Глава 14. Архитектурные особенности вычислительных машин

Раньше назначение программ заключалось в управлении нашими вычислительными машинами, теперь назначение вычислительных машин состоит в исполнении наших программ.

Эдгар Дейкстра

14.1. Несоответствие скоростей работы процессора и оперативной памяти

Зло есть не что иное, как несоответствие между бытием и долженствованием 🙄.

Георг Вильгельм Фридрих Гегель

Для лучшего понимания тех важных архитектурных особенностей, которыми обладают современные компьютеры, сейчас будет оценена скорость работы различных устройств ЭВМ. Оперативная память персональных ЭВМ массового производства способна считывать и записывать данные примерно каждые 5-10 наносекунд (нс.), 1 нс. = 10^{-9} сек.. На современных ЭВМ это примерно 15-30 тактов ЦП), а процессор может выполнить машинную операцию над содержимым своих регистров менее, чем за 1 нс. [i] [см. сноску в конце главы] После некоторого размышления становится понятным, что «что-то здесь не так».

Действительно, рассмотрим, например, типичную машинную команду `add eax,X`. Для выполнения этой команды процессор должен сначала считать из оперативной памяти саму эту команду (в рассматриваемой архитектуре это 6 байт), затем операнд `X` (это ещё 4 байта), потом произвести операцию сложения. Таким образом, процессор потратит на выполнение этой команды

(Чтение команды и числа) 10 байт*5 нс.+ (Выполнение команды) 1 нс. = 51 нс.,

причём собственно процессор будет работать только 1 нс., и 50 нс. будет ждать, пока команда и число поступят из оперативной памяти на его регистры. Спрашивается, зачем делать процессор таким быстрым, если всё равно более 98% своего времени он будет ждать, пока производится обмен командами и данными между оперативной памятью и его регистрами? Налицо явное *несоответствие* в скорости работы оперативной памяти и процессора ЭВМ, для иллюстрации этого положения на рис. 14.1 показано соотношение роста скорости работы процессоров и оперативной памяти.

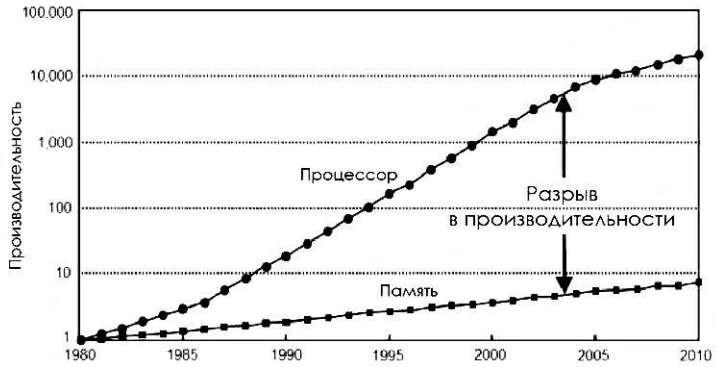


Рис. 14.1. Рост скорости работы процессора и оперативной памяти ЭВМ.

Скорость работы процессора примерно определяется его тактовой частотой. С 1980 по 2010 годы тактовая частота выроста с 20 Мгц до 4 Ггц, т.е. примерно в 20000 раз. В то же время скорость работы оперативной памяти выросла менее, чем в 10 раз. Это связано с тем, что наша оперативная память хранит свои данные-биты в крошечных конденсаторах, перезарядку которых нельзя сделать очень быстрой (это порядка 5 наносекунд). Далее уменьшать ёмкость этих конденсаторов уже нельзя, они просто станут очень быстро разряжаться «сами по себе». А вот скорость переключения элементов

процессора связана только со временем изменения напряжения на затворе транзистора, а это время измеряется уже пикосекундами (1 пс. = 10^{-12} сек.).[1]

Итак, память работает значительно медленнее процессора, хотя, начиная с 2005 года, рост производительности процессора сильно замедлился, в основном это связано с тем, что где-то в последние 10 тактовая частота процессора перестала расти. Выделение тепла пропорционально четвёртой степени частоты работы, поэтому сейчас частота работы обычно не выше 4 Ггц.[2] Кроме того, при столь маленьких размерах транзисторов число электронов, «бегающих туда-сюда» в районе затвора транзистора при его переключении, уже измеряется буквально сотнями штук, при дальнейшем увеличении частоты транзистор просто перестанет работать. К сожалению, сделать затвор транзистора толщиной менее примерно 5 нм. не получается, вступает в силу квантовый тоннельный эффект, электроны пересекают затвор не «обращая внимания» на напряжение на затворе, и транзистор тоже перестаёт работать.

Первое, что приходит в голову для преодоления этого несоответствия, это увеличить на порядок скорость работы оперативной памяти. Технически это вполне возможно, однако при этом стоимость компьютера также сильно возрастёт, что, конечно же, недопустимо, так как эти дорогие ЭВМ просто не будут покупать в таких количествах. Кроме того, как вскоре будет ясно, одно лишь увеличение скорости работы оперативной памяти ненамного улучшит положение.

На современных ЭВМ проблема несоответствия скорости работы оперативной памяти и процессора решается в совокупности несколькими способами, которые сейчас и будут рассмотрены. Сначала, выяснив, что главным тормозом в работе является оперативная память, эту память стали делать таким образом, чтобы за одно обращение к ней она выдавала не по одному байту, а сразу по несколько байт с последовательными адресами. Для этого оперативную память разбивают на отдельные микросхемы, которые могут работать параллельно. Этот приём называют расслоением памяти (interleaved memory, address interleaving).[3] Например, если память разбита на 4 микросхемы, каждая из которых может выдавать два байта, то за одно обращение к такой памяти можно сразу считать 8 байт, с последовательными адресами (каждая микросхема выдаёт по два байта памяти). Таким образом, за одно обращение к памяти можно считать несколько команд или данных, расположенных в памяти подряд. Сейчас такой набор микросхем называется банком памяти, причём память может быть связана с процессором не одним, а двумя или тремя каналами, так что за одно обращение такая память с двумя или тремя банками памяти выдаёт сразу не 8, а 16 или 24 байта с последовательными адресами [ii] [см. сноску в конце главы].

Несмотря на то, что память работает много медленнее процессора, её скорость очень велика, пропускная способность современной динамической памяти сейчас примерно 24 Гбайт/сек. Таким образом, например, вся оперативная память объёмом 12 Гбайт может быть прочитана за 0.5 секунды. Очень впечатляюще, не правда ли?

Здесь требуется какое-то образное сравнение, чтобы это почувствовать. Так, если считать, что каждый байт хранит символ печатного текста и учесть, что на стандартной странице обычной книги помещается примерно 2000 символов, то получается, что за 1 секунду процессор обычной пепрсональной ЭВМ может прочитать из оперативной памяти $12*10^6$ страниц, т.е. большую библиотеку из 24000 томов по 500 страниц в каждом томе ⚠. Учтите, что скорость работы топовых моделей памяти графических карт ещё больше, порядка $5*10^{12}$ байт/сек=5 Тбайт/сек! Именно на такой памяти хорошо себя чувствует искусственный интеллект 🙂.

Легко, однако, вычислить, что, несмотря на эту огромную скорость, даже такая, обладающая расслоением по банкам, оперативная память продолжает тормозить работу процессора. Проведя заново расчёт времени выполнения нашей предыдущей команды **add** eax,X получим:

5 нс. (чтение команды) + 5 нс. (чтение числа) + 1 нс. (выполнение команды) = 11 нс.

---

[1] К сожалению, скорость переключения затвора транзистора определяется не скоростью света, а скоростью движения электронов и «дырок» в полупроводнике, а это не более 10 км./сек.

[2] Проводились эксперименты, когда удавалось поднять частоту работы процессора более 8 Ггц., но его пришлось при этом охлаждать жидким азотом 🙄.

[3] Расслоение памяти (на две схемы) впервые появилось в компьютере IBM STRETCH в 1961 году, а в отечественной ЭВМ БЭСМ-6 в 1967 году.

Как видно, хотя ситуация и несколько улучшилась, однако всё ещё 90% своего времени процессор вынужден ждать, пока из оперативной памяти поступят нужные команды и данные. Для того чтобы исправить эту неприятную ситуацию, в архитектуру современных компьютеров встраивается специальная память, которую называют памятью типа кэш (иногда кеш), или просто кэшем (cache).

Впервые термин кэш (англ. *cache*, фр. *cacher* – прятать, т.е. это «тайная», спрятанная память) появился в 1968 году в журнале IBM System Jornal с лёгкой руки его редактора Лайла Джонсона (Lyle R. Johnson). Статья была посвящёна усовершенствованию памяти в компьютерах семейства IBM-360. Впервые кэш-память появилась в отечественной ЭВМ БЭСМ-6 в 1968 году, там она называлась «по-русски» *свехоперативной памятью*. Более совершенная кэш-память была реализована в ЭВМ IBM-360 в 1969 году. В персональных ЭВМ кэш-память команд и данных появилась только в 1985 году в процессоре Intel 386.

Кэш делается на очень быстрых (и дорогих) интегральных схемах *статической* памяти SRAM и работает с такой же скоростью, как и сам процессор (на той же тактовой частоте), т.е. может, например, выдавать по 64 байта за такт. Это особая *безадресная* память, т.е. команды и числа в ней не имеют «обычных» адресов, как уже упоминалось, она называется *ассоциативной* памятью.

Для программиста кэш является невидимой памятью в том смысле, что эта память не адресуемая, к ней нельзя обратиться из программы по какой-либо команде чтения или записи данных. Конечно, существуют команды для работы с кэшем как с единым целым, это, например, команды очистки кэша от всех находящихся там команд и данных: **`invd`** (очистка без сохранения) и **`wbinvd`** (очистка с сохранением изменённых данных в оперативной памяти). Начиная с процессора Pentium 4 можно было выполнять и другие операции с этой памятью: **`prefetch`** (загрузит в кэш строку из оперативной памяти), **`clflush`** (очистить, т.е. сделать недействительной строку кэш). Кроме того, существуют и команды машины, которые обмениваются данными с памятью, минуя кэш (далее будут примеры). Для эффективной работы программист должен учитывать особенности работы кэш памяти в своём алгоритме.

Процессор работает с кэшем по следующей схеме. Когда процессору нужна какая-то команда или данное, то сначала он смотрит, не находится ли уже эта команда или данные в кэше, и, если они там есть, читает их оттуда, *не обращаясь* к оперативной памяти, это называется кэш попадание (cache hit). Разумеется, если требуемой команды или данных в кэше нет, то случился промах кэша (cache miss), и процессор вынужден читать их из относительно медленной оперативной памяти, сначала считывая данные в кэш, а затем получая их из кэша. Стоит также заметить, что вместе с требуемыми байтами в кэш одновременно попадают и определённое количество соседних с ними байтов памяти. Обычно это называется *строкой* (или линейкой) кэш памяти (cache line), типичная длина строки 64 байта, что как будет видно далее, является очень полезным для увеличения быстродействия ЭВМ. Кроме того, процессоры выполняют и *опережающее* чтение в кэш данных из ячеек памяти, следующих за текущей командой и операндами.

Аналогично, при записи данных процессор записывает их в кэш, и помечает, что эта строка теперь не совпадает со строкой в оперативной памяти. Особая ситуация складывается, если требуется что-то записать в кэш, а свободной строки там нет. В этом случае по специальному алгоритму из кэша удаляется некоторая строка, обычно та, к которой дольше всего не было обращения из процессора, это алгоритм LRU (Least Recetnly Used). Так проще всего сделать, если вести *очередь* обращений к данным в кэше, при каждом чтении или записи данных в некоторую строку, эта строка ставится в начало такой очереди. Легко понять, что тогда в конце очереди автоматически соберутся те строки кэша, к которым дольше всего не было обращения. При этом, если эти данные в кэш памяти изменялись, то они переписываются в оперативную память, причём возможна так называемая *отложенная запись* (lazy write), когда строки кэша сначала помещаются в небольшой временный буфер записи (write buffer), а «настоящая» запись в оперативную память будет производится, когда эта память не занята другими (более срочными) обменами. Польза отложенной записи заключается ещё и в том, что если до «настоящей» записи в оперативную память программе опять понадобится строка кэша из буфера записи, то она *возвращается* в кэш без её чтения из оперативной памяти. Можно сказать, что это маленький «кэш для кэша» 🙂 [iii] [см. сноску в конце главы].

Проведённое программное моделирование работы кэш-памяти показало, что, как ни странно, алгоритм удаления при промахе *случайной* строки по эффективности практически не отличается от остальных стратегий (такой алгоритм, например, реализован в процессорах фирмы AMD).

В архитектуре многих ЭВМ кэш разделяется на кэш команд и кэш данных. При этом обычно команды в программе запрещается менять, поэтому в кэш команд очень редко производится запись, что повышает его эффективность. [iv] [см. сноску в конце главы]

Таким образом, в кэше автоматически накапливаются наиболее часто используемые команды и данные выполняемой программы, например, все команды не очень длинных циклов после их первого выполнения будут находиться в памяти типа кэш. Часто говорят, что кэш образует *буфер* между быстрой регистровой памятью процессора и относительно медленной оперативной памятью (буфером обычно называется устройство, сглаживающее неравномерность скорости работы других взаимодействующих между собой устройств). Количество промахов кэша на современных процессорах меньше 5% от общего числа обращений в память. На рис. 14.2 показана схема взаимодействия процессора и оперативной памяти с использованием кэша. [v] [см. сноску в конце главы]

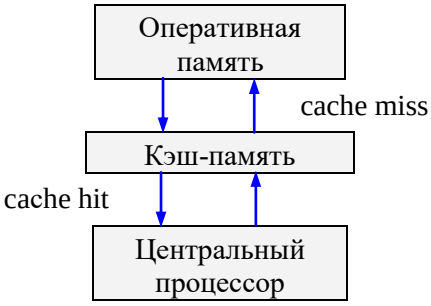


Рис. 14.2. Схема обмена с использованием памяти типа кэш.

Как видно из алгоритма работы кэша, он весьма «болезненно» реагирует на прерывания, так как при этом производится переключение на *другую* программу. При этом команды и данные в кэше, относящиеся к прерванной программе, постепенно *вытесняются* командами и данными *новой* программы. Таким образом, при возобновлении счёта прерванной программы команды и данные в кэше будет необходимо обновить (заново считать из ОЗУ), что замедляет счёт программ при частых прерываниях. Смена большой части кэш памяти (первого уровня) может обойтись в десятки и сотни тысяч процессорных тактов, поэтому, как уже говорилось, сигнал прерывания от внешних устройств направляется ядру, занятому наименее приоритетной работой.

Когда случается промах кэша, то приходится достаточно долго ждать, пока из кэша следующего уровня или из оперативной памяти считается нужная строка.[1] Исходя из этого, специальный *механизм предвыборки* производит *опережающее* чтение строк из оперативной памяти в кэш, пытаясь угадать, какие команды и данные вскоре понадобятся. В простейшем случае просто читаются следующие строки, уповая на то, что по принципу локальности программа движется по памяти последовательно. Более интеллектуальная предвыборка учитывает статистику промахов кэша, пытаясь предсказать поведение конкретной программы (так работают процессоры x86, начиная с Pentium 4). Кроме того, современные процессоры позволяют программисту на Ассемблере (и компиляторам), используя команду **`prefetch`**`{0,1,2} m8`, заблаговременно загружать нужные строки в кэш-память:

```
prefetch0 m8; Загрузить строку в кэш  L1,L2,L3
prefetch1 m8; Загрузить строку в кэш  L2,L3
prefetch2 m8; Загрузить строку только в кэш  L3
```

К сожалению, для повышения эффективности никаких проверок на «чужую» память не делается, ошибка возникает только потом при «настоящем» чтении команды или данных.

Как уже говорилось, память типа кэш строится из очень быстрых и, следовательно, дорогих интегральных схем, поэтому из экономических соображений её объём сравнительно невелик, примерно 5% от объёма оперативной памяти. Однако, несмотря на свой относительно малый объём, кэш вызывает значительное увеличение скорости работы ЭВМ, так как по статистике примерно 90-95% всех обращений из процессора за командами и данным производится именно в память типа кэш.

---

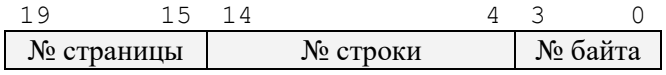
[1] К счастью, в современных ЭВМ большинство контроллеров кэш-памяти позволяют обращаться к *другим* строкам кэша *параллельно* с чтением из памяти в кэш «промахнувшейся» строки (так называемая *не блокируемая* кэш-память).

На старых персональных ЭВМ для изучения влияния памяти типа кэш на производительность компьютера можно было произвести такой эксперимент. Сначала замеряется усредненная производительность ЭВМ с помощью какой-нибудь предназначенной для этого программы. После этого при перезагрузке машины память типа кэш отключается в настройках BIOS, и снова замеряется производительность компьютера, которая при этом снижается в несколько раз. Главное, после этого эксперимента не забыть включить обратно на своем компьютере память типа кэш .На персональных компьютерах новых моделей возможность отключать кэш из BIOS обычно не предусмотрена, как совершенно бесполезная в практической работе пользователя. Кэш память в современных процессорах фирмы Intel можно отключить с помощью привилегированных команд установкой флага CD (Cache Disable) в управляющем регистре CR0.

В качестве примера будет рассмотрена одна из простейших реализаций памяти типа кэш, так называемый кэш прямого отображения (Direct Mapped Cache), смысл этого названия вскоре будет ясен из алгоритма его работы. Пусть размер оперативной памяти нашей ЭВМ составляет всего 1 Мб (2^{20} байт), и выберем объём кэш памяти равным 1/32 от объема оперативной памяти, это 32 Кб (2^{15} байт), что составляет около 3%. Далее, разобьём всю оперативную память на участки размером по 32 Кб, такие участки обычно называются *страницами*, как видно, здесь размер страницы совпадает с размером самого кэша. Каждую страницу, как находящуюся в оперативной памяти, так и страницу кэша, в свою очередь, будем рассматривать как состоящую из *строк* длиной по 16 байт. Таким образом, имеем

Вся память 2^{20} = (2^5 страниц) * (2^{11} строк) * (2^4 байт).

Тогда 20-разрядный физический адрес любого байта оперативной памяти тоже можно разбить на три поля: адрес (номер) страницы (5 бит), адрес строки в странице (11 бит) и адрес байта в строке (4 бита):



Теперь каждую строку в кэш памяти (это 16 байт или 128 бит) снабдим шестью дополнительными битами, пять из которых будут хранить номер некоторой страницы оперативной памяти, а в шестом бите будет содержаться признак изменения строки в кэш памяти, это так называемый грязный (dirty) бит (0 – из строки было только чтение, 1 – была запись). Объём этой дополнительной (служебной) памяти составляет всего около 6% кэш памяти (6 бит/16 байт), что незначительно увеличивает общий объём кэша.

Алгоритм работы процессора при наличии такой кэш памяти будет заключаться в следующем. Сначала из физического адреса байта, по которому необходимо обратиться в оперативную память, процессором выделяется номер строки (разряды с 4 по 14), и проверяется строка в кэш памяти с этим же номером. Если номер страницы, приписанный данной строке в кэше, совпадает с номером той страницы, к которой необходимо обратиться, то нужный байт уже находится именно в этой строке кэша, и обращаться в оперативную память не надо. В противном случае строку с данным номером в кэш памяти необходимо сменить на строку с этим же номером из нужной страницы оперативной памяти. Разумеется, при такой замене сначала проверяется бит-признак изменения этой строки в кэш памяти, если строка менялась, то её, конечно, надо записать на своё место в оперативной памяти.

Таким образом, получается, что, если некоторая строка оперативной памяти присутствует в кэше, то она находится в нём на том же месте, что и в своей странице оперативной памяти. Именно поэтому так организованный кэш и называется кэшем прямого отображения (строк из страниц оперативной памяти на строки кэш памяти). Заметим, что если случился промах кэша, то из оперативной памяти считывается в кэш сразу вся строка (в нашем примере 16 байт), это занимает всего одно обращения к оперативной памяти при её четырёхкратном расслоении и 128-битной шине данных (и дополнительное обращение, если в строку была запись).

Описанный выше кэш прямого отображения быстро работает и его просто реализовать, но он имеет и существенный недостаток. Заметим, что, если попеременно обращаться к байтам с одинаковыми номерами в двух страницах оперативной памяти, то строки из этих страниц будут всё время сменять друг друга в кэш памяти (это называется трудно переводимым термином trashing, т.е. «мерцание» или «дрожание» памяти). Такая ситуация, конечно, весьма негативно скажется на скорости выполнения программы. Современные компьютеры снабжаются более сложно устроенной кэш памятью, в которой вероятность описанной выше неприятной ситуации значительно ниже. Обычно это

так называемые многоканальный и полностью ассоциативный кэш (Fully-Associative Cache), его устройство напоминает хэш-таблицы, элементом которой является строка кэша, состоящая из пары <адрес,данные>.

Для ещё большего увеличения скорости чтения и записи команд и данных процессором можно включить в архитектуру не один, а два или три последовательно подключённых кэша. При этом самый внутренний (т.е. ближайший к процессору) кэш называется кэшем первого уровня (L1 – Level 1), следующий – второго уровня (L2), и т.д. Сейчас все кэши обычно располагается прямо в процессоре (там же, где регистры), редко когда кэш L3 на отдельной микросхеме. Чтения данных из L1 кэша требует (по сравнению с регистрами) примерно 3 такта, L2 – 10 тактов, L3 – 30 тактов, а из оперативной памяти –уже (в худшем случае) 100-200 тактов.

В многоядерных процессорах архитектуры x86 у каждого ядра свой кэш первого уровня L1, а вот кэши L2 и L3 могут быть как свои у каждого ядра, так и общие, это порождает большие проблемы *согласования* данных, так как одно ядро может читать из ячейки, которая уже модифицирована в кэш-памяти другого ядра. Для согласования данных принимаются специальные меры как на аппаратном (так называемые протоколы целостности (consistency) и когерентности (coherency) кэш-памяти), так и на программном уровнях [vi] [см. сноску в конце главы].

Здесь, однако, необходимо уяснить для себя следующее. Успешное применение памяти типа кэш (как, впрочем, и изученного ранее расслоения памяти) базируется на свойстве локальности программ, это свойство уже упоминалось при изучении близких относительных переходов. Свойство локальности заключается в том, что выполняемые команды и обрабатываемые данные программы не разбросаны по памяти в хаотическом беспорядке, а обнаруживают тенденцию группироваться в некоторые относительно небольшие области. Это, например, команды в теле циклов, данные внутри массивов, близко расположенные переменные в выражениях и т.д. Особенно эффективны стековые кадры процедур, при возврате они уничтожаются, но остаются в кэш памяти, что позволяет эффективно использовать их в следующих вызовах подпрограмм.

Конечно, можно специально написать программу, не обладающую свойством локальности, такая программа будет после каждой команды случайным образом переходить на выполнение следующей команды в любой части программы, а обрабатываемые данные также выбирать из областей памяти со случайными адресами. Так вот, Вам необходимо понять, что при выполнении такой «нелокальной» программы кэш будет бесполезен (и даже вреден). Конечно, это верно только тогда, когда объём команд и данных такой «нелокальной» программы будет существенно превышать размер памяти типа кэш (что, впрочем, обычно выполняется для большинства программ).

Процессор предоставляет программисту на Ассемблере средства для управления размещением данных к кэш-памяти. Это можно делать, используя так называемые не временны́е команды, они могут обмениваться данными с памятью, минуя кэш. Например, команда `vmovdqa ymm0,m256` читает 32 байта из памяти на регистр `ymm0`, как всегда, оставляя копию в кэше данных первого уровня, а вот команда `vmovdntqa ymm0,m256` читает данные из памяти на регистр напрямую, не оставляя копию в кэше (и удаляя из кэша старую копию этих данных, если они там были). Буквы **nt** в коде операции означает Non-Temporal, т.е. это данные «не временны́е» (не обладающие временно́й локальностью), они в ближайшее время процессору не понадобятся, и пусть они лучше не «засоряют» кэш.

При работе с кэш памятью из неё в буфер команд и буфер данных конвейера читается сразу по одной строке (обычно 64 байта). Вот теперь, на ЭВМ с памятью типа кэш, наша рассмотренная выше команда `add eax,X` будет при уже прочитанных из кэша строки команд и строки данных, выполняться за время

1 нс. (чтение команды) + 1 нс. (чтение числа X) + 1 нс. (выполнение команды) = 3 нс.

Как видно, ситуация коренным образом улучшилась, хотя всё равно получается, что сам процессор работает только примерно 30% от времени выполнения команды, а остальное время ожидает поступления на свои регистры команд и данных. Для того чтобы исправить эту неприятную ситуацию, конструкторам пришлось снова существенно изменить архитектуру процессора.

14.2. Конвейерные ЭВМ

Ничего особенно не трудно, если разделить работу на небольшие части.

Генри Форд

Как уже говорилось, современные ЭВМ могут одновременно выполнять несколько команд. Для этого они должны либо иметь несколько центральных процессоров (ядер), либо процессор такого компьютера строится по так называемой *конвейерной* (pipeline) архитектуре. Сейчас будет рассмотрена схема работы таких конвейерных ЭВМ.

Сначала мы рассмотрим простой конвейер, так он был устроен на предыдущих поколениях ЭВМ. Современные конвейеры устроены много сложнее, схема их работы описана в разд. 14.2.1.

Впервые принцип конвейерной обработки машинных команд сформулировал советский академик С.А. Лебедев в 1956 году на конференции в Дармштадте (ФРГ), он называл его «принципом водопровода». Может быть именно поэтому конвейер в западной литературе называют трубопроводом (pipeline) [см. сноску в конце главы]. Отметим, что на заводах конвейер чаще называют *сборочной линией* (assembly line).

Первый конвейер из четырех шагов был реализован в 1963 году на машине ATLAS. Простой конвейер имела и отечественная ЭВМ БЭСМ-6, серийно выпускавшаяся с 1968 года, в то время она была одной из самых быстродействующих в мире (1 миллион операций с вещественными числами в секунду). На *персональных* ЭВМ конвейер (пятиступенчатый) впервые реализован только в 1989 году в процессоре Intel 486.

Любопытно, что первый промышленный конвейер был запущен в американском городе Цинциннати ещё в 1860 году, но он использовался для разделки свиных туш и производстве консервов, а вот Генри Форд построил свой конвейер из 84 шагов по сборке автомобильных двигателей только в 1913 году. Во время войны с конвейера Нижнетагильского завода (Уралвагонзавод) каждые сорок минут сходил готовый (и уже заправленный топливом) танк Т-34. Всего за годы войны на заводе было выпущено белее 25 тыс. танков, это больше, чем на всех заводах фашистской Германии! Сейчас здесь производятся танки Т-72 и Т-90, с 2014 года завод находится под санкциями Евросоюза, США и ряда других стран.

Выполнение каждой команды любым процессором, как уже известно, состоит из нескольких шагов, обычно называемых *микрооперациями (мопами)*. Можно, например, выделить следующие основные шаги выполнения команды:

1. Выбор команды (F – fetch) из буфера команд, этот буфер непрерывно пополняется из кэш-памяти.

Современные процессоры просматривают и читают программу вперёд по *предполагаемому* пути её выполнения (об этом будем говорить дальше) примерно на 4 КБ (это около 1000 команд и всего 0.3-0.5 микросекунды счёта ⚠).

2. Определение кода операции и операндов (D – decode), так называемое *декодирование* команды.
3. Вычисление адресов операндов (A – address generate).
4. Выбор операндов (L – load) чтение операндов из оперативной памяти (или кэша) на регистры арифметико-логического устройства.
5. Выполнение (E –execute) требуемой операции (сложение, умножение, сдвиг и т.д.) над операндами на регистрах арифметико-логического устройства.
6. Запись (W – write back) результата операции, выработка и установка флагов.

В конвейерных ЭВМ процессор состоит из нескольких блоков, каждый из которых выполняет один из перечисленных выше шагов команды. Эти блоки стараются строить так, чтобы все они выполняли шаг команды *за одно и то же время*. Более точно, такт работы выбирается таким, чтобы успеть выполнить самый длинный шаг конвейера (для современных процессоров это 0.3-0.5 нс.). Теперь понятно, что такие блоки можно заставить работать параллельно (для разных команд), обеспечивая, таким образом, одновременное выполнение процессором нескольких последовательных команд программы. На рис. 14.3 приведена схема работы процессора конвейерной ЭВМ, направление движения команд на конвейере показано толстой серой стрелкой.

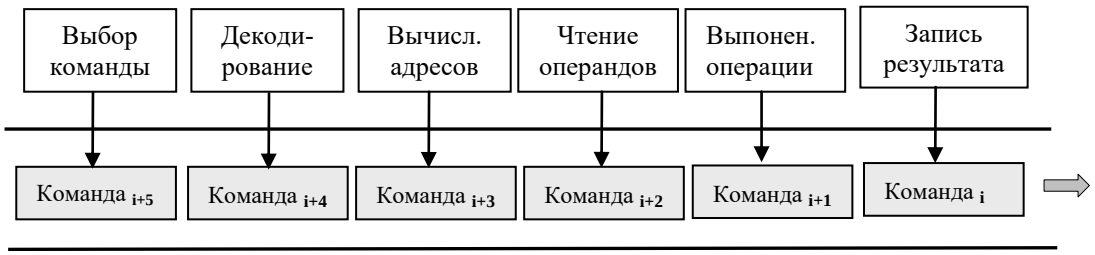


Рис. 14.3. Схема работы конвейера.

Так как выполнение каждой команды разбито на шесть шагов, то одновременно на нашем конвейере может находиться до шести команд программы.[1] Из показанной схемы понятно, почему такие ЭВМ называются конвейерными. Как, например, на конвейере автомобильного завода одновременно находятся несколько машин в разной стадии сборки, так и на конвейере процессора находятся несколько команд в разной стадии выполнения. Надо отметить, что теперь в процессоре уже нет привычного для прежних ЭВМ *регистра команд*, на котором располагается текущая выполняемая команда. Вместо этого, как уже упоминалось, поток выполняемых команд располагается в специальных *буферах команд*, они часто называются *буферами предвыборки* команд.

Сначала следует отметить хорошее свойство такого конвейера: хотя выполнение каждой команды, как в этом примере, занимает шесть шагов, однако *на каждом шаге* с конвейера «сходит» полностью выполненная команда. Таким образом, использование такого рода конвейера позволяет, в принципе, в шесть раз повысить скорость выполнения программы.

Вот теперь, после введения в архитектуру ЭВМ конвейера, наконец, достигнуто *соответствие* скорости работы процессора и памяти. Действительно, если предположить, что каждое из шести устройств на конвейере выполняет свой этап обработки команды за `1 нс.`, тогда каждая команда выполняется на конвейере за `6 нс.`, и за это время процессор успевает произвести все необходимые обмены и командами и данными с памятью (чаще всего из кэша). В то же время, как уже отмечалось, скорость выполнения *потока команд* процессором получается в 6 раз больше за счёт работы конвейера. Поэтому и говорят, что, например, компьютер, работает со скоростью 10^9 операций в секунду, (то есть, как бы выполняет по одной команде каждую наносекунду), хотя Вы теперь знаете, что на самом деле выполнение каждой отдельной команды занимает в несколько раз больше времени.

Разумеется, как всегда, не всё обстоит так хорошо, как кажется с первого взгляда. Первая неприятность будет поджидать, если одна из следующих команд использует результат работы предыдущей команды, а это случается достаточно часто по самой сути алгоритма. Например, пусть есть фрагмент программы:

```
add al,[ebx]
sub X,al
inc ebx
inc edi
```

Для второй команды этого фрагмента нельзя выполнять операцию вычитания, пока первая команда фрагмента не запишет в регистр `AL` свой результат, т.е. выполнится полностью и не сойдёт с конвейера.[2] Таким образом, вторая команды должна быть задержана на третьей позиции конвейера

---

[1] Конвейеры ЭВМ могли разбивать выполнения команд и на большее число шагов. При числе шагов больше шести конвейер часто называли *суперконвейером*. Например, в процессоре Pentium-IV конвейер состоял из 20 более мелких шагов, а в процессоре Prescott уже из 31 шага. Сейчас такие длинные конвейеры не используются, они плохо реагируют на прерывания и команды условных переходов. Устройство современного конвейера описано как дополнительный материал в разд. 14.2.1 в конце этой главы.

[2] Такая зависимость между данными называется в научной литературе `RAW` (Read after Write – чтение после записи, т.е. команда должна отложить чтение своего операнда до его записи одной из предыдущих команд). Существуют и две другие зависимости между данными: `WAR` (Write after Read – запись после чтения) и `WAW` (Write after Write – запись после записи). Постарайтесь понять, что не существует зависимости `RAR` (Read after Read). На современных конвейерах зависимости `WAR` и `WAW` для регистров не блокируют конвейер благодаря использованию так называемых *теневых* регистров (shadow registers), о чем будет говориться далее.

(на четвёртой позиции уже надо читать операнд из регистра AL, а этот операнд ещё не помещён в этот регистр). Вместе со второй командой из нашего примера остановится, конечно, и выполнение следующих за ней команд и на конвейере образуются два «пустых места», которые называют «пузырь» (pipeline bubble) или «ступор» конвейера (pipeline stall).

В таблице 14.1 показана схема движения команд из приведённого выше фрагмента программы на конвейере, каждая строка соответствует одному шагу работы конвейера, «пузыри» закрашены. Таким образом, выполнение этих команд заняло 11 тактов вместо минимально необходимых 9.

В архитектуре ЭВМ принят термин *латентности* (latency – задержки) выполнения машинной команды, это число машинных тактов, которые требуется для полного выполнения команды (включая такты ожидания). Кроме того, существует и понятие *пропускная способность* (throughput) – это количество команд, независимых по данным друг от друга, которые могут одновременно запускаться на исполнение за один такт. Полезным является и понятие *взаимной пропускной способности* (reciprocal throughput), это среднее время между началами выполнения команд, независимых друг от друга по данным.

Таблица 14.1. Движение команд по конвейеру

Выборка	Декодирование	Вычисление адресов	Чтение операндов	Выполнение операции	Запись результата
add al,[ebx]					
sub X,al	**add** al,[ebx]				
inc ebx	**sub** X,al	**add** al,[ebx]			
inc edi	**inc** ebx	**sub** X,al	**add** al,[ebx]		
	inc edi	**inc** ebx	**sub** X,al	**add** al,[ebx]	
	inc edi	**inc** ebx	**sub** X,al		**add** al,[ebx]
	inc edi	**inc** ebx	**sub** X,al		
		inc edi	**inc** ebx	**sub** X,al	
			inc edi	**inc** ebx	**sub** X,al
				inc edi	**inc** ebx
					inc edi

Например, вещественное сложение **fadd** на процессоре Intel Core 2 имеет латентность 3 и пропускную способность 1. Это означает, что для выполнения команды надо 3 такта, но каждый такт можно запускать на выполнение новую такую команду, если она независима по данными от предыдущих команд. А вот целочисленное сложение **add** имеет латентность 1 и взаимную пропускную способность 0.33-0.5, т.е. команда выполняется за один такт, но каждый такт можно запускать на выполнение 2-3 такие команды (в зависимости от наличия свободных исполнительных устройств конвейера). Целое деление **div** является сложной операцией, оно имеет латентность 20-80 (в зависимости от значений операндов, длинные числа делить сложнее, вспомним школьное деление в столбик 🙂). Деление имеет взаимную пропускную способность 20-40, т.е. две команды деления параллельно вообще не выполняются.

Для продвинутых в матане. Обычно команда деления реализуется через команды умножения, эта операция использует таблицы коэффициентов при вычислении интерполяционного многочлена для функции `f(y)=1/y`, тогда `div(x,y)=x*(1/y)`.

Видно, что попадая на исполнительный блок конвейера, который не может выполнить свою работу, команда «тормозится» на нём до тех пор, пока эта часть обработки команды не сможет быть завершена. Ясно, что скорость выполнения всей программы, в которой есть много таких, как говорят, *зависимостей по данным* (иногда говорят конфликтов по данным), может при этом сильно упасть. Зная такую особенность работы, «умный» конвейер, исправляя оплошность программиста, может работать с нарушением исходного порядка выполнения команд в машинной программе (Out of Order Execution). [viii] [см. сноску в конце главы] Тогда, например, получается такой *эквивалентный* фрагмент программы:.

```
add al,[ebx]
inc ebx
inc edi
sub X,al
```

Вот теперь, если построить таблицу состояний конвейера при выполнении этого фрагмента, то легко увидеть, что ячейки конвейера уже не будут пустовать. Современные процессоры могут вести себя весьма «разумно», например, из цепочки команд:

```
mov eax,1
mov eax,2
mov eax,3
```

только последняя будет выполняться на конвейере, а две первые будут пропущены сразу после этапа их выборки в буфер команд и декодирования на мопы. Заметим, что иногда достаточно просто переставить местами две команды и немного изменить их, чтобы убрать задержку, вызванную зависимостью по данным, например:

```
add ebx,4          mov eax,[ebx+4]
mov eax,[ebx]  →   add ebx,4
```

В качестве ещё одного примера рассмотрим вызов процедуры `P(X,Y,Z)` со стандартными соглашениями о связях. Ниже в двух колонках таблице показано два способа такого вызова:

```
push X          sub esp,16
push Y          mov [esp+12],X
push Z          mov [esp+8],Y
call P          mov [esp+4],Z
                mov [esp],offset L
                jmp P
                L:
```

Вызов из левой колонки таблицы создаёт многочисленные зависимости по данным и тормозит конвейер, а вызов из правой колонки лишён этих недостатком.

Задача. Найдите количество тактов работы конвейера для первого и второго способов вызова процедуры (15 и 11).

Отметим, однако, что в языке машины есть и так называемые команды *упорядоченного управления* (или *сериализации*), которые не поступают на конвейер, пока он полностью не освободится, т.е. не выполняться все предыдущие команды (точнее, не выполнятся все предыдущие обмены с памятью).

Это, например, команда **cpuid**, которая выдаёт технические характеристики процессора, команда возврата из прерывания **iret** и команда **rdtscp** (об этой интересной команде будет рассказано позже). Следует отметить и команду **mfence** (memory fence – «забор», «ограда»), она «затормозит» конвейер примерно на 100 циклов, пока не завершатся все предыдущие операции обмена с памятью. Существуют и её более «слабые» формы **lfence** (load fence – ждёт окончания всех чтений) и **sfence** (store fence – ждёт окончания всех записей). Все эти команды по праву считаются самыми длительно выполняющимися командами.

Возможен и другой подход к борьбе с зависимостью по данным. Обнаружив, что следующая команда имеет зависимость по данным с предыдущей, можно прекратить загружать на конвейер команды *данной* задачи и начать загружать команды *другой* готовой к счёту задачи (процесса). Ясно, что между командами *независимых* задачам зависимость по данным невозможна. Таким образом, на конвейере будут обрабатываться команды сразу двух программ. Ясно, что это возможно только в том случае, если в процессоре есть два набора всех регистров (общего назначения, сегментных, `EIP`, управляющих и т.д.) и каждая команда на конвейере знает, с каким именно набором регистров она работает. В этом случае можно сказать, что на одном реальном процессоре (например, с одним конвейером) реализованы два (а бывает, что и четыре) *виртуальных* процессора. Обычно такая технология называется аппаратной или одновременной многопоточностью (Simultaneous Multithreading) или гиперпоточностью (Hyper Threading). Заметим, что в этом случае надо реализовать и два контроллера внутренних прерываний. В персональных компьютерах эта технология была впервые реализована в процессоре Intel Pentium 4 в 2000 году.

14.2.1. Предсказание переходов

Предвидение будущего должно опираться не на предсказания и приметы, а на мудрость.

Марк Туллий Цицерон

Другая неприятность в работе конвейера случается, когда на него поступает команда *условного перехода*, по статистике это происходит через каждые 5-7 команд. Что надо делать после выполнения этой команды, производить переход в другое место программы, или же продолжить последовательное выполнение команд, выяснится только тогда, когда команда условного перехода сойдёт с конвейера. Так, спрашивается, из какой же ветви условного перехода выбирать на конвейер следующую команду? Это конфликт по управлению и обычно при конструировании конвейера для этого случая принимается одно из следующих двух решений.

Во-первых, можно поочерёдно выбирать на конвейер команды из *обеих* ветвей условного перехода. Разумеется, в этом случае половина команд будет выполняться зря и их потом придётся «недоделанными» выбросить с конвейера. В литературе этот метод называется *спекулятивным выполнением* программы. Режущее для российского уха название связано с неудачным выбором значения английского термина speculative: лучше переводить его не как *спекулятивный*, а как *умозрительный*, имея в виду умозрительное, а не настоящее движение алгоритма сразу по двум ветвям условного перехода, что, конечно, противоречит самой сути вычислительного процесса.

Во-вторых, можно выбирать команды из наиболее *вероятной* ветви условного оператора. Например, очевидно, что в нашем компьютере для команды цикла **`loop`** переход на повторение тела цикла значительно более вероятен, чем выход из цикла вниз на следующую команду. В том случае, если выбранная ветвь оказалась неправильной, то все её команды на конвейере помечаются как «поддельные» (bogus instructions) и исключаются из вычислительного процесса.

Надо также сказать, что при использовании конвейера, команды циклов **`loop`** и **`rep`** могут вообще не загружаются на конвейер. Вместо этого электронные схемы обнаружения циклов (loop stream detector) записывают тело цикла в специальный буфер (длиной 20-30 команд), после чего, как говорят, линеаризируют (разворачивают) цикл (loops unrolling), т.е. последовательно подают на конвейер команды цикла (точнее, составляющие их мопы) столько раз, сколько предписывается счётчиком цикла в регистре `ECX`. Таким образом, на конвейере не будет самих команд циклов, которые тоже являются «вредными» командами условного перехода.[1]

Также по командам условного перехода процессор может накапливать для конкретной программы *статистику*, насколько часто та или иная команда осуществляет переход, а не естественное продолжение программы. При первом выполнении команды работает *статическое* предсказание условного перехода: бóльшая вероятность приписывается процессором исполнению следующей команды, а не команды по адресу перехода. Это легко понять, если участь, что в момент анализа команды перехода следующая за ней команда уже загружена (см. таблицу 14.1). Исходя из этого, оптимизирующие компиляторы с языков высокого уровня для оператора **`if then else`** первой располагают в памяти ветвь **`then`**, что может учитывать программист.

Далее включается метод *динамического* предсказанием ветвления (branch prediction). Например, уже процессором UltraSPARC III 2001 года выпуска в так называемом *буфере адресов переходов* BTB (Branch Target Buffer) накапливалась статистика о примерно 16000 последних “сработавших” командах переходов.[2] Вероятность перехода в ранних процессорах кодировалась одним битом («был переход» или «не было перехода»), сейчас вероятность обычно кодируется двумя битами в таблице истории переходов BHT (Brunch History Table): вероятность перехода «очень маленькая» (strongly not taken), «маленькая» (weakly not taken), «большая» (weakly taken), «очень большая» (strongly taken).

---

[1] Не надо путать эту *аппаратную* линеаризацию циклов с аналогичным *программным* разворачиванием циклов, производимым оптимизирующими компиляторами с языков высокого уровня (ну, или хорошим программистом на Ассемблере).

[2] Любопытно, что если условный переход не выполнен (переход «вниз»), то он вообще «не замечается» предсказателем переходов. Поэтому можно спокойно вставлять в свою программу «аварийные» переходы

```
if SomethingBad then goto Error
```

Цепочка последних удачно предсказанных переходов храниться в особом регистре глобальной истории переходов GBHR (global branch history register). При попадании в цикл в этом регистре быстро формируется «истинная» картина переходов, что позволяет предсказывать их почти со 100% вероятностью.

Вероятность удачного предсказания ветвления после небольшого «разгонного» периода счёта программного потока у современных конвейерных ЭВМ может доходить до 99%. Это очень существенно, так как на конвейерах одновременно исполняются около 60 команд, в них по статистике примерно 15-20% условных переходов. Таким образом, предсказатель переходов примерно каждые 3 такта выдаёт вероятный адрес следующего перехода. Неверные предсказания перехода (Branch Misprediction) обрабатываются двумя «обработчиками промахов», это приводит к сбросу буфера предвыборки команд, буфера мопов и перезагрузке конвейера, что может потребовать до 100 процессорных тактов.

Как предсказание ветвления, так и спекулятивное выполнения выдвигают новые требования к архитектуре компьютера. Например, что делать, если команда в какой-либо ветви записывает результат в регистр процессора (или, как ещё говорят, архитектурный) регистр `EAX`, хотя на самом деле эта ветвь выполняться не должна, т.е. содержимое регистра фактически испорчено?

Для решения этой проблемы производится так называемое переименование регистров (register rename), для чего приходится реализовывать особые физические регистры (physical register) или *теневые* регистры (shadow registers). Теневые регистры образуют файл физических регистров (PRF – physical register file). Есть такие файлы для регистров общего назначения (РОН), векторных регистров, особых регистров масок и др. Теперь каждый архитектурный регистр связывается с теневым, и вместо записи в архитектурные регистры, производится запись в такие теневые регистры. И лишь после того, как будет ясно, какая ветвь должна была выполняться, нужный теневый регистр копируется в соответствующий «настоящий» регистр. Любопытно, что есть теневые регистры, которые изначально содержат ноль, так что переименование совмещается с операцией обнуления регистра.

Например, уже в процессоре Pentium-IV было 128 таких теневых регистров, они не привязаны к конкретным *логическим* регистрам РОН (`EAX`, `EBX`, `EIP` и т.д.), а выделяются (назначаются) и освобождаются по мере необходимости.

Теневые регистры являются очень важным ресурсом процессора. В современных процессорных ядрах они могут занимать до половины площади в ядре процессора ⚠. Необходимо также отметить, что алгоритмы для спекулятивного выполнения и предсказания переходов очень сложны, для своей реализации они, без преувеличения, требуют многих десятков миллионов вентилей. Разумеется, такая, как говорят «программа в кремнии» содержит ошибки (аппаратные уязвимости процессора). Например, уязвимости с именами Spectre и Meltdown обнаружены в процессорах Intel в 2017 году, они позволяют программам-злоумышленникам, в обход защиты, читать данные из памяти ядра операционной системы. Это самые серьёзные ошибки, найденные в современных процессорах за последнее десятилетие, они разрушают защиту памяти по чтению. Пока, к сожалению, при устранении этих ошибок (патчи микрокода и ОС) резко возрастают расходы на системные вызовы и падает скорость работы конвейера.

Как видно, команды условного перехода могут сильно замедлить работу конвейера. Как уже говорилось ранее, в современных процессорах x86 есть новые команды, позволяющие в некоторых случаях обойтись без условных переходов (branch free). В качестве простого примера рассмотрим фрагмент программы для вычисления `z:=max(x,y)`, где `x`, `y` и `z` – целые знаковые переменные:

```
    mov eax,x
    cmp eax,y
    jge L
    mov eax,y
L:  mov z,eax
```

С использованием команд условной пересылки этот фрагмент можно переписать в виде:

```
mov   eax,x
cmp   eax,y
cmovl eax,y; if eax<y then mov eax,y
mov   z,eax
```

Как видим, в этом фрагменте команд условного перехода больше нет (точнее, условное присваивание «спрятано» в самой команде **cmovl**), следовательно, нет и проблемы выбора ветви алгоритма, из которой надо читать команды на конвейер.

Перспективным считается и использование новых команд формата SIMD (Single Instruction Multiple Data), в которых одна команда обрабатывает сразу много данных. Эти новые наборы команд SSE (Streaming **S**IMD Extension) и AVX (Advanced Vector Extension) работают с данными на специальных *векторных* регистрах `XMM0-XMM15` (128-разрядных), AVX2 регистрах `YMM0-YMM15` (256-разрядных) и `ZMM0-ZMM31` (512-разрядных), эти регистры находятся в арифметическом сопроцессоре. [ix] [см. сноску в конце главы]

Как и при записи из кэш памяти в ОЗУ, сначала результаты выполнения команд на конвейере записываются в так называемые буфера записи/хранения (write/store buffer), а лишь потом в кэш память. Часто это оказывается очень выгодно, например:

```
add X,1
......
add eax,X
```

Здесь переменная X, попав в буфер отложенной записи конвейера, тут же снова читается из него на регистр, не успевая попасть даже в кэш-память! [1] Этот хитрый приём почему-то называется опережающей записью (store forwarding), он без задержек обрабатывает такие пары команд.

Далее, как уже отмечалось ранее, конвейер весьма болезненно реагирует на прерывания, так как при этом производится автоматическое переключение на другую программу и конвейер приходится полностью очищать от частично выполненных команд предыдущей программы. На больших и супер-ЭВМ для целей более эффективного использования конвейера обработку всех прерываний от внешних устройств обычно поручают одному из каналов ввода/вывода (периферийных процессоров), что позволяет не прерывать работу конвейера *центрального* процессора. Архитектура компьютера с каналами ввода/вывода будет изучаться далее.

Ещё одна неприятность, связанная с функционированием конвейера, подстерегает нас, когда в программе возникает аварийная ситуация (в русскоязычной литературе сокращённо АВОСТ). Это может быть деление на ноль, выполнение привилегированной команды в режиме пользователя, попытка нарушения защиты памяти, неверный код операции и т.д. Заметим, что теперь обработчик прерывания аварийной ситуации часто не сможет *однозначно* определить то место (конкретную команду) в программе пользователя, где возникла эта ситуация.

Действительно, аварийную ситуацию может вызвать любая команда, находящаяся на конвейере, это, естественно, вызывает трудности при отладке программ (появляются, как их называют в специальной литературе, «неопределённые» прерывания, т.е. прерывания с неопределённым местом их возникновения в программе). Заметим, что при выполнении на конвейере условных переходов вообще возможны «ложные» прерывания в той ветви условного перехода, которая на самом деле *не должна* была выполняться. Здесь приходится идти на беспрецедентный шаг и *откладывать* возникновение такого «сомнительного» прерывания до так называемой «отставки» команды, когда будут выполнены все предшествующие ей команды программы.

Ситуация ещё более запутывается, если в компьютере есть несколько конвейеров с общим буфером предвыборки команд, например, один конвейер команды плавающей арифметики, а два других – все остальные команды. Пары команд, которые можно параллельно исполнять на двух конвейерах, называются *спаренными* (pairing), такие команды не должны иметь зависимостей по данным.

В случае с двумя (или более) конвейерами, например, команда, стоящая в некотором фрагменте программы второй, может завершиться раньше, чем команда, стоящая первой. Спрашивается, с какой команды возобновить выполнение программы после возврата из прерывания? Обычно результаты таких команд накапливаются в одном из буферов чтения/хранения, описанных при изучении кэш-памяти, а потом уже в правильном порядке (in order) эти результаты записываются в кэш.

Заметим, что если в процессоре реализована многопоточность, о которой говорилось ранее, то на конвейере одновременно выполняются команды из *разных* независимых вычислительных процессов

---

[1] Требуется, чтобы переменная X не пересекала границу строки кэш памяти, иначе будет задежка на 4-5 тактов.

или разных потоков (нитей) одного процесса. Это позволяет снизить зависимость по данным между соседними командами и уменьшить простои конвейера.

Для *примерной* оценки скорости выполнения некоторого фрагмента машинной программы можно использовать команду без явных операндов **rdtsc** (**R**ea**D** **T**ime **S**tamp **C**ounter, код операции 0F31h), эта команда возвращает в регистровой паре <EDX:EAX> значение системного счётчика тактовых импульсов TSC процессора. Для предотвращения параллельного выполнения конвейером команд, следующих за **rdtsc**, надо сначала поставить команду сериализации **cpuid** (о командах сериализации для конвейера говорилось ранее).

В качестве примера рассмотрим три фрагмента программ на Ассемблере (см. Таблица 14.2), они реализуют оператор Паскаля

```
for ecx:=10000 downto 1 do eax:=eax+ecx
```

Показано среднее число тактов, потраченное на цикл, видно, насколько медленная команда **loop** и как выгоднее «плотнее» загрузить конвейер телом цикла.

Таблица 14.2. Эквивалентные фрагменты программ.

cpuid	**cpuid**	**cpuid**
rdtsc	**rdtsc**	**rdtsc**
mov esi,eax	**mov** esi,eax	**mov** esi,eax
mov ecx,10000	**mov** ecx,10000	**mov** ecx,5000
L:**add** eax,ecx	L:**add** eax,ecx	L:**add** eax,ecx
loop L	**dec** ecx	**add** eax,ecx
cpuid	**jnz** L	**sub** ecx,2
rdtsc	**cpuid**	**jnz** L
sub eax,esi	**rdtsc**	**cpuid**
	sub eax,esi	**rdtsc**
		sub eax,esi
esi ≈ 50200	esi ≈ 10800	esi ≈ 6900

14.3. Конвейер современных процессоров

Не бойтесь поднапрячь мозги! Они от этого не лопнут.

Курт Воннегут. «Колыбель для кошки»

Конвейеры современных процессоров устроены весьма сложно. Например, на процессорах фирмы Intel в разных стадиях выполнения может находиться более 60 команд, за один такт из входного потока декодируется сразу до 4-х команд. Процессоры, обладающие возможностью *за один такт* запускать на выполнения сразу несколько команд, называют суперскалярными (superscalar processor).[1]

> Главная идея организации новых конвейеров состоит в следующем. Раньше обрабатывающие элементы конвейера (порты) располагались вдоль линии этого конвейера, и выполняли свою микрооперацию, когда команды проходило мимо них. Таким образом, микрооперации сами выбирали нужные им порты конвейера, просто проходя мимо них. В новых конвейерах всё обстоит наоборот: все микрооперации поступают в специальный буфер, а каждый порт конвейера, когда освободится, сам берёт из этого буфера готовую к выполнению микрооперацию.

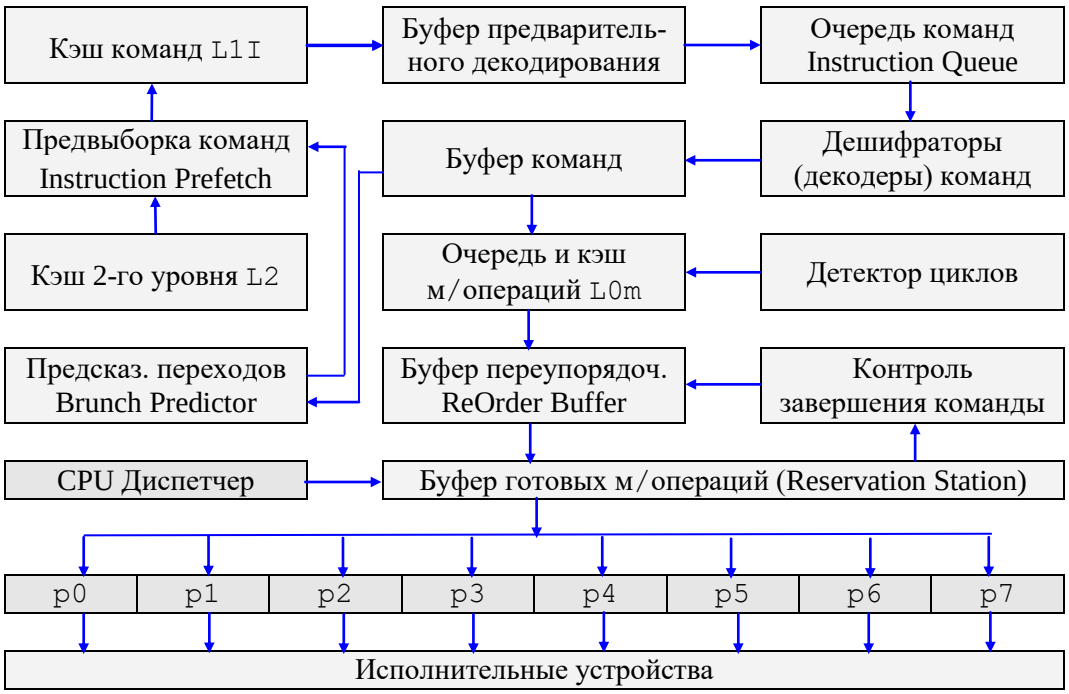
В первом приближении конвейер современного процессора состоит из двух частей. Это головная часть конвейера (Front End) и оконечная часть (Back End). Front End отвечает за выборку команд и данных, декодирование команд и их разбиение на мопы, т.е. полностью готовит команды к выполнению. Back End собственно выполняет мопы и записывает их результаты (см. рис. 14.4).

В головной части конвейера сначала работает блок предвыборки команд (Instruction Prefetch). Просматривая программу вперёд на глубину около 4 Кб по предполагаемому пути её выполнения (с учётом предсказания переходов), производится предвыборка (**prefetch**) команд из оперативной памяти в L1I кэш. При этом работает специальный декодера-«длинномер» (instruction length decoder), определяющий длину каждой команды, эта длина запоминается в специальных дополнительных битах кэша L1I. Современные процессоры декодируют примерно по 4 команды за такт. [x] [см. сноску в

[1] Первым суперскалярным процессором принято считать ЭВМ CDC-6600, выпущенную в 1964 году.

конце главы]. Эта работа производится в «фоновом» режиме, когда шины связи не заняты обслуживанием операций в исполнительных элементах (Execution Unit), подключённым к так называемым *портам* конвейера. Найденные адреса нужных команд и данных помещаются в специальный буфер предварительного декодирования TLB (Translation Lookaside Buffer).[1]

Рис. 14.4. Схема конвейера современных ЭВМ



Затем выбранные команды участками по 32 байта помещаются в очередь команд (Instruction Queue). В этом буфере производится разметка команд: определяются их границы и типы. По типу все команды языка машины подразделяются на простые и сложные, простые декодируются в один-два мопа, а сложные – в три и более. Список простых команд:

1) Пересылки.
2) Сложение, вычитание и логические.
3) **`inc`**, **`dec`**, **`push`**, **`pop`** и **`lea`**.
4) **`jmp`**, **`call`**, **`ret`**, **`jxx`** (только как вторая команда в паре и все **`near`**).

Далее параллельно работают четыре дешифратора (декодера) команд MITE (Micro-Instruction Translation Engine), один для сложных и три для простых команд. Они разлагают каждую команду языка машины на составляющие её мопы, которые являются «внутренним» языком процессора, они реализуют такие простые действия, как пересылка операндов с одного места памяти в другое, целочисленное сложение, сравнения значений на регистрах, некоторые сдвиги, логические операции **`and`**, **`or`** и **`not`** и т.п. Каждый моп, в частности, содержит и ссылку на команду, в состав которой он входит. К сожалению, длина мопа получается большой, примерно 150 бит.[2]

Параллельно с дешифраторами работает предсказатель переходов (Brunch Predictor), управляя предвыборкой команд. Когда в программе встречается условный переход, буфер предвыборки начинает заполняться по новому пути из предсказанной ветви программы.

Итак, за один такт декодеры разлагают на мопы 4-5 команд. Дешифровка простых команд производится аппаратно, т.е. декодер для каждой такой команды «знает», на какие мопы она разлагается. Сложные команды дешифруются в несколько мопов, которые вызывают нужную микроподпрограмму (и передают ей параметры). Все микроподпрограммы хранятся в специальной памяти MSROM (Microcode Sequencer ROM) объёмом несколько килобайт, в этой микроподпрограмме записано, на

---

[1] По запросам буфера TLB команды и данные поступают из кэшей L1I и L1D со скоростью 2x32 байта за такт, эта часть процессора потребляет много энергии и сильно греется.

[2] Формат мопа очень сложный, например, при нехватки собственныйх полей для хранения своих данных, он может брать поля "взаймы" у следующего за ним мопа! Как в русской пословице "Голь на выдумки хитра".

какие мопы разлагается команда (как и во всякой подпрограмме там могут быть условные операторы и циклы из мопов ⚠).

В процессе дешифровки определяются заказы на чтения нужных областей данных, эти заказы отправляются в упомянутый ранее буфер TLB для предвыборки данных в L1D кэш. Оптимальным для работы дешифровщиков является схема следования команд в программе, обозначаемая как 4-1-1, где 4 – «длинная» команда, дешифруемая в 2-4 мопа, а 1 – «короткая» команда, дешифруемая в одну моп. К сожалению, каждый префикс тоже считается командой и дешифруется в один моп.

Таким образом образуется буфер декодированных на мопы команд DIC (Decoded Instruction Cache или Decoded μops Cache), который можно считать кэшем *нулевого* уровня (L0m), там хранятся уже не сами команды, а мопы (на предыдущих процессорах L0m назывался *кэшем трассы* программы (Trace Cache)). Объём кэша L0m составляет около 1-2 тысячи мопов (Кμops, киломопы 🙂 ?). В этом кэше хранятся мопы примерно 1000 последних выполненных команд, вероятность нахождения нужной команды в L0m оценивается в 80%, так что нагрузка на декодеры резко падает. Кроме того, в L0m входит особый буфер цикла (Loop Buffer) примерно на 30-40 команд, выполнение команд тела цикла из этого буфера позволяет стабильно запускать до 6 операций за такт.

Затем мопы из буфера декодированных на мопы команд DIC поступают в буфер переупорядочивания ROB (ReOrder Buffer) на примерно 200 мопов. Далее для каждого мопа назначаются для работы свои физические (теневые, невидимые программисту) регистры-синонимы из регистрового файла (Register Alias Table). Таким образом, один моп может, например, писать в якобы «настоящий» (архитектурный) регистр `EAX`, в то время, как другой моп читает из этого же регистра. Целочисленный регистровый файл содержит порядка двухсот 64-битных регистров, они заранее не привязаны к «настоящим» (архитектурным) регистрам, выделяются и освобождаются по мере необходимости. Вместе с каждым мопом в этом буфере содержаться и номера физических регистров, содержащих данные, необходимые для выполнения этого мопа. Например, рассмотрим фрагмент программы (6 команд):

```
mov eax1,X
add eax1,ebx
mov X,eax1
```

```
mov eax2,Y
sub eax2,21
mov Y,eax2
```

Здесь процессор выделяет для работы теневые регистры EAX_1 и EAX_2 и параллельно выполняет работу с переменными `X` и `Y`. Из этого же регистрового файла при необходимости выделяются и «теневые» регистры `EFLAGS`. Отдельно существует аналогичные регистровые файлы для вещественных и векторных регистров.[1] Здесь надо учесть, что в команде входной и выходной регистр в команде могут обтображаться на разные физические регистры, например

add eax,5; $eax_1 \leftarrow eax$; $eax_2 := eax_1 + 5$;

Как видим, исходный регистр никогда не затирается. Это позволяет свободно использовать внеочередное выполнение мопов, что, в свою очередь, позволяет частично снять зависимость по данным, например, рассмотрим команды (в комментариях их разделение на мопы ❶,❷…):

push eax; ❶$esp_1 := esp-4$; ❷$[esp_1] := eax$

call Fun; ❸$esp_2 := esp_1-4$; ❹$[esp_2] := eip$

Здесь после выполнения мопа ❶ сразу может начаться выполнение двух следующих мопов. Заметим далее, что один моп может по уже упомянутой ранее технологии макрослияния (macro-fusion) кодировать и две простые команды (они называются спаренными, fused), например, **sub** ecx,1 + **jne** L. Спаренный моп как бы расщепляется и выполняется параллельно сразу на двух портах конвейера за один такт. Кроме того, по технологии микрослияния (micro-fusion) два мопа могут сливаться в один, выполняемый на одном порте конвейера. Эти технологии позволяют увеличить пропускную способность конвейера (среднее число команд за такт).

---

[1] Например, в процессоре Intel Core i9-9900KF 180 64-битных регистров в целочисленном регистровом файле и 168 256-битных регистров в векторном регистровом файле (см. главу 17). Значительно больше размер регистрового файла в графических процессорах, например, в NVIDIA GeForce RTX 2080 Ti этот файл имеет объём 256 Кб.

А вот теперь вступает в работу оконечная часть конвейера (Back-end). По мере готовности их операндов, мопы поступают в буфер («станцию») резервации RS (Reservation Station), откуда и посылаются на исполнительные элементы (порты) конвейера. Отметим, что «пустые» команды (**nop**, **excg** ebx,ebx или **xor** rax,rax и т.д.) сразу помечаются как выполненные, они вообще не требуют порты, т.к. выполняются на этапе декодирования и вычисления адресов операндов (требуют ноль «настоящих» мопов).

Выборку мопов для выполнения из буфера резервации производит аппаратный планировщик ядра (CPU Scheduler), учитывая свободные порты и приоритеты мопов для выполнения на конкретном порту конвейера. Сейчас в конвейере 8 портов, пронумерованных от p0 до p7, каждый порт является входным шлюзом к исполнительным устройствам (execution units), к одному порту подключается несколько исполнительных устройств. Например, к порту p1 подключены исполнительные устройства:

1. Целочисленная арифметика (**add**, **sub**, **adc**, **cmp** и т.д.).
2. Вещественная арифметика (**fadd**, **fmul** и т.д.).
3. Целочисленная векторная арифметика (**vpadd**, **vpmul** и т.д.).
4. Логические векторные операции (**vpand**, **vpxor** и т.д.).

Целочисленные операции над скалярными данными могут выполнять четыре порта, три порта могут выполнять целочисленные векторные операции, два порта – вещественные векторные операции и условные переходы.

Два порта (p23) работают на чтение операндов, и два (p47) на запись результатов, остальные специализируются на выполнении мопов. Все четыре порта ввода/вывода умеют вычислять (генерировать) адреса операндов, их называют устройствами генерации адреса AGU (Address Generation Unit), а два из них могут и собственно читать и писать данные. Как будет сказано далее, чтения и запись производится как в кэш L1D, так и в специальные буфера сохранения. Для данных, как и для команд, реализуется предварительная выборка (**prefetch**) в кэш L1D, помещая адреса этих данных в буфер TLB. Блок предвыборки данных может производить как линейное (с последовательными адресами), так и более сложное (с постоянным шагом) предварительное чтение данных в кэш L1D (это, например, позволяет считать *столбец* матрицы).

Простые арифметические и логические операции могут выполнять все четыре “вычислительных” порта (p0156), однако на операции умножения, деления и векторные операции «заточены» только некоторые из них. Естественно, что планировщик старается не загружать специализированные порты простыми мопами. Интересно, что для операции вещественного деления своего порта нет, эта операция выполняется по «хитрому» алгоритму с заменой деления на операции умножения. Фактически, как уже упоминалось, используются таблицы коэффициентов для вычисления интерполяционного многочлена для функции `div(x,y)=x*(1/y)`.

Таким образом, конвейер может запускать на выполнение до 8 мопов за один такт, из них четыре вычислительных. Загруженность конвейера на участке программы (а, следовательно, и эффективность конкретного алгоритма) можно измерять в среднем количестве команд, выполненных за один такт. Например, говорят, что участок программы (конкретный алгоритм) работает со скоростью 3.8 команд/такт (т.е. в среднем за такт запускается на выполнение мопы для 3.8 команд).

Для продвинутых читателей. Эта часть конвейера, по существу, работает по схеме ЭВМ, управляемой потоком данный (Data Flow Computers, см. разд. 15.4).

После исполнения, всех своих мопов, команда помечается в резервации как «вышедшая в отставку» (Retired), она остаётся в резервации, пока в отставку не выйдут все *предшествующие* ей команды программы. Таким образом, несмотря на внеочередное выполнение мопов, сами команды покидают резервацию строго в том же порядка, в каком они находились в исходной программе. Важно отметить, что именно в момент отставки команды происходит прерывание, «виновницей» которого является эта команда. Другими словами, сигнал прерывания, вызванный некоторым мопом, входящим в команду (например, деление на ноль) приходит в момент, когда полностью выполнились все предшествующие команды программы. К этому времени могут оказаться выполненными и несколько следующих команд, которые, в частности, тоже могли вызвать аварийные ситуации. Действия всех таких команд придётся отменить, что весьма непросто 🙁.

Важным является и правильный порядок записи в память (в регистры, L1D кэш или прямо в ОЗУ) результатов выполнения команд (выходных данных). Для этого каждый порт конвейера, одновременно с выполнением следующей операции, посылает результаты предыдущей операции в свой,

так называемый буфер сохранения (Store Buffer). Из буферов сохранения всех портов выходные данные сначала накапливаются в специальном буфере переупорядочивания данных MOB (Memory Order Buffer). Окончательная запись результатов работы команды (из теневых регистров в память или на «настоящий» регистр, включая `EFLAGS`), производится только после отставки этой команды. Таким образом обеспечивается выполнение принципа фон Неймана последовательного выполнения команд программы и алгоритм не теряет свойства детерминированности.

В чём-то это напоминает физический закон сохранения энергии. В соответствии с так называемым соотношением неопределённостей Гейзенберга, энергия должна сохраняться только в среднем за достаточно большой отрезок времени. Внутри же этого отрезка времени закон сохранения энергии не действует, могут порождаться и уничтожаться так называемые виртуальные частицы. По аналогии, принцип фон Неймана последовательного выполнения команд не действует внутри буфера переупорядочивания, где команды (и составляющие их микрооперации) могут выполняться в произвольном порядке.

Работа по отставке команд (Retirement) учитывает и неприятные ситуации, когда в резервации находятся уже выполненные команды из *неправильно* предсказанной ветви условного перехода (эти команды вообще не должны были бы выполняться 🐷). В этом случае производится «откат» (retirement phase) конвейера, когда удаляются все ошибочно выполненные команды (с их мопами) и уничтожаются результаты их работы (это занимает 10-30 тактов процессора).

К сожалению, полностью уничтожить все результаты ошибочно предсказанного условного перехода (спекулятивного выполнения программы) не удаётся, в частности, в кэш памяти остаются загруженные туда ненужные участки программ и данных. Именно на этом основаны аппаратные уязвимости с именами Spectre и Meltdown обнаружены в процессорах Intel в 2017 году, они позволяют программам-злоумышленникам, в обход защиты, читать данные из памяти ядра операционной системы. В 2019 году в процессорах Intel обнаружены новые уязвимости RIDL (Rogue In-Flight Data Load), Fallout и ZombieLoad, которые тоже связаны с плохо организованным спекулятивным выполнением программ. Как видно, алгоритм «отката» конвейера разработан с ошибками, их предстоит исправить в следующих поколениях процессоров, сейчас можно только «залатать» эти ошибки, всё это, однако, приводит к снижению скорости работы процессора на 10-30% 😕.

На этом закончим наше краткое знакомство с архитектурными особенностями современных ЭВМ, ориентированными на повышение производительности их работы, и перейдём к сравнению между собой ЭВМ различной архитектуры.

14.4. ЭВМ различной архитектуры

Страх перед знанием – дело обычное; все мы ему подвержены, и тут ничего не поделаешь. Но каким бы пугающим ни было учение, намного страшнее представить себе человека, у которого нет знания.

Карлос Кастанеда. «Колесо времени»

Сейчас надо отметить одно важное обстоятельство. До сих пор изучалась, в основном, архитектура центральной части компьютера, т.е. процессора и оперативной памяти. При этом практически не рассматривалась архитектура ЭВМ в целом, то есть способы взаимодействия центральной части компьютера с периферийными устройствами, а также способы управления устройствами ввода/вывода со стороны процессора. Такое «однобокое» изучение архитектуры ЭВМ имело свою причину. Дело в том, что, несмотря на большое разнообразие архитектур процессоров современных ЭВМ, различие в этих архитектурах все же значительно меньше, чем в архитектурах компьютеров в целом. Теперь же пора обратить внимание на связь центральной части ЭВМ и её периферийных устройств. Другими словами, надо рассмотреть, как процессор связан с «внешним миром», откуда и как он получает входные данные и выдает результаты счёта программ.

Далее будут рассмотрены две основные архитектуры ЭВМ, которые в каком-то смысле являются противоположными, находятся на разных полюсах организации связи центральной части машины с её периферийными устройствами. Сначала будет изучен способ организации связи между собой устройств компьютера, который получил название архитектуры с *общей шиной*.

14.4.1. Архитектура ЭВМ с общей шиной

Самая серьезная потребность есть потребность познания истины.
Георг Вильгельм Фридрих Гегель

Эта архитектура была разработана, когда появилась необходимость в массовом производстве относительно простых и дешёвых компьютеров (их тогда называли мини- и микро- ЭВМ [9]). Основой архитектуры этого класса ЭВМ была, как можно легко догадаться из названия, *общая шина* (common или unified bus). В первом приближении общую шину можно представить себе как набор электрических проводов (линий, дорожек), снабженных некоторыми электронными схемами. В современных ЭВМ число линий в такой шине обычно несколько сотен. Все устройства компьютера в архитектуре с общей шиной соединяются между собой посредством подключения к такому общему для них набору электрических проводов – шине. По шине передаются адреса, данные и управляющие сигналы (иногда говорят об отдельной шине данных (data bus), шине управления (control bus) и шине адреса (address bus), но обычно они являются частями одной обшей шины).[1] На рис. 14.5 показана схема соединения всех устройств компьютера между собой с помощью такой общей шины.

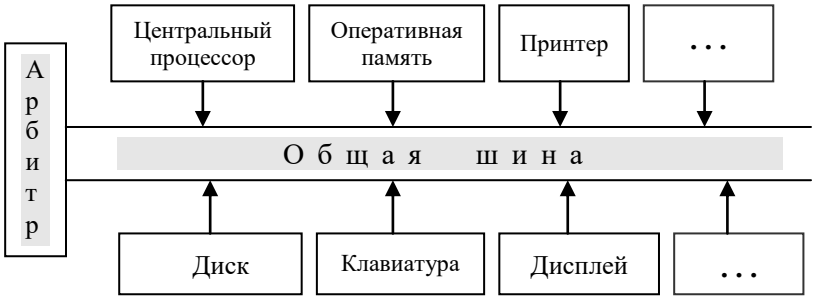


Рис. 14.5. Архитектура компьютера с общей шиной.

В этой архитектуре шина исполняет роль главного элемента, связующей магистрали, по которой производится обмен информацией между всеми остальными устройствами ЭВМ. Легко понять, что, так как обмен информацией производится по шине с помощью электрических сигналов, то в каждый момент времени только *два* устройства могут выполнять такой обмен. Обычно одно из этих устройств является ведущим (инициатором обмена данными), а другое – подчинённым (ведомым). Все устройства компьютера (точнее, их схемы управления – *контроллеры*) подключаются к общей шине посредством специальных электронных схем, которые чаще всего называются *портами* ввода/вывода (не путать с исполнительными портами конвейера).

Каждый такой порт имеет на шине уникальный номер (адрес порта, в рассматриваемой архитектуре это число формата `i16`). Обычно каждому устройству компьютера приписан не один порт, а несколько, так как они, во-первых 8-битные, и для параллельной передачи, скажем, 32 бит требуется четыре порта с последовательными номерами (правда, порядок записи байт в такой составной порт не определён). Во-вторых, порты специализированные: по некоторым портам устройство может читать данные с шины, по другим – записывать (передавать) данные в шину, а есть и универсальные порты, как для чтения, так и для записи.

При использовании общей шины многими устройствами могут возникать конфликты, когда два или более устройств захотят одновременно обмениваться между собой данными. Для разрешения таких конфликтов предназначен арбитр шины (bus arbiter) – специальная электронная схема, которая обычно располагается на одном из концов этой шины. Разрешение конфликтов производится по принципу приоритетов устройств: при конфликте отдается предпочтение устройству с большим приоритетом. В простейшем случае приоритеты устройствам явно не назначаются, а просто считается, что из двух устройств то имеет больший приоритет, которое расположено на шине *ближе* к арбитру (так называемое цепное подключение устройств). Исходя из этого, более «важные» устройства стараются подключить к шине поближе к арбитру. На противоположном от арбитра конце шины обычно

---

[1] В дешёвых *мультиплексных* шинах одни и те же провода могут использоваться сначала для передачи адреса, а затем для передачи данных.

расположено особое устройство, называемое терминатором, оно, как и арбитр, препятствует отражению электрических сигналов и их обратному распространению по шине (чтобы сигналы не «гуляли» по шине туда-сюда).

Разберем теперь общую схему обмена данными между двумя устройствами с помощью общей шины. Сначала *задающее* устройство (инициатор обмена) делает так называемый *запрос шины* (bus request), т.е. посылает арбитру по специальной линии шины сигнал о желании начать обмен данными. Если шина занята, то устройство не получает сигнала о доступности шины и вынуждено ждать её освобождения, а если шина свободна, то устройство производит операцию *захвата шины* (bus mastering) в свое монопольное использование. Это означает, что для остальных устройств арбитр шины теперь не будет выдавать признака доступности.

После захвата шины ведущее устройство определяет, готово ли нужное ему (*подчинённое*) устройство для обмена данными. Для этого ведущее устройство посылает ведомому специальный сигнал и ждет ответа, или же читает из порта ведомого устройства его *флаг готовности*. Определив готовность ведомого устройства, ведущее устройство начинает обмен данными. Каждая порция данных (байт, слово или двойное слово) снабжается номером порта устройства-получателя. Вообще говоря, можно посылать данные по шине и без уверенности в готовности адресата принять эти данные, тогда это похоже просто на посылание адресату письма́ или телеграммы.

Окончив обмен данными, ведущее устройство производит *освобождение шины* (снимает так называемый сигнал занятости DBSY#). На этом операция обмена данными между двумя устройствами по общей шине считается завершенной. Разумеется, арбитр следит, чтобы ни одно из устройств не захватывало шину на длительное время (например, устройство может сломаться, и оно поэтому «забудет» освободить шину, выведя весь компьютер из строя). Все описанные выше операции с общей шиной (запрос, захват и т.д.) производятся по строгим правилам, эти правила называются *протоколом* работы с общей шиной. Обычно все действия, связанные с обменом по шине одной порцией данных, называются циклом шины (говорят о циклах чтения и записи в память, цикле передачи сигнала прерывания и т.д.). Такова в простейшем изложении схема обмена данными по общей шине.

Рассмотрим теперь, как видит общую шину компьютера программист на Ассемблере. Как уже было сказано, у каждого периферийного устройства обязательно есть один или несколько портов с номерами, закрепленными за этим устройством. Процессор может обмениваться с портами байтами, словами или двойными словами (в зависимости от вида порта). Для записи значения в некоторый порт используется (привилегированная) машинная команда

```
out op1,op2
```

Здесь операнд `op1` определяет номер нужного порта и может иметь формат `i8` (если номер порта небольшой и известен заранее) или быть регистром `DX` (если номер больше 255 или становится известным только в процессе счёта программы).

Номера портов от `00h` до `0FFh` обычно используются оборудованием материнской платы (таймер, контроллер прерываний и т.д.). Например, традиционно порты `40h-5Fh` принадлежат таймерам, `60h-6Fh` – клавиатуре и динамику, `80h-9Fh` – сопроцессору, `378h-37Fh` – LPT1 (принтеру) и т.д. Порт `70h` связан с контроллером прерываний (содержит триггер маски прерываний), с его помощью можно, скажем, закрыть (и открыть) внешнее «немаскируемое» прерывание №2:

```
in  AL,70h; AL:=<70h порт>
or  AL,80h; AL[7]:=1
out 70h,AL; закрыть прерывание №2
```

Второй операнд `op2` должен задаваться регистром `AL` (если производится запись в порт байта), `AX` (2 байта), `EAX` (4 байта), `RAX` (8 байт). При выполнении такой команды значение регистра посылается по общей шине в соответствующий порт. Все порты однобайтные (это ещё одно байтовое *адресное пространство* «памяти портов»).[1] При записи двух байт из регистра `AX` берутся, как и в обычной памяти, два соседних порта, номер первого из которых должен быть чётный и задаётся в команде **`out`**, для регистра `EAX` используются уже четыре однобайтных порта с последовательными

[1] На общей шине (в части адреса) есть специальная линия (1 бит), который и задаёт, обращается ли команда в основную память или порт ввода/вывода.

номерами, первый из которых кратен четырём и т.д. Таким образом в отличие от оперативной памяти, в «памяти портов» требуется *выравнивание* адреса порта, с другой стороны, данные в портах представлены в прямом (big endian), а не в перевёрнутом виде, например, команды

```
mov ax,1234h
out 20,ax
```

пошлют в порт 20 число `12h`, а в порт 11 число `34h`.

Для чтения данных из порта в регистр процессора служит команда

```
in op1,op2
```

Здесь уже *второй* операнд `op2` определяет номер нужного порта и может иметь, как и в предыдущей команде, формат `i8` или быть регистром `DX`. Первый операнд `op1` должен быть регистром `AL`, `AX` или `EAX` (для чтения байта, слова или двойного слова). При выполнении этой команды нужное значение читается из заданного порта и по общей шине поступает в процессор на указанный регистр. Позже будет рассмотрен небольшой пример с использованием этих команд.

В старших моделях добавлены также новые цепочечные (строковые) команды **`ins`** и **`outs`** для обмена данными между портами с последовательными номерами и областью (массивом) оперативной памяти. Эти команды не имеют явных операндов и, по аналогии со строковыми командами, имеет несколько модификаций: **`insb`**, **`insw`**, **`insd`** для чтения одного, двух и четырёх байтов соответственно. Адрес порта всегда задаётся в регистре `DX`, а адрес в памяти, как обычно для строковых команд, определяется регистровой парой `<ES:EDI>` при чтении из порта в память, и `<DS:ESI>` при записи из памяти в порт. Флаг `DF` имеет обычное назначение направления для строковых команд. Возможно использование префикса повторения **`rep`**.

Для доступа к портам существует также механизм *отображения* некоторого диапазона портов на ячейки обычной оперативной памяти (memory-mapped I/O), после выполнения этого отображения можно обмениваться данными с портами обычной командой **`mov`**. Необходимо только учитывать, что кэширование при этом отключается. Любопытно, что в некоторых архитектурах, например, в процессорах ARM, порты ввода/вывода всегда отображены на определённый диапазон оперативной памяти.

Команды **`in`** и **`out`** являются «тяжёлыми» для конвейера процессора, они ждут завершения всех предыдущих команд, а следующие за ними команды начинают выполняться только после полного завершения ввода/вывода (это уже упоминавшиеся так называемые команды *сериализации*).

Команды **`in`** и **`out`**, наряду с командами открытия и запрета внешних прерываний **`sti`** и **`cli`**, являются привилегированными, но доступ к ним определяется не текущим уровнем привилегий процессора `CPL` в кодовом сегментном регистре `cs`, а дополнительным, так называемым уровнем привилегий команд ввода/вывода. Этот уровень прописан в поле `IOPL` (I/O Privilege Level – уровень привилегий ввода-вывода) в 12-ом и 13-ом битах регистра `EFLAGS`. Таким образом, как уже говорилось, команды ввода/вывода могут быть разрешены для выполнения программой пользователя, в то время как остальные привилегированные команды запрещены.

В старших моделях семейства можно закрывать и открывать для непривилегированной программы доступ к каждому порту по отдельности. Для этого служит так называемая битовая карта разрешения ввода-вывода (I/O Permission Bit Map), в которой для каждого порта ввода/вывода существует свой бит защиты. Если некоторый бит этой карты сброшен (равен 0), то непривилегированный доступ к соответствующему порту разрешен, иначе запрещён (для доступа к составным портам размеров 2 или 4 байт все соответствующие им биты должны быть нулевыми). Размер этой битовой карты может составлять до 64 Кбит = 8 Кб, обычно она хранится в дополнительной части контекста задачи `TSS`. Можно «ухитриться» задать одну такую карту для нескольких (даже независимых) вычислительных потоков.

Итак, на нашем компьютере есть только очень простые команды для обмена байтом, словом или двойным словом (для 64-битного режима возможен и формат учетверённых слов) между регистром процессора и портом внешнего устройства. Вспомним, что на изученной ранее учебной ЭВМ УМ-3 в нашем распоряжении были очень удобные для программиста команды ввода/вывода *массивов* целых или вещественных чисел. Почему же в архитектуре нашего компьютера команды ввода/вывода такие примитивные и неудобные для программиста?

Ответ на этот вопрос легко понять, если вспомнить, что общая шина связывает между собой очень разные устройства, для которых было необходимо найти общий и приемлемый для всех их формат передаваемых данных. Ясно, что таким форматом может быть только один или несколько байт – те минимальные порции данных, с которыми оперирует процессор. Вот и приходится, например, для ввода целого числа с помощью команд **`in`** и **`out`** выполнять достаточно сложную программу. Контроллер каждого периферийного устройства преобразует простой язык команд **`in`** и **`out`** в более сложные команды управления этим устройством (например, подвести головки чтения/записи к заданному цилиндру диска).

Уяснить для себя способ взаимодействия программы на Ассемблере и «внешнего мира» с помощью общей шины можно на таком образном примере. Программа ведёт свое «существование» во внутренней части ЭВМ (в оперативной памяти и процессоре), и не может покидать этой своей «резиденции». Для связи с «внешним миром» у нашей программы имеется только одна возможность – это порты, которые можно рассматривать как своеобразные «почтовые ящики» на внутренней стороне двери «резиденции» программы. В некоторые из этих ящиков-портов выполняемая программа может бросать свои короткие «телеграммы» для внешних устройств, из других ящиков можно доставать «телеграммы», приходящие от внешних устройств (длина каждой «телеграммы» 1, 2 или 4 байта). Понятно, что для того, чтобы получить достаточно большой объем данных (например, строку текста с клавиатуры), программа должна обменяться с устройством большим числом таких «телеграмм». Можно сказать, что у программы довольно скучная жизнь в её «резиденции», никаких тебе газет и журналов, и тем более радио и телевидения, для общения с «внешним миром» одна только скупая телеграфная связь 🙁. Отметим также, что никто не гарантирует, что «телеграммы» на самом деле получена адресатом (пока он не ответил).

Отметим, что и в сети Интернет существуют два основных протокола связи по обмену данными так называемого транспортного уровня. Один из них (TCP) предварительно устанавливает соединение между поставщиком и получателем, это гарантирует, что поставщик, как при телефонном разговоре, «слышит» что получатель на месте. Другой протокол (UDP), по аналогии с почтовой службой, просто посылает сообщение, рассчитывая, что адресат (возможно) пришлёт потом «квитанцию» о получении.

Теперь будет полезно рассмотреть общую архитектуру связи процессора и периферийных устройств с точки зрения пользователей разного уровня. Рассмотрим, как обстоит дело на внешнем, концептуальном, внутреннем и инженерном уровне ви́дения архитектуры ЭВМ.

- Уровень конечного пользователя. Пользователь-непрограммист бухгалтер Иванов твёрдо уверен, что в его компьютере есть команда «Распечатать ведомость на зарплату», так как именно это происходит каждый раз, когда он нажимает на кнопку меню «Печать ведомости». Он также знает, что в его распоряжении имеются также такие удобные команды его компьютера, как «Подведение баланса», «Проводка счёта» и другие, непонятные неосведомленным лицам операции. Можно считать, что Иванов работает на некоторой виртуальной машине, снабженной такими очень удобными для него командами, с помощью которых его бухгалтерская программа и общается с «внешним миром». Ввод данных при этом производится в терминах профессиональной деятельности пользователя-непрограммиста, например, путем заполнения всевозможных форм и бланков.
- Уровень прикладного программиста. Программист Петров, который и написал бухгалтерскую программу (скажем, на Паскале 🙂), только улыбнется наивности бухгалтера Иванова. Уж он точно знает, что даже для того, чтобы вывести только один, например, символ `'A'`, надо написать оператор стандартной процедуры `Write('A')`. Для Петрова общение его программы с «внешним миром» заключается в работе с потоками ввода/вывода при помощи стандартных процедур, доступных в его языке программирования высокого уровня. Правда, Петрову известно, что *на самом деле* его программа сначала переводится (транслируется) с Паскаля на машинный язык, а лишь потом выполняется на компьютере. Поэтому, зная, что язык Ассемблера очень близок к машинному языку, он из любопытства поинтересовался у программиста на Ассемблере Сидорова, что тот напишет в своей программе, чтобы вывести символ `'A'`. Сидоров, немного подумав, ответил, что обычно для этой цели он пишет предложение Ассемблера **`outchar`** `'A'`. Разница между этими двумя спо-

собами вывода символа в Паскале и в Ассемблере показалась Петрову несущественной.[1] Например, как специалист в программировании, он знает, что в языке C для этой же цели надо вызвать библиотечную функцию `printf("%c",'A');`. Как видим, на языке C вызов самый некрасивый 🙁.

- Уровень операционной системы. Программист на Ассемблере Сидоров, однако, знает, что предложение Ассемблера **`outchar 'A'`** является на самом деле не командой машины, а *макрокомандой*, на её место Макропроцессор подставит на этапе компиляции макрорасширение, например, такого вида

```
.data
buf  db 0,0
.code
  mov buf,'A'
  invoke StdOut,offset buf
```

  Вот этот, как говорят, *системный вызов* с именем `StdOut` и будет, с точки зрения Сидорова, выводить символ `'A'` на *стандартное устройство вывода*, которое может быть, в частности, как экраном, так и печатающим устройством, текстовым файлом и т.д. Таким образом, общение с «внешним миром» представляется для Сидорова множеством системных вызовов в его программе на Ассемблере. Обработка системных вызовов производится операционной системой, поэтому этот уровень в литературе по архитектуре ЭВМ обычно и называют *уровнем операционной системы*.

- Уровень языка машины. Системный программист Антонов, однако, снисходительно пояснит Сидорову, что его системный вызов `StdOut` в конце концов выходит, например, на команду **`int`** `2Eh`, т.е. вызывает служебную процедуру-обработчик прерывания с номером `2Eh`.[2] А уж эта процедура и произведет *на самом деле* вывод символа, используя, в частности, специальные команды языка машины для обмена с внешними устройствами **`in`** и **`out`**. Именно эти машинные команды и будут, с точки зрения Антонова, используя порты ввода/вывода, производить вывод символа с регистра процессора `AL` уже на *вполне определенное* периферийное устройство. Таким образом, Антонов знает, к какому конкретному устройству в данный момент подключен стандартный поток вывода (экрану, принтеру, файлу т.д.). При этом используются привилегированные команды языка машины. Этот уровень в литературе по архитектуре ЭВМ обычно называют уровнем машинных команд.

- Инженерный уровень. Инженер-электронщик Попов, внимательно прослушав разговор пользователей, скажет, что все это неверно. *На самом деле* процессор выводит символ на экран или печатающее устройство путем сложной многоступенчатой последовательности действий с общей шиной, используя специальный протокол работы. Эти действия включают в себя такие операции с общей шиной, как запрос, захват, передача данных и освобождение этой шины (цикл шины). И только после этого символ, наконец, прибывает с регистра процессора `AL` по назначению на регистр печатающего устройства. Таким образом, команды языка машины **`in`** и **`out`** с точки зрения Попова, не вводят и выводят данные, а просто инициируют циклы шины, посредством чего и производится обмен данными между процессором и периферийными устройствами.

Как Вы догадываетесь, нельзя сказать, кто же из этих людей прав, и, как это часто бывает в современной науке, бессмысленно спрашивать, как всё происходит «на самом деле». Каждый из них прав со своего уровня ви́дения архитектуры компьютера. И, как уже говорилось, опускаться на более низкий уровень рассмотрения архитектуры следует только тогда, когда это абсолютно необходимо для дела. В этом смысле точка зрения бухгалтера Иванова для пользователей-непрограммистов ничем не хуже, чем у системного программиста Антонова.

---

[1] Это очень яркий пример того, как макросредства *повышают* уровень языка Ассемблера: макрокоманда вывода символа в языке *низкого* уровня оказывается для Петрова по внешнему виду (если отвлечься от деталей синтаксиса), очень похожа на соответствующие средства языков *высокого* уровня.

[2] Как уже упоминалось, в современных ОС вместо команды **`int`** `2Eh` используется новая команда **`sysenter`**, она вызывает системную функцию более быстро (без прерывания).

Разберем теперь очень простой пример реализации операции ввода/вывода на уровне системного программиста. Оставим в стороне пользователя-непрограммиста (он нам сейчас неинтересен) и рассмотрим, например, операцию перемещения курсора на экране компьютера в позицию с координатами `(X,Y)`.

Для прикладного программиста, как Вы знаете, для этой цели надо выполнить, например, оператор процедуры языка Free Pascal `GotoXY(X,Y)` из модуля с именем `Crt`. На уровне же операционной системы для позицирования курсора можно, например, выполнить системный вызов с именем `SetConsoleCursorPosition`.

В качестве упрощённого примера рассмотрим, как оператор `GotoXY(X,Y)` обрабатывался в старых ЭВМ с простым видеоконтроллером, управляющим выводом данных на текстовый, так называемый VGA дисплей. Этот оператор преобразовывался в системный вызов **`int`** `10h` для вызова соответствующей процедуры-обработчика программного прерывания:[1]

```
; gotoXY(X,Y)
  mov ah,2
  mov bl,0
  mov dl,X
  mov dh,Y
  int 10h
```

Как видим, координаты `X` и `Y` для позицирования курсора передаются в регистрах `DL` и `DH`. Системный вызов **`int`** `10h` может выполнять различные операции с экраном компьютера, в зависимости от своих параметров, передаваемых ему на регистрах. Рассмотрим (в сильно упрощенном виде) тот фрагмент процедуры-обработчика этого системного вызова, который выполняет запрос на позицирование курсора.

Во-первых, нам необходимо понять, а как вообще дисплей (точнее, электронная схема – *контроллер* дисплея) «знает», куда необходимо в каждый момент времени поставить курсор на экране. Оказывается, что у контроллера дисплея, как, впрочем, и у любого другого периферийного устройства, есть свои *регистры*. Нас будут интересовать регистры дисплея с десятичными номерами 14 и 15 (обозначим их `R14` и `R15`), каждый из них имеет размер 8 бит, но их совокупность, как это часто бывает, является регистровой парой и может хранить 16-битное целое число, как показано ниже

16 бит

R14	R15

Далее, дисплей «считает», что его экран имеет не 25 строк и 80 столбцов,[2] как думают обычные прикладные программисты, а 25*80 *знакомест*, в каждое из которых можно вывести один символ и поставить курсор. Знакоместа́ в первой строке экрана нумеруются не от 1 до 80, как считает, например, прикладной программист, а от 0 до 79, во второй строке – от 80 до 159 и т.д. Другими словами, все позиции экрана рассматриваются как одномерный массив, пронумерованный, начиная с нуля. Так вот, чтобы курсор переместился в нужную нам позицию `(X,Y)` в пару регистров `<R14:R15>` необходимо записать число

```
80*(Y-1)+(X-1)
```

Следовательно, сначала процедуре-обработчику прерывания необходимо вычислить это число, используя параметры `X` и `Y` из системного вызова. Освоив язык Ассемблера, Вы уже знаете, что это можно сделать, например, такими командами:

```
mov al,80
dec dh; Y-1
```

---

[1] Современные компьютеры оснащаются сложными графическими картами, которые по существу являются специализированными процессорами вывода, программирование операций ввода/вывода для них значительно сложнее. В то же время, и самые «навороченные»" графические процессоры умеют работать в режиме VGA, более того, при включении электрического питания они, как правило, начинают работу именно в этом режиме.

[2] Рассматривается, естественно, работа дисплея только в стандартном VGA текстовом (консольном) режиме, когда на экране располагается 25 строк по 80 символов в каждой строке. Для программ, работающих в текстовом (консольном) режиме, операционная система эмулирует такую работу в оконном режиме.

```
mul dh; ax:=80*(Y-1)
dec dl; X-1
add al,dh
adc ah,0; ax:=80*(Y-1)+(X-1)
mov bx,ax; Спасём ax на bx
```

Теперь необходимо переслать содержимое регистров BL и BH соответственно в регистры R15 и R14 нашего дисплея. Для этого будут использоваться два порта дисплея (в каждый можно записывать для передачи дисплею операнд размером в байт). Порт с шестнадцатеричным номером 3D4h позволяет выбрать номер регистра дисплея, в который будет производиться очередная запись данных. Для этого в этот «порт выбора регистра» необходимо записать номер соответствующего регистра (в этом примере это десятичные номера 14 и 15). После выбора номера регистра назначения, запись в этот выбранный регистр производится посредством посылки байта в «транспортировочный» порт дисплея с номером 3D5h. В итоге получается следующий фрагмент программы:

```
mov dx,3D4h; Порт выбора № регистра
mov al,15
out dx,al;   Выбираем R15
inc dx;      Порт записи в регистр 3D5h
mov al,bl;   Младший байт из bx
out dx,al;   Запись в R15
dec dx;      Порт выбора № регистра
mov al,14
out dx,al;   Выбираем R14
inc dx;      Порт записи в регистр
mov al,bh;   Старший байт из bx
out dx,al;   R14:=bh
```

Ясно, что здесь напрашивается написание макрокоманды

```
MovReg macro reg,value; reg:=value
  mov dx,3D4h;   Порт выбора № регистра
  mov al,reg
  out dx,al;     Выбираем reg
  inc dx;        Порт записи в регистр 3D5h
  mov al,value;  Байт для записи в регистр
  out dx,al;     Запись value
       endm
```

Тогда наш фрагмент программы перепишется как

```
MovReg 15,bl
MovReg 14,bh
```

Вот теперь курсор будет установлен в нужное место экрана, и можно возвращаться на команду, следующую за системным вызовом. Разумеется, этот фрагмент драйвера весьма примитивен. Например, если после записи в 15-й регистр дисплея и до записи в 14-й регистр произойдёт прерывание, то курсор прыгнет в непредсказуемое место экрана, так что по-хорошему надо было бы на время работы нашего фрагмента закрыть прерывания от внешних устройств или, лучше, *заблокировать* для контроллера чтение данных из регистров дисплея. Это, разумеется, делается записью некоторого значения в определённый регистр дисплея, для чего понадобятся и другие команды **in** и **out**. Кроме того, хорошо бы предварительно убедиться, что дисплей вообще включен и работает в текстовом режиме (в котором именно 80, а не, скажем, 132 колонки), для чего потребуется, например, считать из определенного регистра дисплея некоторые его *флаги состояния*.

Из этого примера видно, как тяжело работать с портами ввода/вывода при помощи команд **in** и **out**. Как уже говорилось, для исправления этой ситуации есть возможность отобразить адресное пространство портов на участок оперативной памяти. После этого можно работать с портами с использованием «обычной» команды **mov**, что удобно для программиста. Необходимо, однако, понять, что эта особая команда **mov**, при её выполнении процессор сам инициирует цепочку действий по об-

мену данных с нужными портами по протоколу шины, т.е. это просто «замаскиро́ванные» команды **`in`** и **`out`** (фактически, это аппаратная макрокоманда!).

Вот, примерно так и видит общение программы с «внешним миром» системный программист на компьютере с общей шиной. Надеюсь, что этот простенький и сильно упрощенный фрагмент реализации системного вызова для старых ЭВМ не отобьёт у Вас охоту стать системным программистом и заниматься написанием драйверов периферийных устройств 🙂.

14.4.2. Достоинства и недостатки архитектуры с общей шиной

Оптимист ищет достоинства в недостатках, а пессимист – недостатки в достоинствах.

Валерий Афонченко, афорист 🙄

Из рассмотренной схемы связи всех устройств компьютера с помощью общей шины легко увидеть как достоинства, так и недостатки этой архитектуры. Её несомненным достоинством является ее простота и возможность легкого подключения к шине новых устройств. Для подключения нового устройства необходимо оборудовать его соответствующими портами, присвоив им свободные номера, благо этих номеров много (2^{16}).[1] Далее, конечно, системный программист (обычно работающий на заводе-изготовителе данного внешнего устройства), должен написать программу-драйвер, которая предоставляется пользователям вместе с устройством. При работе компьютера этот драйвер будет процедурой-обработчиком событий, связанных с этим устройством.

Главный недостаток этой архитектуры тоже очевиден: пока два устройства обмениваются данными, остальные не могут этим заниматься и должны простаивать (говорят, что они пропускают циклы шины). Можно сказать, что компьютер в какие-то периоды времени вынужден соизмерять скорость своей работы со скоростью *самого медленного* устройства на общей шине. Этот недостаток давно осознан конструкторами ЭВМ и с ним пытаются бороться, создавая дополнительные шины. Между собой эти шины при необходимости соединяются электронными устройствами, которые называются *мостами*.

Например, персональные ЭВМ имеют отдельную «главную», так называемую системную или *процессорную шину* FSB (Front Side Bus), в последних процессорах она обычно называется по-другому. Обычно к этой системной шине подключены самые быстрые устройства ЭВМ (процессоры, оперативная память, кэш второго уровня и др.), именно процессорная шина определяет производительность компьютера. Эта шина обычно работает на частоте в несколько раз меньшей, чем тактовая частота самого процессора. Подсоединённая к системной шине кэш-память L3 работает на частоте этой шины, а вот частота работы основной DRAM памяти в несколько раз меньше.

С ростом количества процессорных ядер, объёма кэш памяти, видео ускорителей и контроллеров сопряжения с другими шинами, пропускной способности этой главной процессорной шины стало не хватать. В современных ЭВМ высокоскоростные устройства стали соединять между собой с помощью специальной коммуникационной решётки (матрицы), иногда её называют «коммуникационной фабрикой» (Communication Fabrics).

В процессорах фирмы AMD контроллер оперативной памяти обычно встраивается прямо в процессорное ядро и связывается с оперативной памятью с помощью отдельной высокоскоростной шины.

К процессорной шине через главный, так называемый северный мост (Northbridge) подключалась вспомогательная шина для обмена данными с высокоскоростными внешними устройствами (контроллер оперативной памяти, диски, видео карта, сетевая карта и т.д.). К этой шине средней производительности (шине расширения) через ещё один, южный мост (Southbridge), подключается общая шина для работы с медленными периферийными устройствами (контроллер внешних прерываний, клавиатура, мышь, принтер и т.д.). Весь этот набор шин и мостов определяет логическую организацию материнской платы компьютера и обычно называется чипсет (chipset). Как видно, по аналогии с иерархической организацией памяти ЭВМ (регистры, кэш, оперативная память, внешняя память), так же иерархически устроена и шинная организация.

---

[1] Обычно устройство делается так, что оно может работать с любым диапазоном портов, этот диапазон выделяется ему операционной системой при подключении устройства к ЭВМ.

Инженеры-схемотехники когда-то давно присвоили северному и южному мосту такие имена, чтобы ориентироваться по месту их расположения на чертеже материнской платы (северный мост был сверху). Современные архитектуры включают схему северного моста прямо внутрь микросхемы процессора, так что этот термин сейчас уже не используется.

Ясно, однако, что невозможно соединить своими шинами все возможные пары устройств, это просто экономически нецелесообразно, не говоря уже о том, что такую архитектуру очень трудно реализовать.

Здесь можно ещё упомянуть и о трудностях определения состава оборудования, подключенного к общей шине. Действительно, как может процессор (точнее, программа операционной системы) определить, какие устройства подключены в данный момент к общей шине? Практически единственная возможность – это посылать сообщения ко всем возможным портам (а их 2^{16}) в надежде, что некоторые из них «откликнутся», и можно будет прочитать из них тип и характеристики данного устройства. Те пользователи, которым приходилось устанавливать на свой компьютер операционную систему (например, Windows), могли заметить, что, когда дело доходило до определения состава подключенного к компьютеру оборудования, программа установки надолго «задумывалась». Частичное решение этой проблемы предоставляют периферийные устройства типа Plug and Play (хорошего перевода нет, что-то вроде «Вставь и сразу пользуйся»). Эти устройства при включении компьютера сами уведомляют операционную систему о своем присутствии и характеристиках, посылая по общей шине соответствующие сигналы прерывания.

Исходя из таких очевидных недостатков архитектуры с общей шиной, видна и необходимость другой архитектуры связи устройств компьютера между собой, которая была бы лишена этих недостатков. Обычно в литературе она называется архитектурой с каналами ввода/вывода [1,3].[1]

14.4.3. Архитектура ЭВМ с каналами ввода/вывода

Если вы точно назовёте, чего не может делать машина, я всегда смогу создать машину, которая сможет именно это.
Джон фон Нейман

Архитектура ЭВМ с каналами ввода/вода (input-output channels) предполагает возможность параллельной работы нескольких устройств компьютера. Сначала надо понять, какие же работы при общении с внешними устройствами нам надо производить параллельно. Оказывается, что в основном нужно обеспечить параллельный обмен данными нескольких устройств с оперативной памятью (такая оперативная память называется *многовходовой*). Действительно, когда рассматривался мультипрограммный режим работы ЭВМ, говорилось, что для эффективного использования аппаратных ресурсов необходимо обеспечить как можно более полную загрузку всех устройств компьютера.

Например, одна программа может выполнять свои команды на процессоре, другая – читать массив данных с диска в оперативную память, третья – выводить результаты работы из оперативной памяти на печать и т.д. Как видим, здесь оперативная память должна параллельно работать с несколькими устройствами: процессором (он читает из памяти команды и данные, а записывает в память результат выполнения некоторых команд), диском, печатающим устройством и т.д. Скорость работы оперативной памяти должна быть достаточна для такого параллельного обслуживания нескольких устройств (здесь, как уже говорилось, сильно помогает расслоение оперативной памяти и использование вспомогательной памяти типа кэш).

Отсюда ясно, что свой кэш может использоваться как буфер между оперативной памятью и внешними устройствами. Например, обмен между оперативной памятью и диском производиться через дисковый кэш, выполняющий те же функции, что и кэш между оперативной и регистровой памятью, свой кэш имеет принтер и т.д.

Как известно, процессор выполняет обращения к оперативной памяти, подчиняясь командам выполняемой программы. Ясно, что и все другие обмены данными с оперативной памятью в этой новой для нас архитектуре должны выполняться под управлением достаточно «интеллектуальных» уст-

[1] Эта архитектура первоначально использовалась на больших ЭВМ 3-го поколения и возникла раньше архитектуры с общей шиной, последняя, как уже говорилось, была разработана для дешевых ЭВМ массового производства.

ройств ЭВМ. Вот эти устройства и называются *каналами ввода/вывода*, так как они управляют обменом данными между оперативной памятью и, как говорят, периферией. По существу, канал ввода/вывода является *специализированным процессором* со своей системой команд (своим машинным языком). Машинные языки каналов ввода/вывода обычно проще машинного языка процессора, так как они предназначены только для узкой задачи описания алгоритмов обмена данными между компьютером и «внешним миром».[1]

В современной литературе по архитектуре ЭВМ у термина «канал ввода/вывода» есть много синонимов. Часто их называют процессорами ввода/вывода или периферийными процессорами (смысл этих названий легко понять из назначения данных устройств). Наиболее «навороченные» каналы называют иногда *машинами переднего плана* (front-end computers). Здесь имеется в виду, что все внешние устройства, а, следовательно, и пользователи, могут общаться с центральной частью компьютера (обычно это супер-ЭВМ) только через эти машины переднего плана. Кроме того, эти машины могут разгрузить процессор, взяв на себя, например, обработку прерываний от *внешних* устройств, весь диалог с пользователями, компиляцию программ в объектный код, получение загрузочного модуля и т.д. Процессоры супер-ЭВМ при этом будет выполнять только свою основную работу – быстрый счёт уже подготовленных для них загрузочных модулей.

Чаще всего на больших компьютерах есть несколько каналов ввода/вывода, так как эти каналы выгоднее делать *специализированными*. Например, на одной из первых ЭВМ с такой архитектурой CDC 6600 было 10 каналов ввода/вывода. Обычно один канал ввода/вывода успевает обслуживать все медленные внешние устройства (клавиатура, печать, дисплеи, линии связи и т.д.), такой канал называется *мультиплексным*.[2] Остальные каналы работают с быстрыми внешними устройствами (обычно это дисковая память), такие каналы называются *селекторными*. В отличие от мультиплексного канала, который успевает, быстро переключаясь с одного медленного внешнего устройства на другое, обслуживать их все как бы одновременно, селекторный канал в каждый момент времени может работать только с одним быстрым внешним устройством. На рис. 14.6 показана архитектура ЭВМ с каналами ввода/вывода.

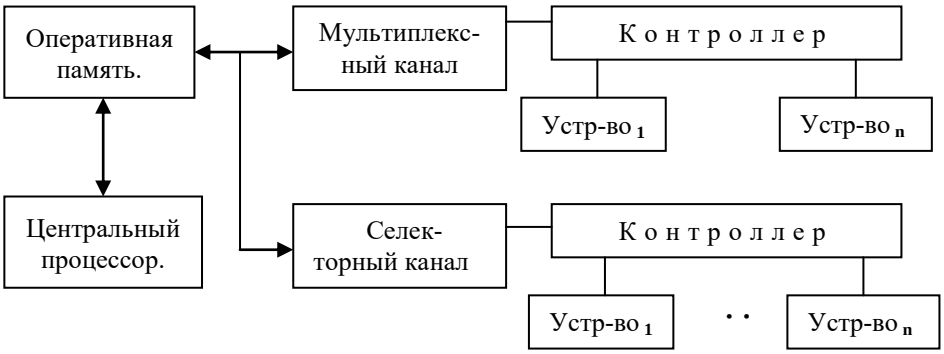


Рис. 14.5. Схема ЭВМ с каналами ввода/вывода.

Как видно из этого рисунка, внешние устройства подключаются к каналам не напрямую, а через специальные электронные схемы, которые называются контроллерами. Это связано с тем, что каналы являются универсальными, они должны допускать подключение внешних устройств, очень разных по своим характеристикам. Таким образом, канал работает как бы с некоторыми *обобщёнными* (абстрактными) внешними устройствами, а все особенности связи с конкретными устройствами реализуются в контроллерах.

---

[1] Исключением являются каналы для управления очень сложными устройствами, это, например, современные графические процессоры (графические карты). Их машинный язык по сложности сопоставим с языком центрального процессора.

[2] На современных персональных ЭВМ некоторым приближением к такому мультиплексному каналу является так называемый контроллер USB шины, к нему может подключаться до 127 различных внешних устройств. Правда, это не канал, а именно контроллер, так как он не допускает программирование для него процедур ввода/вывода.

Например, один контроллер предназначен для подключения к каналу жестких дисков, другой – архивных накопителей на магнитной ленте (так называемых стримеров), третий – накопителей на лазерных дисках и т.д. Примерами таких устройств на старых персональных ЭВМ являлись, например, контроллеры `IDE1` и `IDE2`, к каждому из которых можно подключить по два достаточно быстрых устройства для работы с жесткими и оптическими дисками. Сейчас для такого подключения используются более совершенные контроллеры (`SATA` и др.). Правда, на персональных ЭВМ все такие контроллеры подключаются не к каналу ввода/вывода, а к более примитивным (не программируемым) так называемым устройствам прямого доступа к памяти (DMA – Direct Memory Access).

Как уже говорилось, для компьютеров с общей шиной при выполнении системного вызова процессор переключается на процедуру-обработчика прерывания, и эта процедура и реализует требуемое действие, например, чтение массива с диска в оперативную память. Другими словами, во время выполнения процедуры-обработчика прерывания программа пользователя, естественно, не считается, так как центральный процессор занят выполнением служебной процедуры.

Совершенно по-другому производится обработка системного вызова на компьютере с каналами ввода/вывода. После того, как программа пользователя произведёт системный вызов, вызванная процедура-обработчик прерывания посылает соответствующему каналу приказ начать выполнение *программы канала*, реализующей требуемое действие, после чего может производиться немедленный возврат на продолжение выполнения программы пользователя. Далее может производиться *параллельная* работа процессора по выполнению программы пользователя и канала, выполняющего свою собственную программу по обмену с внешними устройствами, например, по чтению массива с диска в оперативную память. Вот пример, на некотором «параллельном» языке:

```
type Mas=array[1..N] of double;
     FM=file of Mas;
var X,Y: Mas; F: FM;
...Read(F,X); {обработка массива Y и чтение массива X};
...Read(F,Y); {обработка массива X и чтение массива Y};
```

В архитектуре с общей шиной обработку массива `Y` нельзя начать, пока не введётся массив `X`, так как процессор в это время занят вводом `Y`. В Архитектуре с каналами ввода/вывода процессор обрабатывает массив `Y`, пока канал параллельно вводит массив `X`. В процессорах фирмы Intel похожую работы могут выполнять так называемые контроллеры прямого доступа в память, но программированию они не поддаются.

В теории программирования ввод/вывод называется блокирующим, если процесс, производящий операцию обмена, ждет полного завершения ввода/вывода, и лишь потом может продолжить свой счёт. До сих пор в языках высокого уровня и Ассемблере Вы имели дело только с таким блокирующим вводом/выводом. Не блокирующий (Lock-Free) ввод/вывод (в некоторых ОС он называется асинхронным), позволяет производить счёт программы *параллельно* с выполнением для неё же операции обмена. Как видим, архитектура ЭВМ с каналами позволяет производить как блокирующий, так и не блокирующий ввод/вывод.

Естественно, что *немедленное* продолжение выполнения программы пользователя после начала обмена возможно только в том случае, если этой программе не требуется *немедленно* обрабатывать данные, которые должен предоставить канал. Например, если программа обратилась к каналу для чтения массива с диска в свою оперативную память, и пожелает тут же начать суммировать элементы этого массива, то такая программа будет переведена операционной системой в состояние ожидания, пока канал полностью не закончит чтения заказанного массива в память. Аналогично программа будет переведена в состояние ожидания, если она обратилась к каналу для *записи* некоторого своего массива из оперативной памяти на диск, и пожелала тут же начать присваивать элементам этого массива новые значения. Для предотвращения таких ситуаций существуют особые аппаратные и программные средства, позволяющие, как говорят, *синхронизировать* параллельную работу нескольких устройств. Такие средства изучаются в курсе по операционным системам. Эти примеры, в частности, показывают, насколько усложняется большинство программ операционной системы в архитектуре с каналами ввода/вывода по сравнению с архитектурой с общей шиной.

В заключение этого краткого рассмотрения организации взаимодействия центральных и периферийных частей компьютера стоит отметить, что в настоящее время архитектура с общей шиной в

чистом виде встречается только в самых простых ЭВМ (обычно в тех специализированных компьютерах, которые встраиваются в стиральные машины, холодильники, плееры и т.д.). Даже современные персональные ЭВМ массового выпуска содержат в своей архитектуре, помимо общей шины, различные средства прямого обмена с оперативной памятью (DMA – Direct Memory Access), похожие на простейшие каналы ввода/вывода (правда, обычно без средств их программирования пользователем). Единственными «настоящими» каналами в персональных ЭВМ сейчас являются видеокарты, снабженные своей памятью и графическими процессорами, работающими, естественно, параллельно с ядрами центрального процессора.

Дальнейшим развитием этой архитектуры является многоядерные процессоры, где нараду с обычными (вычислительными) ядрами есть и специализированные, «заточенные» для специальных целей. Это может быть, например, «экономичное» ядро для работы в режиме энергосбережения, разные тензорные, нейронные и другие экзотические процессоры.

14.5. Уровни параллелизма

Ничто не даётся даром в этом мире, и приобретение знания – труднейшая из всех задач, с какими человек может столкнуться. Человек идёт к знанию так же, как он идёт на войну – полностью пробуждённый, полный страха, благоговения и безусловной решимости. Любое отступление от этого правила – роковая ошибка.

Карлос Кастанеда.
«Учение дона Хуана»

Как Вы знаете, первые компьютеры удовлетворяли почти всем принципам фон Неймана. В этих компьютерах поток последовательно выполняемых в процессоре команд обрабатывал поток данных. ЭВМ такой простой архитектуры носят в литературе сокращенное название ОКОД (*Один* поток *Команд* обрабатывает *Один* поток *Данных*, английское сокращение SISD – Single Instruction Single Data). В настоящее время компьютеры, однако, нарушают почти все принципы фон Неймана, как уже упоминалось ранее, не нарушаются только принцип хранимой программы и принцип программного управления (последний лежит в основе *определения* цифровых ЭВМ.

По оценкам за время развития вычислительной техники производительность компьютеров выросла примерно в миллиард раз. За это время произошло уменьшение времени такта работы с 2 мкс. (одна из первых ЭВМ EDSAC, 1949 год), до примерно 0.25 нс. в современных процессорных ядрах. Это дало прирост скорости только примерно в 8000 раз, всё остальное обеспечило развитие архитектуры параллельной обработки данных.

В период бурного развития вычислительной техники каждый год с 1986 по 2003 год скорость работы компьютеров возрастала примерно на 50%. В дальнейшем прирост производительности упал до 20% в год, а в настоящее время мощность каждого отдельного процессора почти не растёт. Дело в том, что вычислительная мощность современных компьютеров базируется уже не на скорости работы всех узлов ЭВМ, а, в значительно большей степени, на *параллельной обработке данных*. Это связано с тем, что увеличение быстродействия каждого отдельного узла ЭВМ уже приближается к тем критическим ограничениям, которые накладывает конечность скорости распространения электромагнитных сигналов (в просторечии «скорости света»).

Взрывной рост производительности ЭВМ, наряду со стремительным расширением области применения компьютеров, имел и существенный негативный эффект. Дело в том, что для подавляющего числа приложений мощность компьютеров катастрофически (на много порядков) больше необходимой. Не случайно практически всегда единственный вычислительным потоком на персональных ЭВМ является «Бездействие системы» (System Idle Process), он «выполняется» более, чем 99.999% всего времени.

Как следствие, стало практикой использование для написания приложений таких систем программирования, которые дают понятные, но крайне неэффективные программы. Типичным примером является язык Python. Основное внимание при этом уделяется простоте и удобству работы с такими приложениями. Теперь большинство прикладных программистов даже не задаются вопросом об эф-

фективности используемого алгоритма. Действительно, при скорости компьютера в несколько миллиардов операций в секунду не имеет значения, будет ли очередной шаг алгоритма выполнен за `10 мкс.` или `10 мс.`, пользователь всё равно ничего не заметит.

Для хранения данных при этом можно всегда использовать динамические структуры данных (чаще всего разные списки). Как Вы знаете, для доступа к элементу массива нужна одна команда (с индексным и/или базовым регистром), а вот для доступа к элементу списка понадобится цикл. Кроме того, элементы массива обеспечивают хорошую пространственную локальность данных, а элементы списка, хаотически разбросанные по памяти, уже нет.

Другим примером может служить глубокая иерархия системных вызовов ОС. Например, обращение к процедуре `read` повлечёт последовательный вызов целой цепочки системных функций, последняя из которых из ядра ОС и производит, собственно, вывод данных. Можно привести в качестве примера и хранение всех параметров приложений в некоторой универсальной базе данных (реестр в Windows). Например, если вызвать обычный текстовый редактор Блокнот (`notepad.exe`), то перед началом работы для своей настройки он обратиться к реестру более 10000 раз ⚠.

Разумеется, сохранились задачи, требовательные к производительности, для которых надо составлять очень эффективные алгоритмы, но таких задач очень мало, и подавляющее большинство программистов просто не задумывается (да и не понимают) что такое эффективность.

В заключение этого краткого изучения архитектуры современных ЭВМ будет рассмотрена классификация способов параллельной обработки данных на компьютере.

- Параллельное выполнения программ может производиться на одном компьютере, если он имеет несколько процессоров (ЭВМ массового производства этого класса являются многоядерные компьютеры). Как правило, в этом случае такой компьютер имеет и несколько периферийных процессоров (каналов). Существуют (супер) ЭВМ, у которых могут быть от сотен тысяч до многих миллионов процессоров. В таких компьютерах много потоков команд одновременно обрабатывают много потоков данных, в научной литературе это обозначается сокращением МКМД (или по-английски MIMD – Multiple Instruction Multiple Data).[1] Ясно, что какой-то процент этих процессоров обеспечивают ввод/вывод.
- Параллельные процессы в рамках одной программы. Одна программа пользователя может породить несколько параллельных задач обработки данных, каждая такая задача (точнее, составляющие её вычислительные потоки) является для операционной системы самостоятельной единицей работы (эта возможность уже рассматривалась при изучении реентерабельных программ). Таким образом, вычислительные потоки *одной* программы могут параллельно выполняться в мультипрограммном режиме точно так же, как и потоки, порожденные программами разных пользователей.

  В качестве примера рассмотрим простой случай, когда программисту необходимо вычислить сумму значений двух функций `F(x)+G(x)`, причём каждая из этих функций для своего вычисления требует больших затрат процессорного времени и производит много обменов данными с внешними запоминающими устройствами. В этом случае программисту выгодно распараллелить алгоритм решения задачи и породить в своей программе два *параллельных потока*, каждому из которых поручить вычисления одной из этих функций. Можно понять, что в этом случае вся программа будет посчитана за меньшее *физическое* время, чем при использовании одного вычислительного потока. Действительно, пока один поток будет производить обмен данными с медленными внешними устройствами, другой поток может продолжать выполняться на центральном процессоре, а в случае с одним потоком вся программа была бы вынуждена ждать окончания обмена с внешним устройством. Стоит заметить, что скорость счёта программы с несколькими потоки ещё больше возрастет на компьютерах, у которых более одного центрального процессора. Подробно параллельные процессы изучаются в книгах, посвященных операционным системам.

---

[1] В настоящее время популярен также подход, при котором большое количество самостоятельных ЭВМ объединяются высокоскоростными шинами связи в вычислительный комплекс, называемый *кластером*. Такие кластеры могут состоять от нескольких десятков до сотен тысяч компьютеров, которые все могут параллельно решать одну большую задачу, обмениваясь между собой данными по высокоскоростным каналам связи.

- Параллельное выполнение нескольких команд одной программы может производиться, как Вы знаете, *конвейером* процессора. В современных ЭВМ конвейер устроен весьма сложно, в разных стадиях выполнения на нём может находиться несколько десятков команд.
- Параллельная обработка данных в *одном* вычислительном процессе производится и на так называемых *векторных* и *матричных* ЭВМ.

  У *векторных* ЭВМ наряду с обычными (скалярными) регистрами есть и *векторные* регистры, которые могут хранить и выполнять операции над векторами целых или вещественных чисел. Пусть, например, у такой ЭВМ есть регистры с именами `ZMM0` и `ZMM1`, каждый из которых может хранить вектор из `512:32=16` чисел, тогда, например, команда векторного сложения **`vpaddsd`** `zmmx0,zmmx1` будет производить параллельное покомпонентное сложение всех элементов таких векторов по схеме. Как уже говорилось, векторные регистры сейчас есть и на персональных ЭВМ, они могут обрабатывать вектора как целых, так и вещественных чисел.

  У *матричных* ЭВМ на одно устройство управления приходится много (иногда до нескольких сотен или даже тысяч) арифметико-логических устройств. Таким образом, выбранная в устройство управления команда, например сложения, параллельно исполняется над разными операндами в каждом из арифметико-логических устройств. Можно сказать, что на векторных и матричных ЭВМ один поток команд обрабатывает много потоков данных. Отсюда понятно и сокращённое название ЭВМ такой архитектуры – ОКМД (по-английски SIMD: Single Instruction Multiple Data).

В 1966 году Майкл Флинн (Michael J. Flynn) предложил подход к классификации вычислительных систем, основанный на понятии последовательностей (потоков) команд и данных. Исходя из этого, деление архитектур ЭВМ по способу организации вычислительного процесса (ОКОД, МКМД и ОКМД) в компьютерной литературе часто называется *классификацией по Флинну*. Для полноты этой классификации следует упомянуть и архитектуру МКОД (MISD – Multiple Instruction Single Data), при этом один поток данных обрабатывается параллельно многими потоками команд. В качестве несколько надуманного примера можно привести специализированную ЭВМ, управляющую движением самолетов над крупным аэропортом. В этой ЭВМ один поток данных от аэродромного радиолокатора обрабатывается многими параллельно работающими процессорами, каждый из которых следит за безопасностью полета одного закрепленного за ним самолета, давая ему при необходимости команды на изменение курса, чтобы предотвратить столкновение с другими самолетами. С другой стороны, однако, можно считать, что каждый процессор обрабатывает свою *копию* общих входных данных, и рассматривать это как частный случай архитектуры МКМД, так и делается во многих учебниках, в которых класс МКОД считается пустым. Отметим, что у самого Флинна эта классификация более разветвленная, в каждом классе выделяются свои подклассы по способам связи между собой элементов вычислительной системы и единицам обрабатываемых данных.

Параллельная обработка команд и данных позволяет значительно увеличить производительность компьютера. Необходимо сказать, что в современных высокопроизводительных компьютерах обычно реализуется сразу несколько из рассмотренных выше уровней параллелизма.

Познакомиться с историей развития параллельной обработки данных можно, например, по книгам [13,21]. Заметим, однако, что, несмотря на непрерывный рост мощности компьютеров, постоянно появляются все новые задачи, для счёта которых необходимы ещё более мощные ЭВМ. Таким образом, к сожалению, рост сложности подлежащих решению задач все время *опережает* рост производительности компьютеров. Например, только для составления *местного* (например, для Московской области) и краткосрочного (на одни сутки вперед) *достоверного* прогноза погоды необходим компьютер с производительностью порядка 1000 млрд. операций над вещественными числами в секунду.

14.6. Развитие суперкомпьютеров

Программы становятся медленнее куда шустрее, чем компьютеры становятся быстрее.

Никлаус Вирт

Производительность мощных компьютеров меряется в единицах, называемых *флопсами* (flps). Один flps равен одной операции над длинными (`double`, 64-разрядными) *вещественными* числами

в секунду (floating point per second).[1] Подчеркнем, что это операции не над целыми, а именно над вещественными числами, которые для компьютера значительно более трудоемкие. Таким образом, для расчёта такого очень простого прогноза погоды необходима производительность компьютера порядка 100 Gflops (100 Гфлопс). Производительность мощных современных супер-ЭВМ составляет от единиц до сотен Пе(н)тафлопс (Пфлопс) (1Pflps = 10^{15} flps).

Например, производительность теперь уже устаревшего суперкомпьютера серии IBM BlueGene/L выпуска 2005 года оценивалась в 138 Тфлопс, он примерно в 1000 раз мощнее, чем знаменитый компьютер Deep Blue, которому в 1997 году проиграл в шахматы тогдашний чемпион мира Гарри Каспаров. Это вычислительная система, имела в своём составе 2^{16} узлов, содержащих двухядерные процессоры PowerPC 440 с общей памятью (правда, для вычислений использовалось только одно ядро, второе ядро каждого узла занималось взаимодействием с остальными процессорами, и 1024 узла заняты операциями ввода/вывода). На каждом узле (кроме узлов ввода/вывода) мог выполняться только один пользовательский процесс, по одному программному потоку на ядро. В среднем раз в 10 дней некоторый узел давал сбой, это требовало отбраковку этого узла и повторения вычислений, что, однако, не сказывалось на работоспособности всей ЭВМ. Максимальная скорость работы этой ЭВМ оценивалась в 4.6 10^{14} команд в секунду. Deep Blue строили 6 лет, он стоил 6 млн. долларов. При игре использовалась база данных из примерно 700 000 партий с дебютами и эндшпилями, компьютер оценивал позицию примерно на 12 ходов вперёд.

Производительность мощных ЭВМ обычно оценивается с помощью особых тестов, например, тестов High-Performance LinPack benchmark (или просто LinPack), ориентированных на операции с вещественными числами. Данные о 500 самых мощных компьютерах по этой системе тестирования приведены на сайте http://www.top500.org, данные обновляются каждые полгода. Данные о 50 самых мощных суперкомпьютеров стран СНГ приведены на сайте http://top50.supercomputers.ru/newsfeed.

Суперкомпьютеры имеют чёткую модульно-иерархическую структуру. В качестве примера слева показана архитектура суперкомпьютера Blue Gene/P фирмы IBM выпуска 2007 года. Видно, как из многоядерных микропроцессоров и устройств памяти последовательно собирается суперкомпьютер.

По состоянию на июнь 2008 года самым мощным суперкомпьютером счиался гетерогенный (неоднородный) кластер Roadrunner фирмы IBM с производительсотью 1026 Тфлопс. Этот компьютер состоит из 122400 ядер на базе процессоров PowerXCell 8i 3.2 Ггц, которые, собственно, и производят вычисления, и 6562 двухядерных процессоров AMD Opteron DC 1.8 Ггц, которые управляют всей работой и обеспечивают ввод/вывод. Стоимость IBM Roadrunner составила 133 миллиона долларов. Суперкомпьютер занимал площадь свыше 500 квадратных метров и весит более 220 тонн, он потребляет 942.35 Мвт, при этом энергоэффективность составляет 437 Mflops/Вт. Для сравнения, первая ЭВМ ENIAC выпуска 1946 года работала в десятичной системе счисления, содержала 17.5 тысяч электронных ламп, занимала зал площадью 150 квадратных метров и весила около 30 тонн. Отсюда можно понять шутливое определение, что суперкомпьютер – это ЭВМ, которая весит более тонны.

Начиная с 2013 года, три года подряд лидером с производительностью 33.86 Пфлопс являлся суперкомпьютер Tianhe-2 (Млечный путь-2), сконструированный в китайском Национальном университете оборонных технологий совместно с компанией Inspur. Этот компьютер установлен в Суперкомпьютерном центре в городе Гуанджоу, он содержит 16000 узлов, в каждом по два 12-тиядерных процессора Intel Xeon E5-2692 (частота 2.2 Ггц) и три специализированных сопроцессора Xeon Psi 31S1P по 57 ядер в каждом (частота 1.1 Ггц), узел оснащен оперативной памятью 88 Гб. Общее количество вычислительных ядер составляет 3.12 миллиона (384 тыс. Intel Xeon и 2736 тыс. Xeon Phi). Суперкомпьютер размещается примерно в 150 шкафах-стойках, потребляемая мощность 24 Мвт (хватит примерно на 16 тыс. квартир – большой микрорайон.[2] Стоимость этого суперкомпьютера составляет около 390 млн. долларов.

---

[1] В настоящее время набирает популярность и дополнительные методы оценки производительности ЭВМ, это отношение числа операций к единице затраченной электрической мощности (flops/Вт) и числа операций к единице площади, занимаемой ЭВМ (flops/м^2).

[2] Не так уж и много, современный дата центр (или ЦОД – центр обработки данных) потребляет порядка 100 мегаватт (Мвт), из них не менее 30% на охлаждение. На все такие центры расходуется около 1% мирового потребления электроэнергии. Для сравнения, Москва потребляет порядка 12000 Мвт=12 Гвт.

Начиная с 2016 года лидерство среди суперкомпьютеров захватила машина китайского производства Sunway TaihuLight (Божественная сила света озера Тайху), с производительностью 93 Пфлопс, к тому же этот суперкомпьютер занимал третье место по энергоэффективности (6 Гфлопс/ватт, всего 28 Мвт). Эта ЭВМ содержит 40960 RISC процессоров китайского производства SW26010, каждый имеет 256 вычислительных и 4 управляющих ядра (всего 10.649.600 ядер). В 2019 году Sunway Taihu Light переместился на 3-е место.

Суперкомпьютер IBM Summit

В 2018 году, после 5-летнего первенства китайских ЭВМ, лидерство перешло к американскому суперкомпьютеру фирмы IBM Summit с пиковой производительностью 148.8 Пфлопс, он установлен в Национальной Лаборатории Ок-Ридж (США). Summit содержит 4356 узлов, в каждом из которых два 22-ядерных процессора IBM Power9, плюс 6 графических процессоров NVidia Tesla V100, всего 2 414 592 вычислительных ядер. Эта супер-ЭВМ стоит 200 млн.долл., весит 340 тонн, занимает зал площадью 860 кв.м., стойки с оборудованием соединяются оптическими кабелями общей длиной 300 км. ЭВМ потребляет 15 Мвт, по её контуру охлаждения прокачивается 15 тонн воды в минуту. При отключении системы охлаждения и 90% нагрузке температура в машинном зале такой супер-ЭВМ начинает повышаться на один градус в секунду 😕.


Суперкомпьютер Frontier, 2022 год

Сейчас развёрнуты разработки супер-ЭВМ с проектной производительностью 1 экс(з)афлоп (10^{18} флопс). Первый такой компьютер Frontier на базе процессоров EPYC Milan фирмы AMD в мае 2022 года показал на тесте Linpack производительность 1.1 экс(з)афлоп, а пиковая скорость достигала 1.69 экс(з)афлоп. Система состоит из 74-х стоек, в которых размещены 9408 вычислительных узлов, каждый узел содержит 64 ядерный процессор Trento (итого 602112 процессорных ядер), 4 графических процессора Radeon Instinct MI250X (итого 8 138 240 графических ядер) и в сумме 4.6 Пб памяти DDR4. Суперкомпьютер потребляет 21 Мвт, его насосы системы водяного охлаждения прокачивают 6000 галлонов воды в минуту и способны заполнить олимпийский бассейн за 30 минут ⚠.


Система охлаждения Frontier, 2022

Эта супер-ЭВМ занимает первое место в рейтинге самых энергоэффективных ЭВМ (так называемом Green-500) с показателем 62.68 гигафлопс/ватт.

Правда, на конец 2022 года этот компьютер всё еще страдал от регулярных сбоев и находился в стадии тестирования.

В 2022 году Россия занимает 8-ое место в мире по совокупности факторов количества и мощности супер-ЭВМ. Самый мощный суперкомпьютер России Червоненкис находится на 19-м месте в мире, к сожалению, его производительность всего 21.53 петафлопса. Суперкомпьютер «Ломоносов-2», установленный в Московском государственном университете на Ленинских Горах, находился на 241-м месте с производительностью 901 Тфлопс.

Разумеется, ЭВМ этого класса выпускаются в единичных экземплярах по специальному заказу.[1] Необходимо также учитывать, что такая высокая производительность достигается суперкомпьютерами только на специальных задачах, допускающих глубокое распараллеливание вычислений. При выполнении «обычных» программ не удается загрузить работой большое количество вычислительных

[1] Интересно отметить, что значительная часть парка суперкомпьютеров используется в игровой индустрии для реализации виртуальных миров, в этих играх одновременно могут принимать участие десятки и сотни тысяч игроков. Самым крупным рынком видеоигр является Китай, например, в 2017 году трансляцию финала чемпионата компьютерной игры League of Legends из Пекина смотрело 106 млн человек.

ядер и эффективность применения суперкомпьютеров для таких задач может упасть в несколько десятков и даже сотен тысяч раз.

Относительно высокую вычислительную мощность показывают и специализированные компьютеры. Например, игровая приставка седьмого поколения PlayStation 3 выпуска 2006 года показала производительность 2 Тфлопс, правда на коротких 32-разрядных, а не на длинных 64-разрядных вещественных числах.

Заканчивая первое знакомство с параллельной обработкой данных, необходимо сказать и о так называемых GRID-системах. GRID технология позволяет использовать для решения задач большое число не очень мощных (например, персональных) ЭВМ, объединенных в единую систему посредством не очень быстрых (например, сеть Интернет) каналов связи. GRID технология применима для решения сложных задач, которые допускают распараллеливание на очень большое число слабо связанных между собой подзадач (вариантов). Каждый узел GRID-системы получает с одного из центральных узлов (сайтов) для решения свою подзадачу с её входными данными, и после решения подзадачи отправляет результат своей работы назад центральному узлу, после чего получает для решения новую подзадачу и т.д. Существуют стандартные средства (например, Globus Toolkit, http://www.globus.org) для объединения в такую систему компьютеров с разными аппаратными платформами и работающими под управлением разных операционных систем.

Самым известным проектом GRID технологии считается проект SETI (Search for Extraterrestrial Intelligence) – некоммерческий проект для поиска внеземных цивилизаций с помощью свободных ресурсов на компьютерах пользователей, которые добровольно предоставляют эти ресурсы проекту. Обычно это персональные компьютеры, на которые ставится специальная программа заставки (хранителя экрана), на самом деле являющаяся клиентом сервера проекта. Во время простоя компьютера эта программа запускает на нём счёт варианта задачи поиска. Другой известный проект, в котором использовалась GRID технология – расшифровка генома человека.

Вопросы и упражнения

Сомневайся во всём.

Рене Декарт

1. Что такое расслоение оперативной памяти и для чего оно нужно ?
2. Что такое память типа кэш и для чего она необходима ? Почему эта память строится на схемах статической, а не динамической памяти ?
3. Что такое конвейер процессора и как он работает ?
4. Как будет работать конвейер, если в потоке выполняемых команд есть зависимость по данным?
5. Нарисуйте новую схему работы конвейера (см. таблицу 15.1) для оптимизированного фрагмента этой программы (без зависимостей по данным) и убедитесь, что там не будет пустых мест.
6. Как обрабатываются на конвейере команды условных переходов ?
7. Почему работа конвейерной ЭВМ сильно замедляется при частых сигналах прерывания ?
8. Какие компьютеры называются суперскалярными ?
9. Что собой представляет шина ?
10. Для чего нужен арбитр общей шины и как он работает ?
11. Какие достоинства и недостатки имеет архитектура связи процессора с устройствами ввода/вывода при помощи общей шины ?
12. Что такое канал ввода/вывода ?
13. Что такое блокирующий и не блокирующий ввод/вывод ?
14. Для чего нужна многовходовая оперативная память ?
15. В чём главная идея архитектуры компьютеров с каналами ввода/вывода ?
16. За счёт чего достигается параллельная работа процессора и устройств ввода/вывода ?
17. Когда может понадобиться синхронизовать работу процессора и устройства ввода/вывода ?
18. Что означает, что в компьютере реализована схема МКМД обработки данных ?
19. Что такое векторная ЭВМ ?

[i] Для продвинутых читателей.

Сначала о скорости работы памяти. Доступом к основной (оперативной) памяти (ОЗУ) управляет специальная электронная схема – контроллер памяти, именно к нему обращаются остальные устройства для обмена с ОЗУ. Сейчас контроллер располагается внутри процессора, он обслуживает все процессорные ядра и кэш-память. Обычно контроллер имеет две и более шины для связи с оперативной памятью, он, как говорят, является многоканальным. Например, двухканальный контроллер по каждой из двух 64-битных шин читает из основной памяти сразу по 16 байт с последовательными адресами. Как мы вскоре узнаем, считанные байты чаще всего поступают не прямо в процессор, а в специальную промежуточную кэш-память. А вот кэш-память связана с процессорными ядрами уже более широкой шиной, обычно 256 бит (32 байта).

Современная динамическая оперативная память быстрее всего работает в так называемом *потоковом* режиме, когда обмен производится сразу большими блоками (скажем, по 2 или 4 Кб). При этом на поиск первой ячейки с заданным начальным адресом тратится достаточно большое время (около 200 тактов), это так называемая *латентность* (latency – задержка) памяти. Эта задержка тем больше, чем дальше новый адрес расположен относительно предыдущего адреса обмена. Дальнейшая скорость чтения и записи ячеек памяти с последовательными адресами резко возрастает, составляя около 20-25 Гбайт в секунду. Таким образом, приведённое время 5-10 нс. является средним за достаточно большой промежуток времени. Можно сказать, что это нарушение принципа Фон Неймана однородности памяти. Любопытно отметить, что так же работала и память на уже упоминавшихся ранее ртутных линиях задержки в первых ЭВМ, построенных по схеме Фон Неймана. По схожему принципу устроена и память на жёстких магнитных дисках: после достаточно большого времени поиска нужного сектора на магнитной дорожке далее следует быстрое последовательное чтение секторов диска (или цилиндра).

Теперь о скорости выполнения команд в процессоре. Сейчас в 1 нс. примерно 2-4 процессорных такта (частота процессора 2-4 Ггц). В каждом ядре имеются 4 обрабатывающий устройства (порта) для выполнения операций над данными и 4 порта для ввода/вывода, так что в пределе может обрабатываться до 4-х команд за такт. Таким образом, при частоте 3 Гг. максимальная скорость составляет около 10 млрд. операций в секунду (в реальности, конечно, значительно меньше).

Сложные команды требуют больше тактов для своего выполнения, например, команда (целочисленного) умножения требует примерно в 3-5 раз больше тактов, чем сложение, а деления – в 20-25 раз больше (примерно столько же тактов занимает и деление *вещественных* чисел).

Школьный алгоритм умножения двух N-значных целых чисел по существу сводится к сложению (со сдвигом) «в столбик» N таких чисел, т.е. имеет сложность N (команд сложения). Первый алгоритм быстрого умножения целых чисел предложил в 1960 году математик А.А. Карацуба. Теоретически можно снизить сложность алгоритма целочисленного умножения до $O(\log_2 N)$ сложений, такой алгоритм найден в марте 2019 года математиками Дэвидом Харви и Йорисом ван дер Хувеном, правда, он полезен только при умножении *очень* больших чисел. Реализованные сейчас в ЭВМ алгоритмы умножения имеют немного бóльшую сложность. Вот и получается, что процессору для умножения двух 32-разрядных целых двоичных чисел требуется примерно $\log_2 32=5$ операций сложения. Теоретически, используя упомянутые ранее асинхронные схемы, можно ухитриться сделать все 5 сложений за один такт, но обычно так сделать не удаётся и требуется 3-4 такта. Быстрее работают двух и трёхадресные команды умножения нашей ЭВМ, которые дают только *младшую* часть произведения (т.е. работают не всегда правильно).

Для деления целых чисел дело обстоит хуже, этот алгоритм имеет сложность порядка $O(\log_2 N)^2$, что и даёт для деления двух 32-разрядных двоичных чисел примерно 25-30 операций (вычитания). В современных RISC процессорах иногда реализуют более быстрый алгоритм, дающий только частное (без остатка), для его реализации необходимо много (например, 32 или 64) параллельно работающих схем вычитания. Разумеется, использования конвейера позволяет существенно увеличить скорость выполнения *потока* команд, среди которых не так много команд умножений и деления, которые будут выполняться параллельно с простыми арифметическими и логическими командами. Более подробно об этом будет говориться далее, при объяснении принципов работы конвейера.

[ii] Для продвинутых читателей. Два банка памяти можно лодключить к одному каналу, пока один банк ищет в своеё памяти нужные данные, второй передаёт данные по каналу, затем наоборот.

Иногда возникает необходимость обеспечить ещё более широкую шину передачи данных, например, в графических картах, где в реальном времени надо читать, обрабатывать и выводить на современный дисплей (например, с разрешением 4К) очень большой объём данных. Возникает необходимость передавать по шине сразу, скажем, 1024 бит (пусть даже они будут читаться немного медленнее). Одно из решений этой проблемы называется стековая DRAM, при этом каждый банк такой памяти, в свою очередь, делится на стопку (stack) из 4-8 микросхем с параллельным доступом (двухуровневое расслоение). В итоге получается

4 банка*8 стеков*8 байт=256 байт=2048 бит

Такая технология организации памяти называется HBM (High Bandwidth Memory), другие варианты названия Wide I/O (Samsung) и HMC (Hybrid Memory Cube – фирмы Intel и Micron). В следующем поколении такой памяти стопка уже из 8-ми микросхем.

Одно из главных достоинств такой памяти состоит в том, что она может работать с меньшим напряжением питания и на меньшей частоте, что позволяет повысить энергоэффективность. Это очень важно, современные мощные (игровые) видеокарты потребляют энергии больше, чем весь остальной компьютер.

[iii] Для продвинутых читателей. Современная кэш-память может работать в специальном, так называемом Non-Eviction или No-Fill Mode режиме, как «обычная» адресуемая (статическая) SRAM память. В этом случае никаких «промахов» кэш и вытеснения (Eviction) страниц, конечно, не бывает. Обычно это нужно, когда «настоящая» оперативная динамическая SRAM память для процессора недоступна. Например, так бывает при включении компьютера, когда программа начальной загрузки начинает работать в режиме самых первых процессоров Intel. Оперативная память в этом случае имеет «древнюю» так называемую сегментную организацию, современные процессоры работать с такой памятью просто «не обучены», вот они и используют кэш-память в качестве секции данных (команды при этом читаются из ROM памяти программы начальной загрузки). Как говорится, голь на выдумки хитра.

[iv] Для продвинутых читателей. В литературе архитектура, в которой команды и данные хранятся в отдельных устройствах и для доступа к ним используются разные шины, называется *Гарвардской* (Harvard architecture), а архитектура с общей памятью команда и данных называется, как Вы уже знаете, архитектурой Фон Неймановской (иногда *Принстонской*, т.к. именно там тогда работал Фон Нейман). Гарвардская архитектура была впервые реализована в компьютере MARK III в Гарвардском университете, на современных ЭВМ так обычно устроена только кэш-память первого уровня.

Процессоры Intel имеют раздельные кэши команд (L1I) и данных (L1D) первого уровня (обычно по 32 или 64 Кб), причем свои для каждого процессорного ядра. Кэш-память следующих уровней (L2 и L3) уже общая для команд и данных, обычно L2 своя у каждого ядра (примерно по 512 Кб), а L3 общая для всех процессорных ядер (примерно по 2 Мб на ядро). Быстрая статическая память L1 и L2 обычно работает на частоте процессора. Ширина шин связи между процессором и L1, а также между кэш памятями разного уровня обычно 256 или 512 бит.

На работу такой многоуровневой системы кэш-памяти может тратиться до 50% потребляемой процессором электрической мощности, однако надо учесть, что при промахе кэша на обращение к основной DRAM памяти придётся потратить уже в 10 раз больше энергии.

Для *самомодифицирующихся* программ (у нас ЭВМ должна удовлетворять принципам фон Неймана!) при записи в секцию кода соответствующая строка кэш-памяти команд автоматически обновляется из кэш-памяти данных, сбрасывается кэш декодированных команд (см. ниже), а модифицированные команды на конвейере помечаются как недействительные. Ясно, что такие самомодифицирующиеся программы хоть и возможны, но не приветствуются. Частая самомодификация кода резко снижает скорость работы, можно сказать, что процессоры Intel «притворяются» , что имеют Фон Неймановскую архитектуру (т.е. не различают команды и данные), а на самом деле только эмулируют её, а всякая эмуляция, как известно, снижает производительность. В сложных случаях (программа выполняется на одном ядре, а модифицируется на другом) при этом возможна даже некорректная работа ЭВМ.

---

[v] Для (сильно) продвинутых читателей. Как уже говорилось, всё логическое адресное пространство оперативной памяти делится на страницы (обычно длиной по 4 Кб). Современные процессоры могут назначать разным областям оперативной памяти свои режимы, как говорят, *политики* (Policies) работы с кэш памятью. Можно назначать разные политики и для всей памяти, памяти одной задачи (процесса) из её текущего каталога страниц, а с помощью специальных регистров MTRRs (Memory Type Range Registers) для около сотни различных областей физической памяти, а также отдельным страницам через их атрибуты.

При записи данных из порта конвейера они сначала поступают в так называемый буфер сохранения (Store Buffer). Из этого буфера данные направляются либо в L1D кэш, либо сразу в оперативную память, это делается параллельно с выполнением в этом порте следующей микрооперации. Типичными являются следующие политики (стратегии) записи (Write Policies) в память.

Write Protected (WP). Запись запрещена, эта область памяти полностью закрыта на запись.

Write Back (WB). Полное кэширование. Это обычная, так называемая "обратная" запись только в кэш, в основную память изменения попадёт только тогда, когда данная строка будет удалена из кэшей *всех* уровней. Эта память допускает спекулятивное чтение команд и данных одновременно для двух ветвей условных операторов (об этом будет говориться далее, при рассмотрении работы конвейера). Это политика по умолчанию, если данная область памяти не подпадает под действие остальных политик.

Write Through (WT). Так называемая *сквозная запись*, т.е. запись в кэш и одновременно (принудительно) в основную память.

Write Combining (WC). Запись в память через буфер, вмещающей всего одну строку кэш памяти первого уровня. Это, как уже упоминалось, "кэш для кэша". При записи данных с близкими адресами, такие данные сначала накапливаются в этом буфере, пока не наберётся целая строка кэш. Запись в кэши всех уровней при этом не производится, хотя, если данные там есть, они объявляются недействительными. Важно, что при этом теряется порядок, с которым записываются данные, это более эффективно, но для некоторых программ может быть неприемлемо. Для программиста для такой записи предоставляется команда **movdir64b**, *атомарно* записывающая 64 байта по адресу в сегменте ES, заданному первым операндом этой команды. Есть аналогичные команды **movdiri** для записи 4 и 8 байт.

Uncacheable (UC). Кэширование запрещено, запись только в память. Обычно используется для буферов ввода/вывода, данные из которых драйверы должны сразу направлять из оперативной памяти во внешние устройства, скажем, на диск. Спекулятивное чтение команд и данных запрещено. Строго соблюдается последовательность операций (данные одной команды не могут попасть в память раньше данных из предыдущих команд). Различают строгую и мягкую политику UC, для мягкой возможно использование политики WC. Программист может сделать некоторую секцию своей программы некэшируемой, задав параметр K для редактора внешних связей, например:

```
link /subsystem:console /section:.data,K p.obj
```

Этот режим не действует для записи в память с вещественных регистров (включая векторные). Такое ограничение несущественно, так как современные процессоры имеют специальные команды, которые пишут данные с регистров (включая векторные) сразу в память, минуя кэш. Например, для обычных регистров это команда `movnti m32,r32`. Для векторных регистров команда `vmovdqa m256,ymm0` пишет 32 байта из регистра ymm0 в память как обычно, через L1D кэш, а вот команда `vmovdntqa m256,ymm0` пишет с регистра прямо в память, минуя кэш (требуется MASM 10.xx). Буквы **nt** в коде операции означает Non-Temporal, т.е. это данные «не временны́е» (не обладающие временно́й локальностью), они в ближайшее время процессору не понадобятся, и пусть они лучше не «засоряют» кэш. Ясно, что при этом адрес записи не должен попадать в страницу с атрибутом Write Protected. Не надо понимать слова «минуя кэш» слишком буквально, при этом обязательно проверяется, что ни в каком кэше (во всех ядрах) не осталось старой копии этих данных, иначе они удаляются из кэш памяти (точнее, соответствующие строки кэш помечаются как недействительные). Команды не временно́го обмена выполняются примерно в 400 раз медленнее, чем обычные команды, так что их надо использовать с осторожностью, только если Вы точно знаете, что делаете. Их можно эффективно использовать только для обмена объёмом не менее 512 Кб.

Конечно, существуют и соответствующие политики для чтения (Read Policies) из памяти команд и данных.

Отдельно стоит упомянуть технологию Intel SGX (Software Guard Extensions), для этого с 2015 года в процессорах появился набор новых команд. Эта технология позволяет создавать в виртуальной памяти так называемые *анклавы* (Enclave) – защищённые области памяти PRM (Processor Reserved Memory). В отличие от обычных страниц виртуальной памяти, каждая страница PRM может принадлежать только *одному* анклаву. Обмен с такими областями памяти производится процессором с помощью аппаратного модуля шифрования MEE (Memory Encryption Engine), т.е. другие программы не смогут даже прочитать из памяти анклава.

В анклавах можно запускать приложения, достаточно надёжно защищённые от других программ, даже работающих в режиме ядра, супервизора или системного управления. Привилегированные программы, конечно могут пытаться читать и писать в память анклава, но, без правильного ключа шифрования, модуль MEE будет выдавать исключения по защите памяти. Новые команды SGX позволяют передавать управление внутрь анклава, эти команды реализуют механизм, аналогичный переключению в защищённый режим работы. При возникновении прерывания процессор сначала выполняет так называемый Асинхронный Выход из Анклава AEX (Asynchronous Enclave Exit), при этом переключается в режим пользователя, а лишь затем производит аппаратную обработку этого прерывания. Отметим, что область сохранения TSS защищённого процесса при этом находиться внутри анклава. Для работы с виртуальными страницами памяти анклава процессор тоже использует специальные команды SGX.

Полезно сравнить анклав с так называемой «песочницей» (sandbox), областью памяти для выполнения опасного программного кода, который может попытаться навредить окружающей системе. В анклаве, наоборот, запускаются программы, которые надо защитить от влияния окружающей среды (даже операционной системы!), так как допускается, что эта среда уже «скомпрометирована» и может работать не так как предписано.

Можно также упомянуть, что с 2017 года в Intel появился новый набор команд TME (Total Memory Encryption) для полного шифрования DRAM. Шифрование производится в контроллере памяти, так что в процессоре и кэш памяти данные, к сожалению, по прежнему содержаться в незашифрованном виде. Включение и выключение шифрования производится через BIOS.

[vi] Для продвинутых читателей. Обеспечение когерентности памяти является одной из сложнейших проблем в современных процессорах, это требует многих миллионов вентилей. К сожалению, современные (многоядерные) процессоры так сложны, что содержат многочисленные ошибки (от десятков до сотен) в реализации каждой конкретной модели процессора. Обычно это «хитрые»ошибки, возникающие при очень редком стече-

нии обстоятельств. Бóльшая часть этих ошибок связана именно с ошибками в межъядерных взаимодействиях. В «ерьёзных»организациях дело доходит до того, что в ключевых управляющих компьютерах в процессоре отключают все ядра, кроме одного. О методах борьбы с аппаратными ошибками немного говорится далее в разделе, посвящённом принципу микропрограммного управления.

Кэш память называется *эксклюзивной* (Exclusive), если информация может находится в кэш памяти только одного уровня (уникальна). Наоборот, кэш-память называется *инклюзивной* (Inclusive), если кэш L2 содержит в себе данные, которые уже находятся в L1 кэше, а кэш L3 – данные кэша L2. *Не эксклюзивная* кэш память может вести себя как угодно. Процессоры Intel традиционно используют не эксклюзивный кэш, а процессоры AMD – эксклюзивный.

Для обеспечения когерентности кэшей существует несколько протоколов, в основном это протокол под именем MESI (Modified/Exclusive/Shared/Invalid), сейчас используются его модификации. По этому протоколу к каждой строке кэш памяти (всех уровней) прилагается двух битовый тэг, который имеет следующие значения:

M – модифицирована (Modified) – эта строка кэша изменена. Если определённая область основной памяти считана в один или несколько кэшей (разных ядер!), то строка только в одном кэше может иметь такой тэг. Этот тэг означает, что данная строка была изменена соответствующим ядром, но до основной памяти эти изменения ещё не дошли (там они устаревшие). Ядро-хозяин этого кэша может читать и писать в такую строку без опроса остальных кэшей.

E – эксклюзивная (Exclusive) строка. Имеющая такой тег строка, так же, как и строка с M-тегом, может находиться только в одном кэше. Данные в такой строке полностью идентичны данным в основной памяти, а в других кэшах эта строка отсутствует. Своё процессорное ядро может спокойно читать и писать в такую строку, однако после любой записи данная строка меняет свой тэг на значение M (Modified).

S – разделяемая (Shared) строка. Несколько строк с этим тегом могут одновременно содержаться в разных кэшах и использоваться совместно, чтение из таких строк не вызывает проблем. В то же время запросы на запись всегда синхронизируются, это приводит к тому, что строка модифицируется только в одном кэше, а содержание основной памяти обновляется. При этом все строки с таким адресов в остальных кэшах помечаются тэгом I (Invalid) как недействительные.

I – недействительная (Invalid) строка. Этим тэгом помечаются пустые строки, а также строки содержащие устаревшие данные. Попытка доступа к такой строке приводит к промаху кэша.

Как видно, этот протокол требует, чтобы контроллеры всех кэшей активно общались между собой по широковещательным (один передаёт, остальные слушают) шинам. Эта технология называется Snooping («подглядывание»), контроллер каждого ядра «присматривает» за кэш памятью остальных ядер.

В качестве примера разберём случай, когда производится запись данных в отсутствующую строку (случается промах кэша), значит, эту строку нужно сначала загрузить в свой кэш, а лишь потом модифицировать. Перед операцией чтения контроллер посылает широковещательное сообщение остальным ядрам, и, пусть данная строка содержится в каком-нибудь другом кэше. Рассмотрим два случая:

1) если строка в другом кэше имеет тег M (модифицирована), то чтение строки в первый кэш задерживается. Сначала ядро кэша с модифицированной строкой получает сигнал и записывает эту строку в основную память и присваивает своей строке тэг I (недействительная), а лишь затем возобновляется чтение строки в первый кэш. Затем ядро первого кэша модифицирует считанную строку и присваивает ей тэг M, при этом ни в одном другом кэше и в основной памяти нет правильной копии этой строки;
2) если строки в другом кэше не имеют тег M (не модифицированы), то чтение строки в первый кэш не задерживается, а все стоки в других кэшах получают тэг I (недействительны).

[vii] То, что «сверхсекретного» академика, да ещё в этот момент сильно простуженного, выпустили в 1956 году за границу, можно считать невероятным событием. К этому времени он уже создал БЭСМ (Большую [или Быстродействующую] Электронную Счётную Машину) и она была самой быстродействующей в Европе. Вскоре он приступил к разработке противоракетной обороны СССР, в 1961 году ракета, управляемая ЭВМ М-40, перехватила на испытаниях баллистическую ракету. Американцы смогли повторить этот результат только через 20 лет! В 1959 году С.А.Лебедев, вместе с двумя другими конструкторами ЭВМ, побывал в Америке, где ознакомился с компьютерами фирмы IBM. В 1975 году полётом кораблей проекта Союз-Апполон управляля разработанная С.А.Лебедевым ЭВМ БЭСМ-6. В 1996 году ему была присуждена престиэная медаль Computer Pioneer Award за выдающиеся достижения в области конструирования ЭВМ.

В 1961 году в МЭИ была создана первая в нашей стране кафедра «вычислительная техника», С.А.Лебедев читал там первый курс по цифровой вычислительной технике и программированию.

[viii] Для продвинутых читателей. Впервые режим работы конвейера OoO (Out of Order), был реализован в 1964 году в IBM System/360 model 91 и в ЭВМ CDC 6600 Сеймуром Креем (Seymour Cray), а на персональных ЭВМ в процессоре Intel Pentium Pro только в 1995 году. Отметим, что такая техника переупорядочивания операций практически не применима для *вещественных* чисел, так как операции над ними в дискретной матема-

тике не ассоциативны, то есть в общем случае `(x+y)+z≠x+(y+z)`. Блок OoO требует десятки миллионов вентилей и потребляет много энергии, поэтому в простых и экономичных процессорах (например, Intel Atom) он может не реализовываться (или банально отключаться). Любопытно, что на реализацию блока OoO требуется больше вентилей, чем на собственно выполнение операций.

Обычно, когда процессор переупорядочивает команды для из внеочередного выполнения, невозможно переставить операцию чтения из памяти (Load) перед записью (Store) из-за возможной зависимости по данным. Современные процессоры, однако, пытаются устранить эту зависимость с помощью особой технологии под трудно переводимым названием Memory Disambiguation. По существу, это спекулятивное чтение данных «впрок», которые, возможно, в дальнейшем и не понадобятся. Этот метод, как и спекулятивное выполнение команд, может повысить скорость счета программы на 5-20%, однако при плохой реализации потенциально опасен. В частности, именно эту технологию используют аппаратные уязвимости Spectre и Meltdown, о которых уже упоминалось ранее.

Попытка устранить эти уязвимости, отключив Memory Disambiguation, естественно, сразу снижает скорость выполнения программ. Современные версии ОС Линукс дают выполняющейся программе самой выбирать, стоит ли выключать спекулятивное чтение данных.

---

ix Для продвинутых читателей. Отметим, что на *векторных* AVX2 регистрах можно выполнять 3-адресные операции над, например, 4-элементными векторами вещественных чисел одинарной и двойной точности (нужен Ассемблер MASM 10.xx):

```
.data
  org 1000h
X zmmq (?); var X:array[0..7] of double (dq)
Y zmmq (?); var Y:array[0..7] of double
.code
  vmovups      zmm0,X;       zmm0:=X[7..0]
; vmovdqa      zmm0,X;       zmm0:=m512
  vaddps       zmm1,zmm0,Y;  zmm1:=zmm0+Y=X+Y
  vmovaps      X,zmm1;       X:=X+Y
  vfmadd231ps  zmm0,X,zmm1;  zmm0:=X*zmm1+zmm0
; 231 => op2*op3+op1; 321 => op3*op2+op1
; 123 => op1*op2+op3 и т.д.
  vmovaps      X,zmm0;       X:=X*(X+Y)+X
  vmovNTda     X,ymm0;       Запись 64 байт, минуя кэш
```

Как всегда, не надо забывать о перевёрнутом представлении данных в памяти. Обратите внимание на команду **`vfmadd231pd`**, она относится к так называемым 3-х адресным командам FMA (Fused Multiply-Add), реализующим 3-адресные команды вида `A:=A*B+C`, а есть и 4-адресные команды для операций вида `D:=A*B+C`. Одним из преимуществ таких "составных" команд является отсутствие промежуточного округления после умножения `A*B`, что позволяет поднять точность вычислений, например, пусть

```
var A,B,C,D: int64;
    . . .
    A:=1+1E-52; B:=1-1E-52; C:=-1;
  { D:=A*B+C }
    D:=A*B; {D=1!} D:=D+C; {D=0!}
```

Однако, если операцию `D:=A*B+C` выполнить как векторную команду **`vpmacsdd`** `D,A,B,C`, то будет $D:=A*B+C=2^{-103}$.

На векторном регистре возможны также так называемые *горизонтальные* операции, например, горизонтальная сумма векторов X и Y:

```
X  dq N dup(?); X: array[0..N-1] of double;
Y  dq N dup(?); Y: array[0..N-1] of double;
   vmovapd ymm1,X; ymm1:=X[3..0] - перевёрнутое
   vhaddpd ymm0,ymm1,Y
;  ymm0:={X[3]+X[2],X[1]+X[0],Y[3]+Y[2],Y[1]+Y[0]}
```

Любопытно, что, как и «обычные» вещественные регистры `st0-st7`, векторные регистры могут быть «пустыми», не содержащими чисел, работать с такими регистрами нельзя. А вот пример операции над двумя целочисленных векторами в упоминавшемся ранее режиме с насыщением:

```
;   Multiply Accumulate with Saturation Doubleword to Doubleword [m64]
```

```
    vpmasdd ymm0,ymm1,ymm2,ymm3/m256
;   for i:=3 to 0 do ymm0[i]:=ymm1[i]*ymm2[i]+ymm3/m256[i]
```

Это новый мир удивительных возможностей для работы со сложными структурами данных (векторами, матрицами, изображениями, строками и т.д.) для вещественной (и целочисленной!) арифметики, см. главу 17.

[x] Для продвинутых читателей. На первый взгляд, это невозможно. Действительно, пока за первый такт мы не определили границу первой команды, нельзя начинать декодирование следующей, и получается только по *одной* декодированной команде за такт, а вовсе не по четыре. Чтобы справиться с этой проблемой, конструкторы применили метод «грубой силы». Каждый такт в буфере длиной в очередные 32 байта программы работают 32 схемы декодирования (так называемые «длиномеры» (instruction length decoder), каждая, начиная со своего байта. В результате каждый такт получаются 32 «возможные» команды.

Другие схемы на каждом такте занимаются отбраковкой неверных команд. Например, если длина первой команды оказалась 5 байт, то на втором такте отбраковываются 4 команды, которые предположительно начинались в границах первой «настоящей» команды, и выполняется вторая «настоящая» команда. Пусть средняя длина команды 6-8 байт, тогда за 5-6 первых тактов у нас останется 4-5 «верных» команд. А далее нужно понять, что на *каждом* следующем такте тоже будет добавляться по 4-5 правильных команд. Конечно, работа немного замедляется, если команда переходит границы 32-байтных участков декодирования.

Это *по результату* чем-то похоже на работу конвейера, но с совершенно другим исполнением. Понятно, что на этот сложный «алгоритм в кремнии» требуется очень много вентилей.