала (так оно и называется, хотя на самом деле, это у нас канал), и дальше идут внешние устройства. Имеется, естественно, устройство управления (контроллер) и само устройство (одно или несколько). Например, стандартный контроллер управления звеньями несколькими контрольными сериями ЭВМ, выпускался ещё дважды, было 4 таких устройств. Оно даже очень интересно называлось «УГУ» — устройство группового управления.



Что собой представляет устройство каналов и как оно управляет прохождением данных? Оно управляет этими данными, т.е. централизованное управление со стороны устройства каналов. В устройстве каналов мы выделили с вами 3 регистра. *Регистр данных* — данные берутся из буфера устройства управления, туда и обратно. Фактически передача данных идёт «память-память». При этом если так называемый *селекторный канал*, то тогда у вас передача данных идёт не прерываясь, т.е. как данные поступают в этот буфер, так они сюда и транслируются непрерывно. Это означает, что видимо эти данные идут достаточно быстро, чтобы здесь не переходить временно к управлению устройства передачи данных от него. Да, такие устройства есть — это магнитные диски (быстрая память на магнитных дисках). Обмен идёт непрерывно. Процессор передаёт контроллеру некоторые указания о запуске обменов с магнитным диском, и обмен начинается. Что значит «процессор»? Естественно, всё это делает драйвер. Драйвер передаёт указание на устройство пуск обмена, предварительно подготавливая информацию для этого обмена. Что это за информация? Он подготавливает *управляющее слово обмена*. Что здесь указано: адрес по оперативной памяти (ОП), естественно, начальный, потом текущий будет, количество передаваемой информации или адрес конца, операция (считывание, запись).

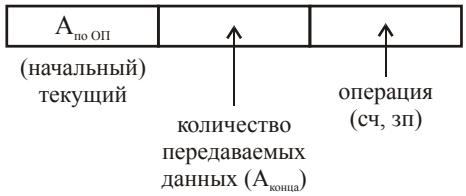


Рис. 15 Управляющее слово обмена

И это управляющее слово должно попасть на *регистр управляющего слова*. Если это всего одно управляющее слово, то тогда его можно просто поместить на *регистр управляющего слова* селекторного канала, и можно делать пуск обмена. Под управлением этого управляющего слова информация поступает в память или берётся из памяти и проверяется на количество переданных (когда заканчивается передача), считывание, запись, всё, как видите, управляется с этого «центрального поста», этим центральным автоматом.

Вот в некоторых книжках пишется «специализированная вычислительная машина» — это плохо, никакая это не вычислительная машина, это просто некий автомат, программируемый драйвером (в данном случае, драйвером магнитного диска). Вот так происходит.

Теперь *регистр адресного слова*. Когда это нужно? А нужно это, когда вот этих управляющих слов (см. Рис. 15) готовится несколько, как бы на несколько обменов, и всю эту в данном случае программу работы этого устройства (цели), или *программу подканала*.

Допустим, эта программа начинается с некоего адреса *N*, и этот адрес *N* и помещается сюда (в регистр адресного слова).

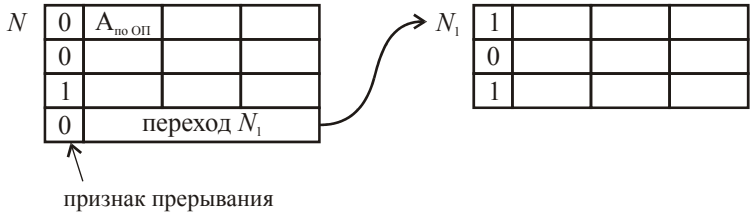


Рис. 16 Программа подканала

Как только закончится обмен под управлением первого управляющего слова, к *N* прибавляется 1, и следующее управляющее слово берётся вот отсюда. В принципе мы можем иметь здесь некую последовательность, какую-то команду перехода на другую последовательность, где тоже пойдут управляющие слова.

Т.е. это адресное слово, которое указывает, из какого адреса памяти берётся текущее управляющее слово и программа подканала. Естественно, что всю эту программу готовят драйверы. Но там не было программных каналов памяти. Там драйвер готовил управляющее

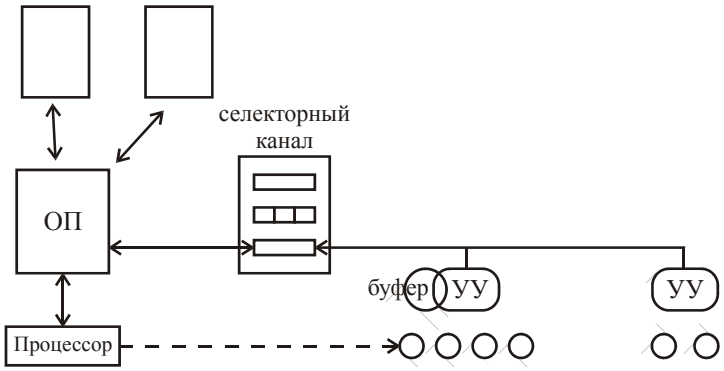
слово и помещал на некий регистр, под управлением которого происходила вот эта вот непрерывная работа, пока весь обмен не закончится с конкретным устройством.

Здесь могли вот таки вещи интересные делаться: вводят *признак прерывания* (см. Рис. 16). Здесь нет, здесь нет, здесь почему-либо есть, здесь нет. Т.е. как только используется обмен, у которого нет признака прерывания в управляющем слове, сразу произошёл обмен, чтобы принять это управляющее слово, и прерывание процессора не производится. А вот как только кончился этот — происходит прерывание процессора, и он узнаёт, что уже там какие-то нужные вещи можно будет делать. Ну, я говорю, конечно, всё о правильной работе. Если неправильная, то это сигнализируется прерыванием, там другие вопросы.

Вот такая вот автоматная работа селекторного канала под управлением управляющих слов обмена. Т.е. обмен ведётся под управлением центрального устройства, которое программируется на эту работу драйвером устройства. У другого устройства имеется своя программа подканала и т.д.

Вообще говоря, пока ведётся здесь обмен, то за это время можно подготовить для других таких устройств.

Я нарисовал вот так, а может быть ситуация вот какая (это не нарушает нашего обсуждения работы селекторного канала): вот здесь как-то так идёт сюда магистраль (кстати, она в IBM'е называется «интерфейсом», а это устройство — «каналом»; я бы назвал наоборот — это было бы естественнее физически).



Таких селекторных каналов может быть несколько, соответственно несколько и таких устройств канала.

Теперь *мультиплексный канал*. Делалось это так. Как правило, селекторных каналов действительно несколько, а мультиплексный один.

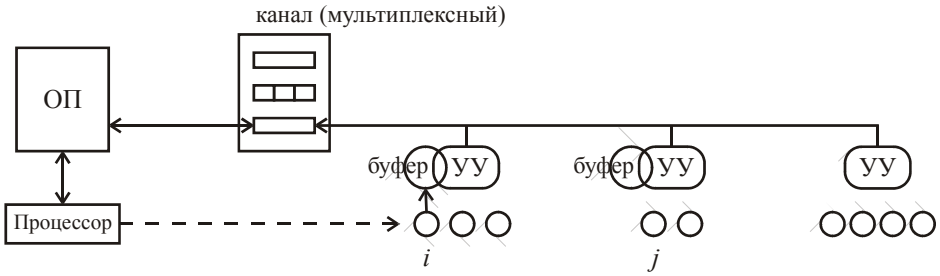


Рис. 17

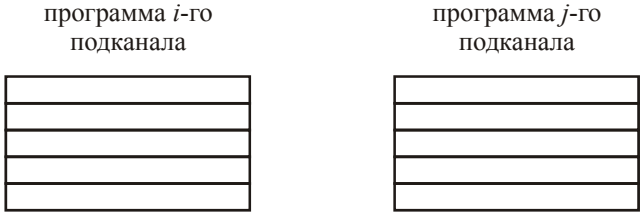
Как будет работать этот канал? Принцип работы мультиплексного канала: вначале готовит драйвер информацию, потом командует пуск устройству, и так одного за одним, и все эти устройства начинают работать на этом канале (подключённые к этому каналу) одновременно, помещая данные или наоборот, забирая данные, которые поступают. А мультиплексный канал (вот этот автомат) делает следующее: как только есть готовые с какого-то из этих (я беру устройства другие, не магнитные диски, более медленные, типа терминалов, печатных, ещё чего-то). У вас имеется сигнал готовности *i*-того устройства и *j*-того (Драйверы *i*-того и *j*-того устройства подготовили информацию, драйверы *i*-того и *j*-того устройства подали команду пуска обмена с *i*-м и *j*-м устройством, и вот контроллеры пищат, сообщают каналу: «я *i*-й готов» и «я *j*-й готов». Это означает, что если *i*-й идёт на пе-

редачу в память, значит, в буфере появились данные. Если какой-то канал готов на выдачу, значит, буфер у него свободен для приёма данных.

Что делает мультиплексный канал? Он по сигналу готовности *i*-того устройства управления (УУ) запускает обмен на какую-то порцию данных между памятью и буфером. После того, как это обмен порцией выполнится — «кто ещё пищал? — *j* пищал». Теперь осуществляется обмен с *j*. Даже если они пищали одновременно по какому-то приоритету сначала берётся обмен некоторой порцией данных с устройством *i*, затем берётся обмен некоторой порцией данных с устройством *j*. Ну а затем с устройством *k*, а если только *i* и *j* пищали, значит, потом снова некоторой порцией с *i*, потом с *j*, и так по очереди, почему и называется «мультиплексный» — мультиплексирует вот эти передаваемые данные.

Какие это порции? Это может быть даже 1 байт и даже есть такой термин «байт-мультиплексный канал»: вы один байт передали отсюда сюда, следующий байт сюда, следующий — сюда, и вот так вот по очереди. Есть «блок-мультиплексный канал», когда у вас передаётся какое-то количество байт (8, 16, не обязательно степень двойки).

Как работает в этом случае устройство управления каналом? Для каждого устройства (в частности для *i*-того, для *j*-того, для других) готовится вот эта самая программа подканала.



Программа *i*-того подканала. Где-то рядом в памяти (так же, как и в том случае) программа *j*-го подканала. Итак, должна быть программа, осуществляющая полностью обмен с *i*-м, а потом полностью обмен с *j*-м.

Вы прекрасно помните, что здесь адрес текущий, объём передаваемых данных и операция (запись/считывание). Как всё происходит? А происходит очень просто. Мы с вами говорили, что пищат *i*-й и *j*-й. Существует *таблица адресных слов подканалов* или устройств. Сколько у вас устройств? Допустим, у вас 256 устройств, значит, будет 256. Где-то здесь есть *i*, где-то здесь есть *j*:

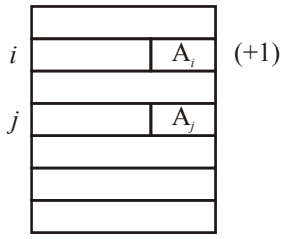


Рис. 18 Таблица адресных слов подканалов

Соответственно, это есть ничто иное, как адрес управляющего слова. Вот он — A_i. А здесь — A_j. В самом начале A_i — это адрес начала группы управляемых слов для *i*-го канала, а A_j — для *j*-го. Итак, пищит *i*. Отсюда (из таблицы адресных слов подканалов) вот это, вы сами понимаете, нет проблем найти.

Цикл работы мультиплексного канала:

1) считывание адресного слова (A_i);
2) считывание (по A_i) текущего управляющего слова;
3) обмен порцией (блоком) данных;
4) изменение управляющего слова;
5) запись изменённого управляющего слова.

На этом всё. Если дальше начинаем обслуживать передачу данных *j*-го подканала, снова начинает работать цикл, только уже для *j*.

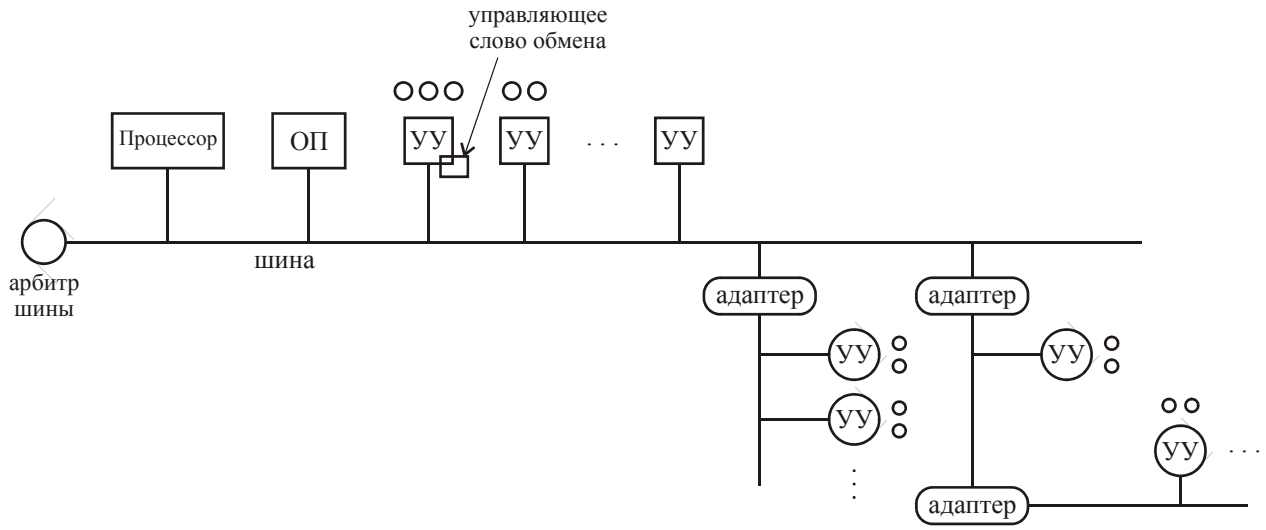
Когда меняется адресное слово? Очень просто, когда здесь нуль и закончился обмен под управлением управляющего слова. Тогда сюда прибавляется единичка, и уже этот записываете сюда.

Здесь происходит 3 обращения в память (на 1-м, 2-м и 5-м шагах цикла). Ничего не поделаешь. Это делается для того, чтобы можно было запустить параллельно много обменов. Работает много задач или даже одна задача может работать с многими устройствами, а это дело благое, чтоб параллельно работали устройства, ведь они работают, потому что задаче это надо, значит нужно по возможности это всё здесь быстро как-то распараллелить. Псевдопараллельность, конечно, в этом случае, как вы понимаете. Но реально устройства работают параллельно.

Вот такой подход к *централизованной* организации обмена данными с внешними устройствами за счёт устройства каналов, и этот автомат работает за счёт информации, подготавливаемой драйвером.

Второй вариант, когда эти же управляющие слова (естественно, никуда от них не деться) тоже готовятся драйверами, но передают эти управляющие слова (а, может быть, даже и программы управляющих слов) в память, которая входит в устройство управления (УУ) устройства (т.е. контроллер). Делается это в варианте *шинной архитектуры DEC*.

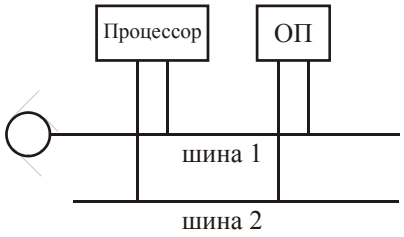
У вас имеется процессор (или несколько), оперативная память (ОП) и вот эти устройства управления (УУ), за которыми находятся устройства. Имеется некая шина, к которой подключены вот эти все устройства. И по этой шине под управлением управляющего слова, которое находится в самом контроллере, после или постоянно, захватывая на передачу каждого объёма данных.



Устройство управления (УУ) под управлением управляющим словом захватывает эту шину, передавая данные в оперативную память (ОП) или забирая эти данные. Естественно, драйвер в своё время управляющее слово передал.

Всегда кто-то является инициатором запроса. За шину борются все компоненты, которые я нарисовал, в своё время являющиеся либо хозяином, либо слугой (master/slave). Естественно, они все будут бороться одновременно, поэтому имеется *арбитр шины* — устройство, которое эти конфликты снимает (пропускает сначала один, потом другой и т.д.). Т.е. всё то же самое, как мы с вами рассматривали, только уже контроллер устройства здесь он уже является либо контроллером устройства, которое ведёт обмен непрерывно, захватывая эту шину периодически, либо здесь какие-то другие есть устройства, они тоже захватываются так. Вот такой вариант. Здесь подсоединяется достаточно небольшое количество устройств управления (контроллеров) — достаточно быстрые устройства, а сюда подключается некий *адаптер*, через который подключается другая шина, к которой подключаются тоже устройства управления с другими устройствами (ввода/вывода, например). И так может быть некоторое количество. Есть шина первого ранга, есть шина второго ранга. И,

сами понимаете, здесь может быть и третьего уровня. Таких шин может быть и несколько, и даже может быть вот так сделано:



И памяти может быть несколько и процессоров — вопрос комплектации. Вот такой подход к управлению, который назвали по сравнению с тем подходом *децентрализованным*. Реально это предложила фирма DEC — Digital Equipment Corporation.

Один симбиоз мы с вами увидим, когда будем рассматривать многомашинные комплексы АС-6. Вот этот симбиоз — средний (промежуточный) вариант во многих машинах существует.

Вот собственно то, что мы хотели рассмотреть по разделу «Обеспечение обмена с внешними устройствами». Обращаю ваше внимание, что задача наша была показать возможность параллельной работы устройств, обмена с внешними устройствами и как этот обмен управляется.

Многомашинные комплексы

Ну что же, у нас осталась одна последняя тема — это рассмотрение многомашинных комплексов. Уж в который раз я говорю про то, что многомашинные комплексы возникли сразу, как только появились две машины, практически везде. Дальше они успешно существовали и продолжали существовать как комплексы *сосредоточенные*, т.е. когда вычислительные машины находятся относительно близко друг к другу и уже по этому признаку, конечно, достаточно быстродействующих каналов связи (из-за короткого расстояния быстродействие и определяется — на небольшом расстоянии при большой частоте передачи легче бороться с возможными искажениями передачи аппаратно), и второй класс — это *рассредоточенные* комплексы. Примеров таких комплексов, конечно, очень много. Вы знаете много проектов, которые вертятся действительно в огромном сообществе людей. Все всегда отмечают проект SETI (SETI@home: Search for Extraterrestrial Intelligence at Home) — это поиск внеземных цивилизаций, попытка расшифровывать сигналы, искать осмысленные сигналы, спектры разных излучений, идущих из разных мест вселенной. Т.е. каждый, включающийся со своей машины через средства массовой коммуникации (через Интернет), решает часть общей задачи.

Не так давно был активный эксперимент по решению биологической задачи. Она была решена при параллельном использовании сотен компьютеров. Организатором был наш отечественный центр МГУ при поддержке вычислительного центра (ВЦ) МГУ, машины Черноголовки, машины Урала, многие институты были подключены для решения этой задачи, и она была решена. Т.е. существовал многомашинный комплекс на базе коммуникаций сети Интернет.

Можно говорить о *сильносвязанных*, о *слабосвязанных* комплексах. Трудно решить такую задачу, но так можно определить, что сильносвязанный комплекс — это такой комплекс, где передача данных от одной машины к другой идёт приблизительно со скоростью работы оперативной памяти (ОП) этой машины, не на порядок отличается от этой скорости (скажем в 2 раза медленнее, в 3 раза медленнее, но не в десятки, не в сотни раз медленнее). Вот такие комплексы называются комплексами с сильной связью. Что это такое? Это комплексы с общей памятью (мы такой комплекс рассмотрим — комплекс АС-6, а потом отметим, что был создан такой комплекс в университете Сан-Диего в Америке уже после того, как у нас были разработаны эти комплексы АС-6, естественно, это произошло не потому, что они узнали, да это и не нужно было, по необходимости сути такой обработки информации, они такой ком-

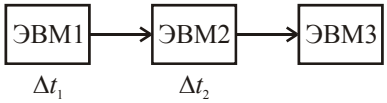
плекс построили, который оказался идеологической копией комплекса АС-6, но был разработан и эксплуатировался в существенно более позднее время), либо с общедоступной памятью, к которой обращение идёт со скоростью её работы или несколько медленнее. А второе — это высокоскоростные каналы. Тут зависит от того, какие это каналы. Если у вас гигабитные каналы (или многогигабитные каналы — самые современные достижения), то, смотря какие машины, для некоторых это и будет как раз то самое. Поэтому даже если гигабитный канал соединяет машины находящиеся в пределах здания, то это есть комплекс с сильной связью. Раз так, были комплексы со слабой связью, они делались во многих случаях с использованием внешней памяти: на одной машине что-то обрабатывалось, и это что-то записывалось на внешнюю память (на магнитный диск, магнитную ленту), и уже другая машина, имеющая доступ к устройству управления (об этом говорили), к контроллеру нескольких входов, оттуда эти данные получала для использования в счётной задаче каких-то там действий, объединённых этой системой. Ясное дело, что поскольку скорость работы внешней памяти существенно медленнее скорости ОП, поэтому это комплекс со слабой связью. Такой комплекс существовал: было две машины БЭСМ-6 и общее поле дисков, и с обоих машин селекторный канал подходил к контроллеру или устройству управления этим полем на магнитных дисках (большое количество дисководов), и вот так эти машины работали, принимая друг у друга информацию. Это был комплекс со слабой связью. Многие говорили, что комплекс со слабой связью всегда мог организовать любой пользователь тем, что он брал ленту в одной организации (прятал для соблюдения секретности) и нёс её в другую организацию, там ставил её на магнитофон, и пожалуйста вам, получал информацию или передавал кому-то, и там эта информация считывалась, использовалась при решении задачЯ ещё раз повторяю, что когда был семинар по связи машины БЭСМ-6 с другими ЭВМ в Киеве где-то в 70-х годах, выяснилось, что число комплексов, в которых участвует машина БЭСМ-6, превосходит количество этих самых машин. Это значит, что каждая машина использовалась даже в нескольких вариантах многомашинных комплексов.

Рассмотрим комплекс с общедоступной памятью (она не общая, конечно), который реализовывал конвейер. Когда мы рассматривали многопроцессорные комплексы, мы говорили, что одним из первых многопроцессорных комплексов был *макроконвейерный* комплекс, т.е. комплекс из многих процессоров, созданный в Киеве, когда были процессоры, выполнявшие вычисления, были процессоры, на которых шли управляющие части программного комплекса. Всё это между собой связывалось по передаче данных. И вот этот комплекс, обладавший свойством постепенной деградации — это когда у вас часть комплекса выходит из строя, то ухудшение параметров по времени, время удлиняется обработки информации, но она продолжает обрабатываться. Это называется *система*, *обеспечивающая постепенную деградацию*. Не очень приятный термин, но такие системы всегда считались хорошими. Хотя при очень высокой надёжности сейчас современные кластерные системы уже отошли от принципа постепенной деградации просто потому, что выход из строя — это ЧП, которое случается очень редко, значит, в этом случае будет невыполненная обработка повторена, конечно, долго придётся делать, но ничего страшного.

Итак, этот комплекс называется многомашинный комплекс АС-6. История начала этого комплекса совершенно не связана с многомашинностью. Была поставлена задача где-то в 67-м году, как только появилась машина БЭСМ-6, сделать для неё аппаратуру сопряжения (откуда и слово «АС») с телеграфными и телефонными каналами связи. Некоторое сопряжение с терминалами было с самого начала. Была связь, конечно, с телетайпами, потом были немецкие рулонные телетайпы (за рубежом ими часто пользовались). Такая связь была, но она не полностью удовлетворяла, и поэтому была поставлена задача, что приходящие из реально дальних пунктов измерения параметров полётов космических аппаратов, получение телеметрии — всё это должно идти где-то телеграфным, где-то телефонным каналом. И вот должно приниматься машиной БЭСМ-6 (наиболее мощной машиной того времени), и там должно оставаться 80% процессорного времени на решение самих задач, а 20% могло быть потеряно на поддержку этого всего операционной системой. Такая аппаратура была создана, но она в чистом виде не пригодилась, поскольку «Лунная программа» (как вы поняли, всё

делалось для «Лунной программы», поскольку в 67-м году) не осуществилась у нас в Советском Союзе.

И делая эту аппаратуру, постепенно стало ясно, что в тех организациях, где действительно ведётся многоплановая обработка информации, где существует система обработки информации в реальном времени, очень полезно иметь комплекс машин, которые бы обеспечивали несколько основных функций для систем обработки информации. Первое — это обеспечить скорость обработки информации, которая фактически складывается из скорости обработки отдельных компонент. Т.е. обеспечить конвейер, где какая-то ступень или часть обработки информации производится на одной машине, затем для последующего этапа обработки информации данные передаются по коммуникации на другую машину.



За это время (Δt_2) проводится как второй этап обработки первой порции информации, так за это время проводится первый этап обработки следующей порции информации. Ну и вот этот конвейер машин можно было обеспечить для ускорения обработки больших потоков данных. Это первая задача.

Вторая была задача такая. У машины имеются свои внешние устройства. При объединении в коллектив (в комплекс) желательно, чтобы эти устройства могли быть использованы для любой задачи (любое устройство любой машины чтоб могло быть использовано в интересах обработки информации в любой задаче на любой машине комплекса). Это очень важно. Таким образом, в случае чего, в явлениях постепенной деградации, устройства одной машины могли бы обслужить задачи, идущие на других машинах. И наоборот, чтобы все устройства всех машин могли быть использованы гротескно в одной задаче. В основном это, плюс разумное резервирование, которое в реале можно было осуществить.

Что же собой представлял сей комплекс? Имелось некоторое количество *решающих узлов*. Таким решающим узлом могла быть машина БЭСМ-6, и где-то ещё там внешние устройства (ВУ). Такая машина была одна или несколько. Были разработаны, так называемые, *центральные процессоры* (ЦП), которые использовались как машины, в том смысле, что каждый ЦП управлялся собственной операционной системой (процессор в этом случае становился машиной), но не имел своих внешних устройств, а иногда не имел собственной оперативной памяти (ОП).

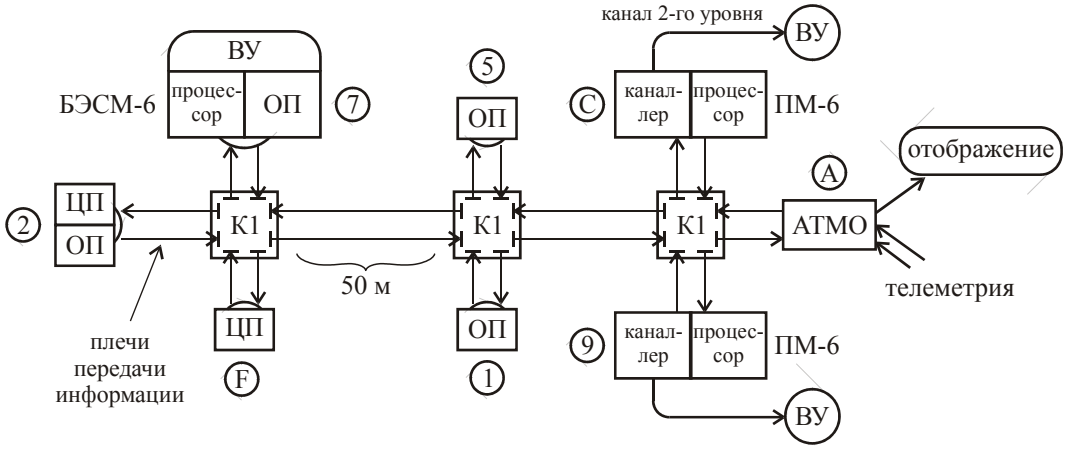


Рис. 19 Многомашинный комплекс АС-6

Центральный процессор (ЦП) отличался от машины БЭСМ-6 тем, чем и должен был отличаться, спустя определённый период времени, потому что система работала в 70-х годах уже полностью, машина БЭСМ-6 — в 60-х годах. У машины БЭСМ исторически (мы это с вами рассматривали, говорили причины этого дела) была очень небольшая виртуальная память. И в какой-то момент, тоже мы отмечали, уже физическая память превзошла виртуальную память для задачи. В данной машине был этот недостаток исправлен, и появилась очень большая виртуальная память, в ряде случаев существенно превосходившая физическую память — ситуация качнулась в обратную сторону. Ну и далее ещё были

мять — ситуация качнулась в обратную сторону. Ну и далее ещё были введены много новых операций для нечисловой обработки данных. Например, обычные операции нечисловой обработки — все операции алгебры логики (конъюнкция, дизъюнкция, сложение по модулю 2), т.е. какое-то количество операций, существенно поддерживающих решение логических задач. И любую обработку с использованием этих операций можно было делать. Ну не было операций над битами, над полями бит, над байтами, над полями байт, вот в центральных процессорах они были сделаны, какие-то задачи этими операциями воспользовались. И в то же время, где-то к концу разработки, замечательный инженер Андрей Андреевич Соколов сказал такую замечательную фразу: «Машина не должна превышать разумного уровня сложности». Она очень хорошо проверена была на машинах Эльбрус, потому как на машинах Эльбрус, что касается внутренней организации процессора, системы команд, работы операций с данными было взято за основу направление Burroughs с многими стеками, со всякими скрытыми стеками и т.д., т.е. с использованием безадресной системы команд. Было много молодых интересных программистов, математиков, инженеров, которые один из них что-то сделал — «Ах ты сделал вот это вот?! А я ещё вот так вот и вот так сделаю! Вот, знай наших!» И сделал. В результате, понимаете, в чём дело, машина не сразу пошла в строй. Ну там были и причины чисто технические, подвела элементная база чуть ли даже не в момент государственных испытаний. Но это тоже вот как-то сказалось, незримо, может быть. Должен быть разумный какой-то уровень сложности. Вот эта вот гениальная фраза, мне кажется, совершенно правильная.

И вот это всё объединялось неким адаптером, выходило на, так называемую, *сеть быстродействующего канала первого уровня*. Что собой представлял этот канал первого уровня? Вот я рисую коммутатор (К1) не потому, что это первый коммутатор, это просто коммутатор канала первого уровня. И к этому адаптеру подсоединяю по 2 регистра. Здесь организуется полный дуплекс: в тот момент, когда сюда передаются данные, отсюда данные могут передаваться сюда. Понятно, что это могут быть и данные из оперативной памяти, и с регистров центрального процессора. Этот коммутатор (К1) был на 4 входа/выхода. Для того, чтобы обеспечить высокую скорость, эти коммутаторы находились друг от друга на расстоянии не более 50 метров. Вот такой сосредоточенный комплекс с 50-ти метровым расстоянием между коммутаторами вполне нормальный, скажем один здесь, другой — в конце дома. Это вполне нормально. Реально эти комплексы были в соседних больших залах, или в залах вверху, внизу (в залах на соседних этажах или через один). И этого вполне хватало, чтобы скорость высокую поддерживать. А какая это скорость? Здесь передавались многоразрядные данные. Здесь были данные 96 разрядные. Что это такое? Это 12 байт. И вот таких передач на каждом *плече* здесь осуществлялось 2 миллиона. 24 Мб/с ≈ 200 Мбит/с — к тому времени это была очень рекордная скорость передачи данных. Вот откуда сейчас мы видим и сильная связь. Центральный процессор (F) мог иметь программу и данные, которые он использовал, в ОП (5), (1) или даже (2) (см. Рис. 19).

Каналом первого уровня считалось это всё вместе: коммутатор плюс наш русскоязычный канал (коммутаторы и плечи).

Я почему вот здесь говорю 200 Мбит/с, потому что потом, когда мы сравним с комплексом, который был в Сан-Диего через 10 лет, там была труба повыше и дым погуще, там было 800 Мбит/с.

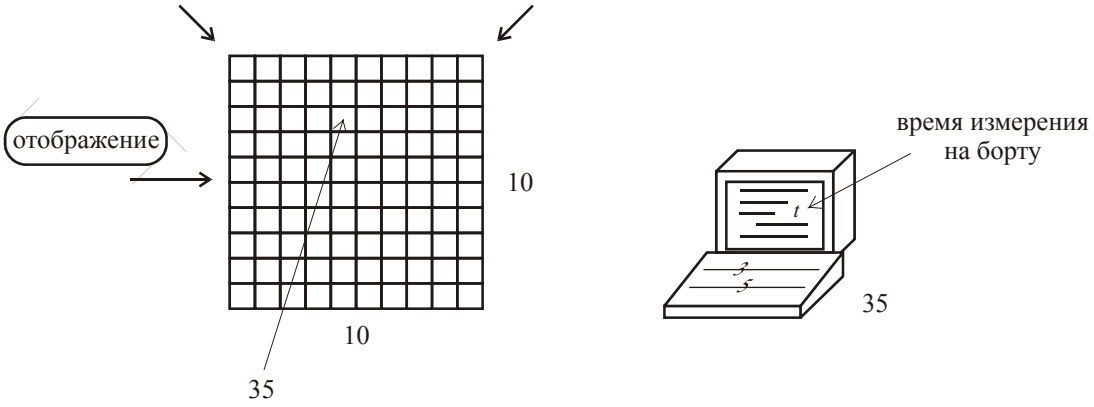
Давайте теперь здесь подсоединим два очень интересных объекта. Реально комплекс такой использовал порядка 8 узлов, которые что-то делали (считали или хранили данные), а в принципе был рассчитан на 16 узлов. И вот здесь находилась *периферийная машина* (ПМ-6). Она была названа периферийной потому, что её была цель управлять внешними устройствами (ВУ). На этой машине никакой целевой обработки данных не велось. Фактически на ней работала только операционная система, состоящая из 2-х слоёв — это программа управления устройствами (т.е. драйвер — нижний слой, канальные подпрограммы, имея в виду каждое устройство как подканал). На самом деле здесь тоже был некий канал второго уровня, об этом чуть позже (см. Рис. 19). Так он был назван по сравнению с этим первым. И плюс к слою канальных программ, конечно, было какое-то количество мониторных программ. Они не управляли устройствами. И состояла эта

не управляли устройствами. И состояла эта периферийная машина из двух частей. Одна называлась *процессор* для выполнения этих программ. Собственной памяти у ПМ-6 не было, сама операционная система и данные находились в памяти (5) или (6). Этот процессор уже не был полноразрядным, если иметь в виду ЦП. Здесь были 2-х байтовые регистры и операции над двумя байтами, что вполне достаточно было для некоей обработки передаваемой/принимаемой информации. Конечно, здесь не было операций с плавающей запятой. Вторая часть — *каналлер*. Фактически — это был некий симбиоз селекторного и мультиплексного каналов.

Были комплексы с двумя ЦП, но их было не много. В основном были с одним ЦП. Такие были поставлены в Институт прикладной математики, было 3 основных центра управления полётами космических аппаратов.

Здесь я нарисую ещё одно очень интересное устройство — *адаптер телеметрии и отображения* (АТМО). Сюда входили потоки телеметрической информации (так называемой, широкополосной) — потоки информации телевизионного типа (с высокой скоростью). Телеметрия шла через телефонные каналы разные, от радиотехнических станций, которые находились на бортах космических аппаратов (сейчас на МКС десятка полтора). И здесь шло отображение (поэтому «О» — адаптер телеметрии и отображения).

Отображение шло вот на такую память, которая представляла собой некую клеточную структуру (но это чисто условно для нашего более удачного понимания). Это была некая память, специально сделанная, куда передавались данные для отображения сразу на машину. Вот с этого комплекса, как вы видите, с других каких-то комплексов передавались сюда данные. И эта информация была в таком виде, чтобы можно было на экран любого терминала, у которого было 2 ряда кнопок по 10 в каждом ряду, и таких было 100 окон (10 на 10), ячеек, квадратов, как угодно. И когда мы здесь набирали какой-нибудь, например, 35 — это было 35-е окно (35-я программа), и вот здесь появлялась информация в буквенно-цифровом виде та, которую вот эти комплексы, машины в результате своей обработки туда поместили. Поэтому группы управления получали эту информацию: какое-то количество квадратов исследовала одна группа управления, другая группа — другие.



А, правда, смотреть все могли всё. Была полная демократия, везде можно было смотреть всё. Кругом было с большой степенью секретности, кругом форма допуска и прочее, прочее. А вот здесь, действительно, можно было смотреть всё.

Вот это всё находилось в одном зале, а машина БЭСМ-6 — в другом зале. В этом же зале в другой его половине стоял другой такой комплекс, а машина БЭСМ-6 была другая во втором комплексе и стояла в зале, где была и первая. И через 2 коридора на расстоянии 50 метров находился зал оборудования.

Ну и конечно, всё можно было смотреть, настолько это было интересно, все смотрели как происходит стыковка, расстыковки, всякие другие вещи. Везде можно было, нажав 01 или 02, смотреть телевидение внутри, снаружи, как выходили, или просто смотреть снаружи, что показывает наружная камера — масса интересного.

И, конечно, вот вам этапы:

1) принималась телеметрическая информация, периферийная машина (ПМ) управляла передачей этих данных в оперативную память (ОП);

2) центральный процессор (ЦП) вёл, так называемую, предварительную обработку информации, выделяя существенные изменения (потом он, правда, занимался научной обработкой);
3) выделенные существенные изменения (всё это в общей памяти, естественно), подхватывала информацию об этом машина БЭСМ-6, осуществляла научную обработку — подготавливали данные для отображения (естественно, что использовались и всякие другие устройства для передачи данных);
4) включалась снова периферийная машина (ПМ), выдавая это на отображение.

В результате, когда мы смотрим здесь какой-то параметр, с ним связывается *t* (время), когда это измерение было произведено, но на борту. Время измерения на борту отличалось от времени, которое мы видели на часах на стенке, на 5-6 секунд за счёт конвейера.

Я вам рассказывал случай, когда машина зациклилась из-за ошибки в программе обработки телеметрии? Сейчас расскажу. Было это давно. Сейчас у нас МКС летает, до этого «Мир» летал, до него ещё «Салюты». Так вот на одном из «Салютов», даже может быть, на «Салюте-6», вышли люди на поверхность станции дела делать. И вот, естественно, динамическая операция, всё идёт нормально, мы сидим в зале 6, смотрим сюда. И вдруг по громкой связи: «Зациклилась БЭСМ-6!» Я принимаю низкий старт, пролетаю эти 50 метров, останавливаю машину. Для чего? Чтобы посмотреть адрес: является ли это адрес программы пользователя (помните защищённые подсистемы), либо это адрес программы операционной системы. Горит ли там лампочка, которая говорит о разных режимах? Если горит, значит, это ОС. Если нет, значит, это у пользователя. Дальше делаю останов, пуск, останов, пуск, останов, пуск — смотрю на (помните БЭСМ-6) счётчик адреса и запоминаю цифру. Вижу цифру — идёт пользовательский цикл. И пока я смотрел, зациклилась вторая БЭСМ-6. Я бегу туда, делаю то же самое, вижу тот же самый цикл — дублирование. Не успел я посмотреть ещё раз цикл, хотя уже видно, что тот же самый, как голос: «Перезапустить оба комплекса!» Нажимаю на кнопку, через 5-6 секунд снова комплекс уже в работе, снова обрабатывает информацию.

Хорошо, перезапустили оба комплекса, проходит какое-то время, и снова обе машины зацикливаются. Получилось, что сеанс не обработался (телеметрию не обработали). Конечно, срочно все прибежали, все смотрели, анализ нужно делать за час не больше, час-два — появится снова. Сделали анализ — поняли, что информация, которая активно пошла во время выхода на поверхность станции, не была приспособлена программа, содержала ошибку, не потянула бы, как эту информацию надо обрабатывать. Значит, конечно, терминал, который находится здесь, перепрограммировали (управление было продумано), сценарий программы обработки телеметрии изменили. В следующем сеансе все пошло нормально.

Вот такой был беспокойный момент, когда это всё произошло.

Предстоит нам понять, что же здесь передаётся. Вот информация здесь передавалась вот этими 96-ю разрядами. Что они собой включали? Это было одиночное сообщение, которое находило себе здесь дорогу, и, так сказать, соревновалось с другими сообщениями по занятию плечей передачи к тому абоненту, к которому оно должно быть передано.

Это мне вспомнилось аэропорт Хитроу. Как-то мы полетели на конгресс по обработке информации. Это международная организация, она постоянно действующая. А до тех пор она собиралась раз в четыре года. Возглавляли этот конгресс поочерёдно представители разных государств. Нас поехала делегация 20 человек, ещё поехало 30 научных (это называется «научный туризм» — за свои деньги, но половину стоимости брало на себя государство). Вот — 50 человек. Американцы привезли 2000.

Когда мы прилетели, была забастовка. А мы должны были лететь из Хитроу, т.е. Лондон-Эдинбург на British Airlines, собственно и полетели, только часов через 6. Так вот там в Хитров, когда мы сидели в вынужденном бездействии, ожидая полёта, там сверху было видно: несколько полос (не очень высокий аэропорт, но сверху очень хорошо видно, как летают,

садятся самолёты). А когда они все идут, они друг другу уступают дорогу: большие — маленьким, маленькие — большим, пока выползают на взлётные полосы. Здесь вот приблизительно то же самое.

Сообщения уступают друг другу дорогу по какому-то приоритету. Если вот сюда, например, пришли, вот сюда, вот сюда и вот сюда, а нужно передать вот сюда все их, чтобы потом куда-то здесь, то тогда, конечно, какой-то приоритет. Так вот, из чего же состоит это слово? Оно состоит из двух частей:

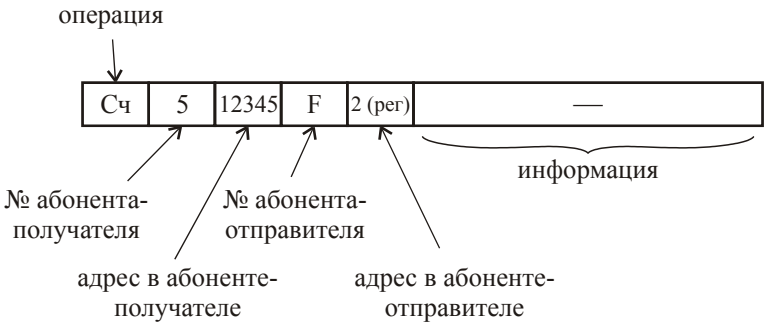


Рис. 20 Сообщение

Такое вот сообщение, аппаратно формируемое и прогоняемое по сети каналов первого уровня. Номер абонента-получателя, скажем 5 (см. Рис. 19). 16 абонентов — шестнадцатеричная система. Допустим, мы хотим записать данные с какого-то регистра ЦП (F) в память (5). Тогда номер абонента-получателя 5, адрес в абоненте-получателе 12345 (адрес ячейки), номер абонента-отправителя (в данном случае F) и адрес в абоненте-отправителе (скажем, 2 — второй регистр). И тогда это попадает сюда, а в К1 зафиксирована картина сети. В начале сделали так, что можно было специальными командами настроить К1, а потом решили, что изменять придётся раз в год, а может — никогда. Сложно перепаять что ль? «Нельзя делать победу науки над здравым смыслом».

И конечно, здесь было всё так запаяно, что когда приходит и написано «5», совершенно ясно, что это нужно передавать сюда. Каждый коммутатор знает картину сети.

Номер абонента-отправителя и адрес в абоненте-отправителе — как бы, и без этого всё дойдёт. Во-первых, это может быть для некоего контроля, или может быть просто сложный комплекс, какому-то абоненту нельзя обращаться к какому-то другому абоненту в какое-то время и т.д., будет нарушение — поймают, прерывание и всякие прочие, связанные с этим дела.

Теперь давайте не запись, а считывание. Что это означает? Это означает, что из 5-го, ячейки 12345, прошу передать мне на мой второй регистр. В данном случае в поле информации ничего нет. Оно формирует ответное сообщение, которое я могу нарисовать здесь смело:

Зп	F	2	5	12345	информация

А если групповая передача, что возможно, значит, разбивается автоматом на отдельные какие-то сообщения. Вот так и проходит с высокой скоростью передача этих сообщений в сети каналов первого уровня.

Теперь, что собой представляет связь с внешними устройствами (ВУ) — это довольно интересная вещь. Я рисую вот эту периферийную машину (ПМ), я в обратную сторону нарисую, мне так удобнее будет. Процессор, каналлер, и вот здесь идёт канал, который соединяет с неким коммутатором К2 (коммутатор не просто второй, а именно коммутатор второго уровня), и от него идёт 8 выходов. Естественно, это разбивается на сообщения, и идёт приём в память или из памяти куда-то там в другую память. Здесь может быть подсоединено устройство управления (УУ), и за ним какое-то количество устройств — до 32-х. И так на каждом выходе/входе. И вот здесь можно подсоединить второй каскад второго уровня (третьего не будет), где до 4-х устройств.

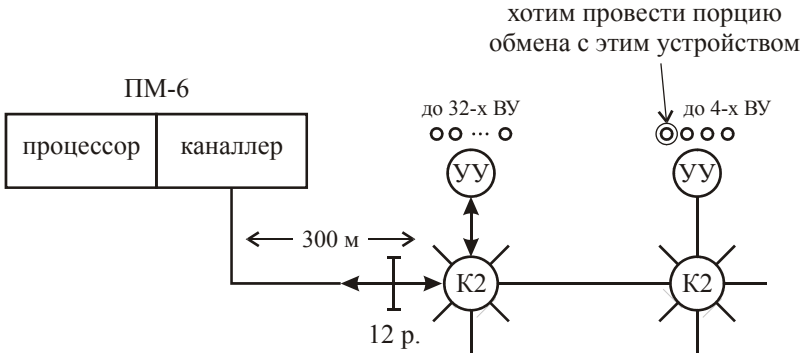
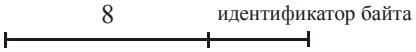


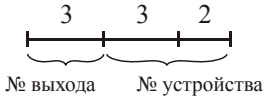
Рис. 21 Связь с внешними устройствами

Тут очень интересно организована передача. Опять же, если это диски, то на одном и том же канале обеспечивается селекторный режим. Если это не диски, а устройства остальные, то мультиплексный режим. Поэтому, это такой симбиозный селекторно-мультиплексный канал.

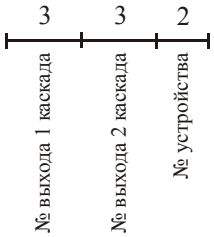
Как это всё происходит? Вот допустим, вы хотите провести порцию обмена с устройством второго каскада (см. Рис. 21). Тогда устанавливается сначала дорога до этого устройства. Ширина этого канала — полтора байта (8 + 4 = 12 разрядов), там было 96 разрядов, и расстояние здесь уже 300 метров:



Идентификатор (вообще здесь 4 разряда, там для других целей) **00** — это управляющий байт. Он содержит номер устройства. Что собой представляет этот номер:



Делится на 3, 3 и 2. Первые 3 — это *номер выхода* на первом каскаде канала второго уровня. Если здесь какое-то устройство управления (УУ), допустим, до 32-х (такие были телефонные каналы — до 32-х телефонных каналов), то тогда это всё (вторые 3 и последние 2) — это *номер устройства*. Если же у вас 2 каскада, то у вас 3 — *номер выхода на 1-м каскаде*, 3 — *номер выхода на 2-м каскаде* и 2 разряда — *номер устройства*:

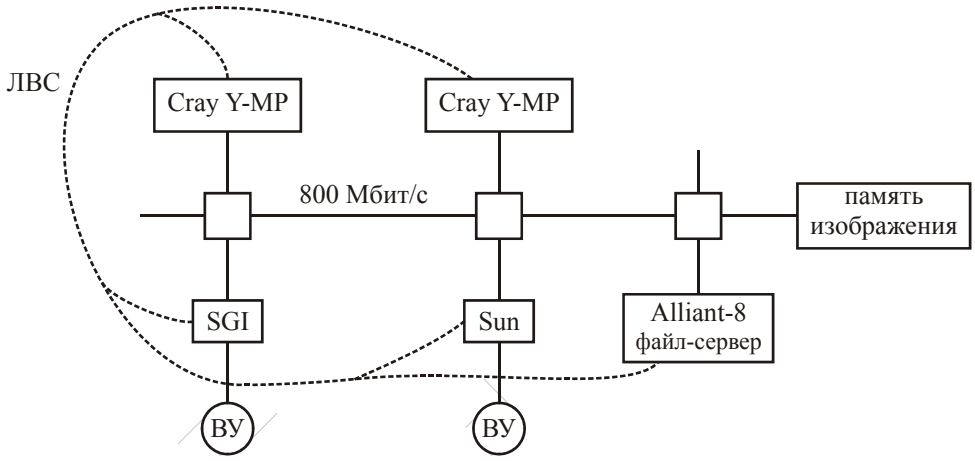


Ясно, что при этом до 4-х устройств, больше не получается.

01 — это текущий («рабочий») байт, и, допустим, **10** — это последний байт. Всё это захватывается как шина на время передачи какого-то количества данных. Затем идёт на обслуживание другого устройства в мультиплексном режиме.

Теперь, что касается Сан-Диего. Что они там сделали? Сделали они вот что. Они сделали систему обработки изображений. Там были так же точно некие обрабатывающие звенья. Обрабатывающими звеньями были Cray и не какой-нибудь, а Cray Y-MP, т.е. многопроцессорный — богатые, что скажешь. Была система каналов первого уровня, к которой кроме основных 8 вычислителей, подсоединялись станции Silicon Graphics (SGI) — графические станции ввода/вывода, станции Sun — это вместо периферийных машин, которые занимались как раз вопросами подготовки ввода и вывода, некой дополнительной обработки. Здесь была машина Alliant-8, используемая как файл-сервер. Я повторяю, что это 800 Мбит/с. И вот здесь, смотрите, почти то же самое, только, конечно, не АТМО, система обработки изображений — *память изображения*, т.е. именно сюда слали в некоем специальном формате, удобном затем для взятия и изображения, и затем они уже брались этими основными про-

цессорами и машинами. Т.е. появилась надобность обрабатывать изображения, и возникла вот такая система.



Правда, они ещё говорили, что на всякий пожарный случай, они вот это дело всё ещё некой локальной сетью объединили. Но основная передача была точно так же, как в комплексе АС-6.

Заключение

Я на этом завершаю. Позволю вам прочесть последние слова. Вот ещё раз вам говорю: господа, быть истинно культурными людьми в нашей области, я вроде как рекламирую, да что там рекламирую — книга (Ф.Брукс «Мифический человеко-месяц») уже второе издание мирового бестселлера, второе издание международного бестселлера. В библиотеках должна быть. Напоминаю: первое издание было 1975 год, подпольное издание у нас — 1976, неподпольное — библиотечка программиста — 1978. Затем, Фредерик Брукс известен как отец IBM System 360, имея в виду программное обеспечение в первую очередь. Известнейший человек в мире. В 1995 году он написал вторую книгу, повторяло издание 75-го года, и дальше идёт его анализ, что из предложенного в первом опыте прижилось, что не прижилось, в общем, делает серьёзный интересный анализ.

Первая книга была очень содержательна. Там была масса, конечно, эмоциональных моментов, но очень многое прижилось. Но многое принято рассматривать в новых условиях, новый подход к созданию программного обеспечения. Это практически системщик — как создавать системное программное обеспечение.

Давайте я вам прочту последние страницы.

«**Эпилог**

Пятьдесят лет удивления, восхищения и радости

В моей памяти все еще живы удивление и восторг, с которым я — мне тогда было 13 лет — читал отчет от 7 августа 1944 года об освящении компьютера Mark I, архитектором которого был Говард Айкен (Howard Aiken), а проектировщиками - инженеры Клер Лейк (Clair D. Lake), Бенджамин Дурфи (B. M. Durfee) и Фрэнсис Гамильтон (F. E. Hamilton). Такой же вызывающей ощущение чуда была статья Ванневара Буша (Vannevar Bush) "That We May Think" в апрельском 1945 года номере "Atlantic Monthly", в которой он предложил организовать знания в виде огромной гипертекстовой паутины и обеспечить пользователей машинами для переходов по существующим ссылкам и создания новых ассоциативных следов.

Новый толчок моя страсть к компьютерам получила в 1952 году, когда, работая летом на IBM в Эдинкоте, штат Нью-Йорк, я получил практический опыт программирования для IBM 604 и формальное обучение программированию для IBM 701, их первой машины с хранимой программой. Аспирантура у Айкена и Иверсона в Гарварде сделала реальностью мои мечты о профессии, и я связал с ней всю свою жизнь. Немногим Бог дает право зарабатывать на жизнь тем, чем они с радостью занимались бы по собственной воле, по увлечению. Я благодарен судьбе.

Для человека, влюбленного в компьютеры, трудно было бы придумать иное время, когда так радостно было жить. От механических устройств до вакуумных ламп, транзисторов и интегральных

схем шло бурное развитие технологии. Первый компьютер, на котором я работал сразу после выпуска из Гарварда, был суперкомпьютер IBM Stretch. Этот компьютер царствовал над миром как самый быстрый с 1961 по 1964 годы; было изготовлено 9 экземпляров. Мой сегодняшний Macintosh Powerbook не только быстрее, с большей памятью и большим диском, но и в тысячу раз дешевле (в пять тысяч раз дешевле с учетом инфляции). Мы были свидетелями того, как поочередно произошли компьютерная революция, революция электронных компьютеров, революция миникомпьютеров и революция микрокомпьютеров, в результате каждой из которых компьютеров становилось на порядки больше.

Область связанных с компьютерами знаний претерпела взрыв, как и соответствующая технология. Будучи аспирантом в середине 50-х, я мог прочесть все журналы и труды конференций. Я мог оставаться на современном уровне во всей научной дисциплине. Сегодня же мне в моей интеллектуальной жизни приходится с сожалением расставаться с интересами то в одной, то в другой подобласти, поскольку количество документов превысило всякую возможность справиться с ними. Масса интересов, масса замечательных возможностей для учебы, исследований, размышлений. Чудесное затруднение! Не только конца не видно, но и шаг не замедляется. В будущем нас ожидают многие радости.»

Вот если вы хотите действительно почувствовать удовольствие от этого, я бы вам советовал найти старый вариант. Дело в том, понимаете, что тот, кто составлял это, дал кому-то подработать. Они снова перевели первое издание 75-го года. Тут то же самое, но там один в один, а здесь случился второй перевод. Тот перевод, который сделан был, на много лучше.

Вопросы по курсу «Вычислительные системы»

1. Способы классификации архитектур ВС.
2. Управление потоками команд и потоками данных в ВС.
3. Уровни параллелизма обработки информации в ВС.
4. Структуры и примеры универсальных ЭВМ (БЭСМ-6).
5. Конвейерность выполнения вычислений и обработки команд в ЭВМ.
6. Векторно-конвейерные ЭВМ (Сгау-1, МКП).
7. Типы и примеры многопроцессорных вычислительных комплексов (МВК). ILLIAC IV, Эльбрус-2, SP-2.
8. Общая и распределенная память МВК.
9. Способы объединения процессоров в МВК.
10. Классификация наборов команд ЭВМ. CISC и RISC архитектуры.
11. Состав средств аппаратной поддержки работы операционных систем.
12. Аппаратура прерываний.
13. Аппаратура многоуровневой защиты в ЭВМ.
14. Специальные регистры и команды процессора для поддержки обработки прерываний и переключения процессора.
15. Аппаратная поддержка взаимодействия программных модулей.
16. Иерархия запоминающих устройств.
17. Организация памяти типа cache.
18. Организация оперативной памяти. Односегментное отображение.
19. Организация оперативной памяти. Сегментация.
20. Страничная организация оперативной памяти.
21. Сегментно-страничная организация оперативной памяти.
22. Организация виртуальной памяти.
23. Типы устройств внешней памяти и ввода-вывода.
24. Способы организации доступа к внешней памяти и устройствам ввода-вывода.
25. Селекторные и мультиплексные каналы связи с периферией ЭВМ. Цикл работы устройства "Мультиплексный канал".
26. Использование шинной архитектуры для связи с периферией ЭВМ.
27. Назначение и типы многомашинных вычислительных комплексов (ММВК). Примеры ММВК. Конвейеры ЭВМ в ММВК.
28. Организация передачи данных в ММВК с общедоступной памятью (АС-6).
29. Организация доступа к общей периферии в ММВК.
30. Организация суперЭВМ как ММВК.

Литература к курсу «Вычислительные системы»

1. **Воеводин В.В., Воеводин Вл.В.** Параллельные вычисления. **СПб, «БХВ - Петербург», 2002**
2. **Королев Л.Н.** Архитектура процессоров электронных вычислительных машин. **М, МГУ им. М.В. Ломоносова, факультет ВМ и К, 2003**
3. **Немнюгин С., Стесик О.** Параллельное программирование для многопроцессорных вычислительных систем. **СПб, «БХВ - Петербург», 2002**
4. **Корнеев В.В.** Вычислительные системы. **М, «Гелиос АРВ», 2004**
5. **Корнеев В.В.** Параллельные вычислительные системы. **«Нолидж», 1999**
6. **Столлингс У.** Структурная организация и архитектура компьютерных систем. **М, СПб, Киев, «Вильяме», 2002**
7. **Таненбаум Э.** Архитектура компьютера. **СПб, «Питер»**
8. **Королев Л.Н.** Структуры ЭВМ и их математическое обеспечение. **М, «Наука», 1978**
9. **Смирнов А.Д.** Архитектура вычислительных систем. **М, «Наука», 1990**
10. Малые ЭВМ высокой производительности. Архитектура и программирование. **М, «Радио и связь», 1990**
11. **Цикритзис Д., Бернстайн Ф.** Операционные системы. **М, «Мир», 1977**
12. **Амамия М., Танака Ю.** Архитектура ЭВМ и искусственный интеллект. **М, «Мир», 1993**
13. http://www.parallel.ru
14. http://osp.ru
15. http://citforum.ru
16. http://ccas.ru/paral/