

Современные десктопные процессоры архитектуры x86: общие принципы работы (x86 CPU FAQ 1.0)

Disclaimer

Наверное, наиболее точно причину появления данного материала можно сформулировать так: «даже не столько нужно, чтобы он был — сколько странно, что его до сих пор не было». И действительно: в комментариях к результатам тестов, мы постоянно оперируем такими понятиями как «ёмкость кэша», «скорость процессорной шины», «поддержка расширенных наборов инструкций», но единой статьи, в которой были бы собраны разъяснения всех этих терминов — на сайте нет. Такое упущение, разумеется, следовало устранить. Данная статья с подзаголовком «x86 CPU FAQ» и является попыткой сделать это. Разумеется, некоторые её разделы могут быть отнесены не только к процессорам архитектуры x86, и не только с десктопным (предназначенным для установки в ПК) их вариантам, однако вот уж на такой глобализм мы совершенно точно не замахиваемся. Поэтому просьба помнить, что в рамках данного материала, если в явной форме не указано иное, слово «процессор» обозначает «процессор архитектуры x86, предназначенный для установки в десктопы». Возможно, в процессе дальнейшего совершенствования и расширения, появятся в статье разделы, посвящённые **серверным** CPU или даже процессорам других архитектур, но это уже дело будущего...

Оглавление

- [Введение](#)
 - [Код и данные: основной принцип работы процессора](#)
- [Общие принципы взаимодействия процессора и ОЗУ](#)
 - [Контроллер памяти](#)
 - [Процессорная шина](#)
 - [Различия между традиционной архитектурой и K8/AMD64](#)
 - [Оперативная память](#)
 - [Разрядность шины памяти, N-канальные контроллеры памяти](#)
 - [Скорость чтения и записи](#)
 - [Латентность](#)
- [Процессор: сведения общего характера](#)
 - [Понятие архитектуры](#)
 - [Архитектура как совместимость с кодом](#)
 - [Архитектура как характеристика семейства процессоров](#)
 - [64-битные расширения классической x86 \(IA32\) архитектуры](#)
 - [Процессорное ядро](#)
 - [Различия между ядрами одной микроархитектуры](#)
 - [Ревизии](#)

- [Частота работы ядра](#)
- [Особенности образования названий процессоров](#)
 - [Рейтинги от AMD](#)
 - [Processor Number от Intel](#)
- [Измерение скорости «в мегагерцах» — как это возможно?](#)
 - [Пара слов о некоторых пикантных особенностях DDR и QDR протоколов](#)
- [Процессор «крупноблочно»](#)
 - [Кэш](#)
 - [Общее описание и принцип действия](#)
 - [Многоуровневое кэширование](#)
 - [Декодер](#)
 - [Исполняющие \(функциональные\) устройства](#)
 - [Арифметико-логические устройства](#)
 - [Блок вычислений с плавающей запятой](#)
 - [Регистры процессора](#)
- [Процессор в подробностях](#)
 - [Особенности кэшей](#)
 - [Частота работы кэша и его шина](#)
 - [Эксклюзивный и неэксклюзивный кэш](#)
 - [Trace cache](#)
 - [Суперскалярность и внеочередное исполнение команд](#)
 - [Предварительное \(опережающее\) декодирование и кэширование](#)
 - [Предсказание ветвлений](#)
 - [Предвыборка данных](#)
- [Заключение](#)

Введение

Код и данные: основной принцип работы процессора

Итак, если не пытаться изложить здесь «кратенько» курс информатики для средней школы, то единственное что хотелось бы напомнить — это то, что процессор (за редкими исключениями) исполняет не программы, написанные на каком-нибудь языке программирования (один из которых, вы, возможно, даже знаете), а некий «машинный код». То есть командами для него являются последовательности байтов, находящихся в памяти компьютера. Иногда команда может быть равна одному байту, иногда она занимает несколько байт. Там же, в основной памяти (ОЗУ, RAM) находятся и данные. Они могут находиться в отдельной области, а могут и быть «перемешаны» с кодом. Различие между кодом и данными состоит в том, что данные — это то, *над чем* процессор производит какие-то операции. А код — это команды, которые ему сообщают, *какую именно операцию* он должен произвести. Для упрощения, мы можем представить себе программу и ее данные в виде последовательности байтов некой конечной длины, располагающуюся непрерывно (не будем усложнять) в общем массиве памяти. Например, у нас есть массив памяти длиной в 1'000'000 байт, а наша программа (вместе с данными) — это байты с номерами от 1000 до 20'000. Прочие байты — это другие программы или их данные, или просто свободная память, не занятая ничем полезным.

Таким образом, «машинный код» — это команды процессора, располагающиеся в памяти. Там же располагаются данные. Для того чтобы исполнить команду, процессор должен

прочитать ее из памяти. Для того чтобы произвести операцию над данными, процессор должен прочитать их из памяти, и, возможно, после произведения над ними определенного действия, записать их обратно в память в обновленном (измененном) виде. Команды и данные идентифицируются их адресом, который, по сути, представляет собой порядковый номер ячейки памяти.

Общие принципы взаимодействия процессора и ОЗУ

Возможно, кого-то удивит, что достаточно большой раздел в FAQ, посвященном x86 CPU, выделен под объяснение особенностей функционирования памяти в современных системах, основанных на данном типе процессоров. Однако факты — упрямая вещь: сами x86-процессоры ныне содержат так много блоков, отвечающих именно за оптимизацию их работы с ОЗУ, что игнорировать эту тесную связь было бы совершенно нелепо. Можно сказать даже так: уж, коль решения, связанные с оптимизацией работы с памятью, стали неотъемлемой частью самих процессоров — то и саму память можно рассматривать в качестве некоего «придатка», функционирование которого оказывает непосредственное влияние на скорость работы CPU. Без понимания особенностей взаимодействия процессора с памятью, невозможно понять, за счёт чего тот или иной процессор (та или иная система) исполняет программы медленнее или быстрее.

Контроллер памяти

Итак, ранее выше мы уже говорили о том, что как команды, так и данные, попадают в процессор из оперативной памяти. На самом деле всё немного сложнее. В большинстве современных x86-систем (то есть компьютеров на базе x86-процессоров), процессор как устройство к памяти обращаться вообще не может, так как не имеет в своем составе соответствующих узлов. Поэтому он обращается к «промежуточному» специализированному устройству, называемому контроллером памяти, а уже тот, в свою очередь — к микросхемам ОЗУ, размещенным на модулях памяти. Модули вы наверняка видели — это такие длинные узкие текстолитовые «планочки» (фактически — небольшие платы) с некоторым количеством микросхем на них, вставляемые в специальные разъемы на системной плате. Роль контроллера ОЗУ, таким образом, проста: он служит своего рода «мостом»* между памятью и использующими ее устройствами (кстати, к ним относится не только процессор, но об этом — чуть позже). Как правило, контроллер памяти входит в состав чипсета — набора микросхем, являющегося основой системной платы. От быстродействия контроллера во многом зависит скорость обмена данными между процессором и памятью, это один из важнейших компонентов, влияющих на общую производительность компьютера.

* — кстати, контроллер памяти физически находится в микросхеме чипсета, традиционно называемой «северным мостом».

Процессорная шина

Любой процессор обязательно оснащён процессорной шиной, которую в среде x86 CPU принято называть FSB (Front Side Bus). Эта шина служит каналом связи между процессором и всеми остальными устройствами в компьютере: памятью, видеокартой, жёстким диском, и так далее. Впрочем, как мы уже знаем из предыдущего раздела, между собственно памятью и процессором находится контроллер памяти. Соответственно: процессор посредством FSB связывается с контроллером памяти, а уже тот, в свою очередь, по специальной шине (назовём её, не мудрствуя лукаво, «шиной памяти») — с модулями ОЗУ на плате. Однако,

повторимся: поскольку «внешняя» шина у классического x86 CPU всего одна, она используется не только для работы с памятью, но и для общения процессора со всеми остальными устройствами.

Различия между традиционной для x86 CPU архитектурой и K8/AMD64

Революционность подхода компании AMD состоит в том, что её процессоры с архитектурой AMD64 (и микроархитектурой, которую условно принято называть «K8») оснащены множеством «внешних» шин. При этом одна или несколько шин HyperTransport служат для связи со всеми устройствами кроме памяти, а отдельная группа из одной или двух (в случае двухканального контроллера) шин — исключительно для работы процессора с памятью. Преимущество интеграции контроллера памяти прямо в процессор, очевидно: «путь от ядра до памяти» становится заметно «короче», что позволяет работать с ОЗУ быстрее. Правда, имеются у данного подхода и недостатки. Так, например, если ранее устройства типа жёсткого диска или видеокарты могли работать с памятью через выделенный, независимый контроллер — то в случае с архитектурой AMD64 они вынуждены работать с ОЗУ через контроллер, размещённый на процессоре. Так как CPU в данной архитектуре является единственным устройством, имеющим прямой доступ к памяти. Де-факто, в противостоянии «внешний контроллер vs. интегрированный», сложился паритет: с одной стороны, на данный момент AMD является единственным производителем десктопных x86-процессоров с интегрированным контроллером памяти, с другой — компания вроде бы вполне довольна этим решением, и не собирается от него отказываться. С третьей — Intel тоже не собирается отказываться от внешнего контроллера, и вполне довольна «классической схемой», проверенной годами.

Оперативная память

Разрядность шины памяти, N-канальные контроллеры памяти

По состоянию на сегодняшний день, вся память, используемая в современных десктопных x86-системах имеет шину шириной 64 бита. Это означает, что за один такт по данной шине одновременно может быть передано количество информации, кратное 8 байтам (8 байт для SDR-шин, 16 байт для DDR-шин). Особняком стоит только память типа RDRAM, применявшаяся в системах на базе процессоров Intel Pentium 4 на заре становления архитектуры NetBurst, но сейчас это направление признано тупиковым для x86-ПК (к слову — руку к этому приложила всё та же компания Intel, которая в своё время активно пропагандировала данный тип памяти). Некоторую неразбериху вносят лишь двухканальные контроллеры, обеспечивающие одновременную работу с двумя отдельными друг от друга 64-битными шинами, благодаря чему некоторые производители заявляют о некой «128-битности». Это, разумеется, чистой воды профанация. Арифметика на уровне 1-го класса в данном случае, увы, не работает: 2×64 вовсе не равно 128. Почему? Да хотя бы потому, что даже самые современные x86 CPU (см. ниже раздел FAQ «64-битные расширения классической x86 (IA32) архитектуры») не могут работать со 128-битной шиной и 128-битной адресацией. Грубо говоря: две независимые параллельно идущие дороги, шириной 2 метра каждая — могут обеспечить одновременный проезд двух автомобилей, шириной 2 метра — но никоим образом не одного, шириной 4 метра. Точно так же, N-канальный контроллер памяти может увеличить скорость работы с данными в N раз (и то больше теоретически, чем практически) — но никак не способен увеличить разрядность этих данных. Ширина шины памяти во всех современных контроллерах, применяемых в x86-системах, равна 64 битам — независимо от того, находится этот контроллер в чипсете, или в самом процессоре. Некоторые контроллеры оснащены двумя независимыми 64-битными

каналами, но на разрядность шины памяти это никак не влияет — только на скорость считывания и записи информации.

Скорость чтения и записи

Скорость чтения и записи информации в память теоретически ограничивается исключительно пропускной способностью самой памяти. Так, например, двухканальный контроллер памяти стандарта DDR400 теоретически способен обеспечить скорость чтения и записи информации, равную $8 \text{ байт (ширина шины)} * 2 \text{ (количество каналов)} * 2 \text{ (протокол DDR, обеспечивающий передачу 2 пакетов данных за 1 такт)} * 200'000'000 \text{ (фактическая частота работы шины памяти равная 200 МГц, то есть 200'000'000 тактов в секунду)}$. Значения, получаемые в результате практических тестов, как правило, чуть ниже теоретических: сказывается «неидеальность» конструкции контроллера памяти, плюс накладки (задержки), вызванные работой подсистемы кэширования самого процессора (см. ниже раздел про процессорный кэш). Однако основной «подвох» содержится даже не в накладках, связанных с работой контроллера и подсистемы кэширования, а в том, что скорость «линейного» чтения или записи является вовсе не единственной характеристикой, влияющей на фактическую скорость работы процессора с ОЗУ. Для того чтобы понять, из каких составляющих складывается фактическая скорость работы процессора с памятью, нам необходимо кроме линейной скорости считывания или записи учитывать ещё и такую характеристику, как *латентность*.

Латентность

Латентность является не менее важной характеристикой с точки зрения быстродействия подсистемы памяти, чем скорость «прокачки данных», но совершенно другой, по сути. Большая скорость обмена данными хороша тогда, когда их размер относительно велик, но если нам требуется «понемногу с разных адресов» — то на первый план выходит именно латентность. Что это такое? В общем случае — время, которое требуется для того, чтобы начать считывать информацию с определённого адреса. И действительно: с момента, когда процессор посылает контроллеру памяти команду на считывание (запись), и до момента, когда эта операция осуществляется, проходит определённое количество времени. Причём оно вовсе не равно времени, которое требуется на пересылку данных. Приняв команду на чтение или запись от процессора, контроллер памяти «указывает» ей, с каким адресом он желает работать. Доступ к любому произвольно взятому адресу не может быть осуществлён мгновенно, для этого требуется определённое время. Возникает задержка: адрес указан, но память ещё не готова предоставить к нему доступ. В общем случае, эту задержку и принято называть латентностью. У разных типов памяти она разная. Так, например, память типа DDR2 имеет в среднем гораздо большие задержки, чем DDR (при одинаковой частоте передачи данных). В результате, если данные в программе расположены «хаотично» и «небольшими кусками», скорость их считывания становится намного менее важной, чем скорость доступа к «началу куска», так как задержки при переходе на очередной адрес влияют на быстродействие системы намного сильнее, чем скорость считывания или записи.

«Соревнование» между скоростью чтения (записи) и латентностью — одна из основных головных болей разработчиков современных систем: к сожалению, рост скорости чтения (записи), почти всегда приводит к увеличению латентности. Так, например, память типа SDR (PC66, PC100, PC133) обладает в среднем лучшей (меньшей) латентностью, чем DDR. В свою очередь, у DDR2 латентность ещё выше (то есть хуже), чем у DDR.

Следует понимать, что «общая» латентность подсистемы памяти зависит не только от неё самой, но и от контроллера памяти и места его расположения — все эти факторы тоже влияют на задержку. Именно поэтому компания AMD в процессе разработки архитектуры AMD64 решила «одним махом» решить проблему высокой латентности, интегрировав

контроллер прямо в процессор — чтобы максимально «сократить дистанцию» между процессорным ядром и модулями ОЗУ. Затея удалась, но дорогой ценой: теперь система на базе определённого CPU архитектуры AMD64 может работать только с той памятью, на которую рассчитан его контроллер. Наверное, именно поэтому компания Intel до сих пор не решилась на такой кардинальный шаг, предпочитая действовать традиционными методами: усовершенствуя контроллер памяти в чипсете и механизм Prefetch в процессоре (про него см. ниже).

В завершение, заметим, что понятия «скорость чтения / записи» и «латентность», в общем случае, применимы к любому типу памяти — в том числе не только к классической DRAM (SDR, Rambus, DDR, DDR2), но и к кэшу (см. ниже).

Процессор: сведения общего характера

Понятие архитектуры

Архитектура как совместимость с кодом

Наверняка вы часто встречались с термином «x86», или «Intel-совместимый процессор» (или «IBM PC compatible» — но это уже по отношению к компьютеру). Иногда также встречается термин «Pentium-совместимый» (почему именно Pentium — вы поймете сами чуть позже). Что за всеми этими названиями скрывается на самом деле? На данный момент наиболее корректно с точки зрения автора выглядит следующая простая формулировка: *современный x86-процессор — это процессор, способный корректно исполнять машинный код архитектуры IA32 (архитектура 32-битных процессоров Intel)*. В первом приближении это код, исполняемый процессором i80386 (известным в народе как «386-й»), окончательно же основной набор команд IA32 сформировался с выходом процессора Intel Pentium Pro. Что означает «основной набор» и какие есть еще? Для начала ответим на первую часть вопроса. «Основной» в данном случае означает то, что с помощью исключительно этого набора команд, может быть написана любая программа, которая вообще может быть написана для процессора архитектуры x86 (или IA32, если вам так больше нравится).

Кроме того, у архитектуры IA32 существуют «официальные» расширения (дополнительные наборы команд) от разработчика самой архитектуры, компании Intel: MMX, SSE, SSE2 и SSE3. Также существуют «неофициальные» (не от Intel) расширенные наборы команд: EMMX, 3DNow! и Extended 3DNow! — их разработала компания AMD. Впрочем, «официальность» и «неофициальность» в данном случае понятие относительное — де-факто все сводится к тому, что некоторые расширения набора команд Intel как разработчик изначального набора признает, а некоторые — нет, разработчики же программного обеспечения используют то, что им лучше всего подходит. В отношении расширенных наборов команд существует одно простое правило хорошего тона: прежде чем их использовать, программа должна проверить, поддерживает ли их процессор. Иногда отступления от этого правила встречаются (и могут приводить к неправильному функционированию программ), но объективно это является проблемой некорректно написанного программного обеспечения, а не процессора.

Для чего предназначены дополнительные наборы команд? В первую очередь — для увеличения быстродействия при выполнении некоторых операций. Одна команда из дополнительного набора, как правило, выполняет действие, для которого понадобилась бы небольшая программа, состоящая из команд основного набора. Опять-таки, как правило, одна команда выполняется процессором быстрее, чем заменяющая ее последовательность. Однако в 99% случаев, ничего такого, чего нельзя было бы сделать с помощью основных

команд, с помощью команд из дополнительного набора сделать нельзя. Таким образом, упомянутая выше проверка программой поддержки дополнительных наборов команд процессором, должна выполнять очень простую функцию: если, например, процессор поддерживает SSE — значит, считать будем быстро и с помощью команд из набора SSE. Если нет — будем считать медленнее, с помощью команд из основного набора. Корректно написанная программа обязана действовать именно так. Впрочем, сейчас практически никто не проверяет у процессора наличие поддержки MMX, так как все CPU, вышедшие за последние 5 лет, этот набор поддерживают гарантированно. Для справки приведем табличку, на которой обобщена информация о поддержке различных расширенных наборов команд различными десктопными (предназначенными для настольных ПК) процессорами.

Процессор	MMX	EMMX	3DNow!	SSE	E3DNow!	SSE2	SSE3
Intel Pentium II	+	—	—	—	—	—	—
Intel Celeron до 533 MHz	+	—	—	—	—	—	—
Intel Pentium III	+	—	—	+	—	—	—
Intel Celeron 533—1400 MHz	+	—	—	+	—	—	—
Intel Pentium 4	+	—	—	+	—	+	+/—*
Intel Celeron от 1700 MHz	+	—	—	+	—	+	—
Intel Celeron D	+	—	—	+	—	+	+
Intel Pentium 4 eXtreme Edition	+	—	—	+	—	+	+/—*
Intel Pentium eXtreme Edition	+	—	—	+	—	+	+
Intel Pentium D	+	—	—	+	—	+	+
AMD K6	+	+	—	—	—	—	—
AMD K6-2	+	+	+	—	—	—	—
AMD K6-III	+	+	+	—	—	—	—
AMD Athlon	+	+	+	—	+	—	—
AMD Duron до 900 MHz	+	+	+	—	+	—	—
AMD Athlon XP	+	+	+	+	+	—	—
AMD Duron от 1000 MHz	+	+	+	+	+	—	—
AMD Athlon 64 / Athlon FX	+	+	+	+	+	+	+/—*
AMD Sempron	+	+	+	+	+	+/—*	+/—*
AMD Athlon 64 X2	+	+	+	+	+	+	+
VIA C3	+	+	+/—*	+/—*	—	—	—

* в зависимости от модификации

На данный момент всё популярное десктопное программное обеспечение (операционные системы **Windows** и **Linux**, офисные пакеты, компьютерные игры, и прочее) разрабатывается именно для x86-процессоров. Оно выполняется (за исключением «дурно воспитанных» программ) на любом x86-процессоре, независимо от того, кто его произвел. Поэтому вместо ориентированных на разработчика изначальной архитектуры терминов «Intel-совместимый» или «Pentium-совместимый», стали употреблять нейтральное название: «x86-совместимый процессор», «процессор с архитектурой x86». В данном случае под «архитектурой» понимается совместимость с определённым набором команд, то есть, можно сказать, «архитектура процессора с точки зрения программиста». Есть и другая трактовка того же термина.

Архитектура как характеристика семейства процессоров

«Железячники» — люди, работающие в основном не с программным обеспечением, а с аппаратным, под «архитектурой» понимают несколько другое (правда, более корректно то, что они называют «архитектурой», называется «микроархитектурой», но де-факто приставку «микро» частенько опускают). Для них «архитектура CPU» — это некий набор свойств, присущий целому семейству процессоров, как правило, выпускаемому в течение многих лет (иначе говоря — «внутренняя конструкция», «организация» этих процессоров). Так, например, любой специалист по x86 CPU вам скажет, что процессор с ALU, работающими на удвоенной частоте, QDR-шиной, Trace cache, и, возможно, поддержкой технологии Hyper-Threading — это «процессор архитектуры NetBurst» (не пугайтесь незнакомых терминов — все они будут разъяснены чуть позже). А процессоры Intel Pentium Pro, Pentium II и Pentium III — это «архитектура P6». Таким образом, понятие «архитектуры» применительно к процессорам несколько двойственно: под ним может пониматься как совместимость с неким единым набором команд, так и совокупность аппаратных решений, присущих определённой достаточно широкой группе процессоров. Разумеется, такой дуализм одного из основополагающих понятий не очень удобен, однако так уж сложилось, и вряд ли в ближайшее время что-то поменяется...

64-битные расширения классической x86 (IA32) архитектуры

Не так давно оба ведущих производителя x86 CPU анонсировали две практически идентичных* технологии (впрочем, AMD предпочитает называть это архитектурой), благодаря которым классические x86 (IA32) CPU получили статус 64-битных. В случае с AMD данная технология получила наименование «AMD64» (64-битная архитектура AMD), в случае с Intel — «EM64T» (расширенная 64-битная технология работы с памятью). Также почтенные аксакалы, знакомые с историей вопроса, иногда употребляют наименование «x86-64» — как общее обозначение всех 64-битных расширений архитектуры x86, не привязанное к зарегистрированным торговым маркам какого-либо производителя. Де-факто, употребление одного из трёх, приведенных выше, наименований, зависит больше от личных предпочтений употребляющего, чем от фактических различий — ибо различия между AMD64 и EM64T уместаются на кончике очень тонкой иглы. К тому же, сама AMD ввела «фирменное» наименование «AMD64» лишь незадолго до анонса собственных процессоров на основе данной архитектуры, а до этого совершенно спокойно употребляла в собственных документах более нейтральное «x86-64». Однако так или иначе, всё сводится к одному: некоторые внутренние регистры процессоров стали вместо 32-битных 64-битными, 32-битные команды x86-кода получили свои 64-битные аналоги, кроме того, объём адресуемой памяти (включая не только физическую, но и виртуальную) многократно увеличился (за счёт того, что адрес приобрёл вместо 32-битного 64-битный формат). Количество маркетинговых спекуляций на тему «64-битности» превысило все разумные пределы, поэтому нам следует рассмотреть достоинства данного нововведения особенно пристально. Итак: что же на самом деле изменилось, а что — нет?

* — Доводы о том, что Intel, дескать, «нагло скопировала EM64T с AMD64» не выдерживают никакой критики. И вовсе не потому, что это не так — а потому, что вовсе не «нагло». Есть такое понятие: «кросс-лицензионное соглашение». Если таковое соглашение имеет место быть, это означает, что все разработки одной компании в определённой области, становятся автоматически доступными другой, равно как и разработки другой автоматически становятся доступны первой. Intel воспользовалась кросс-лицензированием для разработки EM64T, взяв за основу AMD64 (чего никто никогда не отрицал). AMD воспользовалась тем же соглашением для введения в свои процессоры поддержки наборов дополнительных инструкций SSE2 и SSE3, разработанных Intel. И ничего в этом постыдного нет: раз договорились «делиться» разработками — значит, надо

делиться.

Что не изменилось? В первую очередь — быстродействие процессоров. Вопиющей глупостью будет считать, что один и тот же процессор при переходе из привычного 32-битного в 64-битный режим (а 32-битный режим все нынешние x86 CPU поддерживают в обязательном порядке) станет работать в 2 раза быстрее. Разумеется, в некоторых случаях некое ускорение от использования 64-битной целочисленной арифметики может присутствовать — но количество этих случаев сильно ограничено, и большинства современного пользовательского программного обеспечения они никак не касаются. Кстати: а почему мы употребили термин «64-битная целочисленная арифметика»? А потому, что блоки операций с плавающей точкой (см. ниже) во всех x86-процессорах уже давным-давно не 32-битные. И даже не 64-битные. Классический x87 FPU (см. ниже), окончательно ставший частью CPU ещё во времена старого доброго 32-битного Intel Pentium — **уже был 80-битным**. Операнды команд SSE и SSE2/3 — и вовсе 128-битные! В этом плане архитектура x86 достаточно парадоксальна: при всём притом, что формально процессоры данной архитектуры достаточно долгое время оставались 32-битными — разрядность тех блоков, где «большая битность» была реально необходима — наращивалась совершенно независимо от остальных. Например, процессоры AMD Athlon XP и Intel Pentium 4 «Northwood» совмещали в себе блоки, работающие с 32-битными, 80-битными, и 128-битными операндами. 32-битными оставались лишь основной набор команд (унаследованный от первого процессора архитектуры IA32 — Intel 386) и адресация памяти (максимум 4 гигабайта, если не считать «извращений» типа Intel PAE).

Таким образом, то, что процессоры AMD и Intel стали «формально 64-битными», на практике принесло нам лишь три усовершенствования: появление команд для работы с 64-битными целыми числами, увеличение количества и/или разрядности регистров, и увеличение максимального объёма адресуемой памяти. Заметим: реальной пользы этих нововведений (особенно третьего!) никто не отрицает. Равно как никто не отрицает заслуг компании AMD в продвижении идеи «осовременивания» (за счёт введения 64-битности) x86-процессоров. Мы лишь хотим предостеречь от чрезмерных ожиданий: не стоит надеяться на то, что компьютер, покупавшийся «в ценовом классе ВАЗа», от установки 64-битного программного обеспечения станет «лихим Мерседесом». Чудес на свете не бывает...

Процессорное ядро

Различия между ядрами одной микроархитектуры

«Процессорное ядро» (как правило, для краткости его называют просто «ядро») — это конкретное воплощение [микро]архитектуры (т.е. «архитектуры в аппаратном смысле этого слова»), являющееся стандартом для целой серии процессоров. Например, NetBurst — это микроархитектура, которая лежит в основе многих сегодняшних процессоров Intel: Celeron, Pentium 4, Xeon. Микроархитектура задает общие принципы: длинный конвейер, использование определенной разновидности кэша кода первого уровня (Trace cache), прочие «глобальные» особенности. Ядро — более конкретное воплощение. Например, процессоры микроархитектуры NetBurst с шиной 400 МГц, кэшем второго уровня 256 килобайт, и без поддержки Hyper-Threading — это более-менее полное описание ядра Willamette. А вот ядро Northwood имеет кэш второго уровня уже 512 килобайт, хотя также основано на NetBurst. Ядро AMD Thunderbird основано на микроархитектуре K7, но не поддерживает набор команд SSE, а вот ядро Palomino — уже поддерживает.

Таким образом, можно сказать что «ядро» — это конкретное воплощение определенной микроархитектуры «в кремнии», обладающее (в отличие от самой микроархитектуры) определенным набором строго обусловленных характеристик. Микроархитектура —

аморфна, она описывает общие принципы построения процессора. Ядро — конкретно, это микроархитектура, «обросшая» всевозможными параметрами и характеристиками. Чрезвычайно редки случаи, когда процессоры сменяли микроархитектуру, сохраняя название. И, наоборот, практически любое наименование процессора хотя бы несколько раз за время своего существования «меняло» ядро. Например, общее название серии процессоров AMD — «Athlon XP» — это одна микроархитектура (K7), но целых четыре ядра (Palomino, Thoroughbred, Barton, Thorton). Разные ядра, построенные на одной микроархитектуре, могут иметь, в том числе разное быстродействие.

Ревизии

Ревизия — одна из модификаций ядра, крайне незначительно отличающаяся от предыдущей, почему и не заслуживает звания «нового ядра». Как правило, из выпусков очередной ревизии производители процессоров не делают большого события, это происходит «в рабочем порядке». Так что даже если вы покупаете один и тот же процессор, с полностью аналогичным названием и характеристиками, но с интервалом где-то в полгода — вполне возможно, фактически он будет уже немного другой. Выпуск новой ревизии, как правило, связан с какими-то мелкими усовершенствованиями. Например, удалось чуть-чуть снизить энергопотребление, или понизить напряжение питания, или еще что-то оптимизировать, или была устранена пара мелких ошибок. С точки зрения производительности мы не помним ни одного примера, когда бы одна ревизия ядра отличалась от другой настолько существенно, чтобы об этом имело смысл говорить. Хотя чисто теоретически возможен и такой вариант — например, подвергся оптимизации один из блоков процессора, ответственный за исполнение нескольких команд. Подводя итог, можно сказать что «замораживаться» ревизиями процессоров чаще всего не стоит: в очень редких случаях изменение ревизии вносит какие-то кардинальные изменения в процессор. Достаточно просто знать, что есть такая штука — исключительно для общего развития.

Частота работы ядра

Как правило, именно этот параметр в просторечии именуют «частотой процессора». Хотя в общем случае определение «частота работы ядра» всё же более корректно, так как совершенно не обязательно все составляющие CPU функционируют на той же частоте, что и ядро (наиболее частым примером обратного являлись старые «слотовые» x86 CPU — Intel Pentium II и Pentium III для Slot 1, AMD Athlon для Slot A — у них L2-кэш функционировал на 1/2, и даже иногда на 1/3 частоты работы ядра). Ещё одним распространённым заблуждением является уверенность в том, что частота работы ядра однозначным образом определяет производительность. На самом деле это дважды не так: во-первых, каждое конкретное процессорное ядро (в зависимости от того, как оно спроектировано, сколько содержит исполняющих блоков различных типов, и т.д. и т.п.) может исполнять различное количество команд за один такт, частота же — это всего лишь количество таких тактов в секунду. Таким образом (приведенное далее сравнение, разумеется, очень сильно упрощено и поэтому весьма условно) процессор, ядро которого исполняет 3 инструкции за такт, может иметь на треть меньшую частоту, чем процессор, исполняющий 2 инструкции за такт — и при этом обладать полностью аналогичным быстродействием.

Во-вторых, даже в рамках одного и того же ядра, увеличение частоты вовсе не всегда приводит к пропорциональному увеличению быстродействия. Здесь вам очень пригодятся знания, которые вы могли почерпнуть из раздела «Общие принципы взаимодействия процессора и ОЗУ». Дело в том, что скорость исполнения команд ядром процессора — это вовсе не единственный показатель, влияющий на скорость выполнения программы. Не менее важна скорость поступления команд и данных на CPU. Представим себе чисто теоретически такую систему: быстродействие процессора — 10'000 команд в секунду, скорость работы

памяти — 1000 байт в секунду. Вопрос: даже если принять, что одна команда занимает не более одного байта, а данных у нас нет совсем, с какой скоростью будет исполняться программа в такой системе? Правильно: не более 1000 команд в секунду, и производительность CPU тут совершенно ни при чём: мы будем ограничены не ей, а скоростью поступления команд в процессор. Таким образом, следует понимать: невозможно непрерывно наращивать одну только частоту ядра, не ускоряя одновременно подсистему памяти, так как в этом случае начиная с определённого этапа, увеличение частоты CPU перестанет сказываться на увеличении быстродействия системы в целом.

Особенности образования названий процессоров

Раньше, когда небо было голубее, пиво — вкуснее, а девушки — красивее, процессоры называли просто: имя производителя + название модельного ряда + частота. Например: «AMD K6-2 450 MHz». В настоящее время уже оба основных производителя от этой традиции отошли, и вместо частоты употребляют какие-то непонятные циферки, обозначающие неведь что. Краткому объяснению того, что же на самом деле эти циферки обозначают, и посвящены следующие два раздела.

Рейтинги от AMD

Причина, по которой компания AMD «изъяла» частоту из наименования своих процессоров, и заменила её некой абстрактной цифрой — общеизвестна: после появления процессора Intel Pentium 4, который работает на очень высоких частотах, процессоры AMD рядом с ним стали «плохо выглядеть на витрине» — покупатель не верил, что CPU с частотой, например, 1500 МГц, может обогнать CPU с частотой 2000 МГц. Поэтому частоту в наименовании заменили рейтингом. Формальная («де-юре», так сказать) трактовка этого рейтинга в устах AMD в разные времена звучала немного по-разному, но ни разу не прозвучала в том виде, в каком её воспринимали пользователи: процессор AMD с неким рейтингом, должен быть как минимум не медленнее процессора Intel Pentium 4 с соответствующей данному рейтингу частотой. Между тем, ни для кого не являлось особенным секретом, что именно такая трактовка и являлась конечной целью введения рейтинга. В общем, все всё прекрасно понимали, но AMD старательно делала вид, что она тут ни при чём :). Пенять ей за это не стоит: в конкурентной борьбе применяются совсем другие правила, чем в рыцарских поединках. Тем более что результаты независимых тестов демонстрировали: в целом, рейтинги своим процессорам AMD назначает достаточно справедливые. Собственно, именно до тех пор, пока это так — вряд ли имеет смысл протестовать против использования рейтинга. Правда, остаётся открытым один вопрос: а к чему же (нас интересует, понятное дело, состояние де-факто, а не разъяснения маркетингового отдела) будет привязан рейтинг процессоров AMD чуть позже, когда вместо Pentium 4 Intel начнёт выпускать какой-нибудь другой процессор?

Processor Number от Intel

Что нужно запомнить сразу: Processor Number (далее PN) у процессоров Intel — это не рейтинг. Не рейтинг производительности, и не рейтинг чего-либо другого. Фактически, это просто «артикул», элемент строчки в складской ведомости, единственная задача которого — сделать так, чтобы строчка, обозначающая один процессор, отличалась от строчки, обозначающей другой. В рамках серии (первая цифра PN), две остальные цифры, в принципе, кое о чём могут сказать, но, учитывая наличие таблиц, в которых приведено полное соответствие между PN и реальными параметрами, мы не видим особого смысла в том, чтобы заучивать какие-то промежуточные соответствия. Мотивация, которой руководствовалась Intel, вводя PN (вместо опять-таки указания частоты CPU) — более сложная, чем у AMD. Необходимость введения PN (как её объясняет сама Intel) связана,

прежде всего, с тем, что два основных конкурента по-разному подходят к вопросу об уникальности наименования CPU. Например, у AMD название «Athlon 64 3200+» может обозначать сразу четыре процессора с несколько различными техническими характеристиками (но одинаковым «рейтингом»). Intel придерживается мнения, что наименование процессора должно быть уникальным, в связи с чем ранее компании приходилось «изворачиваться», добавляя к значению частоты в наименовании различные буквы, и это приводило к путанице. По идее, PN должен был эту путаницу устранить. Трудно сказать, была ли достигнута поставленная цель: всё равно номенклатура процессоров Intel осталась достаточно сложной. С другой стороны, это неизбежно, так как ассортимент продуктов уж больно велик. Однако независимо от всего прочего, одного эффекта де-факто добиться точно удалось: теперь только разбирающиеся в вопросе специалисты могут по названию процессора быстро и точно «по памяти» сказать, что он собой представляет, и какова будет его производительность в сравнении с другими CPU. Насколько это хорошо? Сложно сказать. Мы предпочтём воздержаться от комментариев.

Измерение скорости «в мегагерцах» — как это возможно?

Никак это невозможно, потому что скорость не измеряется в мегагерцах, как не измеряется расстояние в килограммах. Однако господа маркетологи давно уже поняли, что в словесном поединке между физиком и психологом побеждает всегда последний — причём независимо от того, кто на самом деле прав. Поэтому мы и читаем про «сверхбыструю 1066 MHz FSB», мучительно пытаюсь понять, как скорость может измеряться с помощью частоты. На самом деле, раз уж прижилась такая извращённая тенденция, нужно просто чётко представлять себе, что имеется в виду. А имеется в виду следующее: если мы «закрепим» ширину шины на N битах — то её пропускная способность действительно будет зависеть от того, на какой частоте данная шина функционирует, и какое количество данных она способна передавать за такт. По обычной процессорной шине с «одинарной» скоростью (такая шина была, например, у процессора Intel Pentium III) за такт передаётся 64 бита, то есть 8 байт. Соответственно, если рабочая частота шины равна 100 МГц (100'000'000 тактов в секунду) — то скорость передачи данных будет равна 8 байт * 100'000'000 герц ≈ 763 мегабайта в секунду (а если считать в «десятичных мегабайтах», в которых принято считать *потоки данных*, то ещё красивее — 800 мегабайт в секунду). Соответственно, если на тех же 100 мегагерцах работает DDR-шина, способная передавать за один такт удвоенный объём данных — скорость вырастет ровно вдвое. Поэтому, согласно парадоксальной логике господ маркетологов, данную шину следует именовать «200-мегагерцевой». А если это ещё и QDR (Quad Data Rate) шина — то она и вовсе «400-мегагерцевая» получается, так как за один такт передаёт четыре пакета данных. *Хотя реальная частота работы у всех трёх вышеописанных шин одинаковая — 100 мегагерц.* Вот так «мегагерцы» и стали синонимом скорости.

Таким образом, QDR-шина (с «учетверённой» скоростью), работающая на реальной частоте 266 мегагерц, волшебным образом оказывается у нас «1066-мегагерцевой». Цифра «1066» в данном случае олицетворяет то, что её пропускная способность ровно в 4 раза больше «односкоростной» шины, работающей на той же самой частоте. Вы ещё не запутались?.. Привыкайте! Это вам не какая-нибудь теория относительности, тут всё намного сложнее и запутанней... Впрочем, самое главное здесь — выучить наизусть один простой принцип: если уж мы занимаемся таким извращением, как сравнение скорости двух шин между собой «в мегагерцах» — то они обязательно должны быть одинаковой ширины. Иначе получается как в одном форуме, где человек всерьёз доказывал, что пропускная способность AGP2X («133-мегагерцевая», но **32-битная** шина) — выше, чем пропускная способность FSB у Pentium III 800 (реальная частота 100 МГц, ширина **64 бита**).

Пара слов о некоторых пикантных особенностях DDR и QDR протоколов

Как уже было сказано выше, в режиме DDR по шине за один такт передаётся удвоенный объём информации, а в режиме QDR — учетверённый. Правда, в документах, ориентированных больше на прославление достижений производителей, чем на объективное освещение реалий, почему-то всегда забывают указать одно маленькое «но»: *режимы удвоенной и учетверённой скорости включаются только при пакетной передаче данных*. То есть, если мы запросили из памяти парочку мегабайтов с адреса X по адрес Y — то да, эти два мегабайта будут переданы с удвоенной/учетверённой скоростью. *А вот сам запрос на данные посылается по шине с «одинарной» скоростью — всегда!* Соответственно, если запросов у нас много, а размер пересылаемых данных не очень велик, то количество данных, которые «путешествуют» по шине с одинарной скоростью (а запрос — это тоже данные) будет почти равно количеству тех, которые передаются со скоростью удвоенной или учетверённой. Вроде бы нам никто открыто не врал, вроде бы DDR и QDR действительно работают, но... как говорится в одном старом анекдоте: «то ли он у кого-то украл шубу, то ли у него кто-то украл шубу, но что-то там с шубой не то...» ;)

Процессор «крупноблочно»

Кэш

Общее описание и принцип действия

Во всех современных процессорах есть кэш (по-английски — cache). Кэш — это некая особенная разновидность памяти (основная особенность, кардинально отличающая кэш от ОЗУ — скорость работы), которая является своего рода «буфером» между контроллером памяти и процессором. Служит этот буфер для увеличения скорости работы с ОЗУ. Каким образом? Сейчас попытаемся объяснить. При этом мы решили отказаться от папахивающих детским садом сравнений, которые частенько встречаются в популяризаторской литературе на процессорную тематику (бассейны, соединённые трубами разного диаметра, и т.д. и т.п.). Всё-таки человек, который дочитал статью до этого места, и не заснул — наверное, способен выдержать и «переварить» чисто техническое объяснение, без бассейнов, кошечек и одуванчиков.

Итак, представим, что у нас есть много сравнительно медленной памяти (пусть это будет ОЗУ размером 10'000'000 байт) и относительно мало очень быстрой (пусть это будет кэш размером всего 1024 байта). Как нам с помощью этого несчастного килобайта увеличить скорость работы со всей памятью вообще? А вот здесь следует вспомнить, что данные в процессе работы программы, как правило, не бездумно перекидываются с места на место — они *изменяются*. Считали из памяти значение какой-то переменной, прибавили к нему какое-то число — записали обратно на то же место. Считали массив, отсортировали по возрастанию — опять-таки записали в память. То есть в один какой-то момент программа работает не со всей памятью целиком, а, как правило, с относительно маленьким её фрагментом. Какое решение напрашивается? Правильно: загрузить этот фрагмент в «быструю» память, обработать его там, а потом уже записать обратно в «медленную» (или просто удалить из кэша, если данные не изменялись). В общем случае, именно так и работает процессорный кэш: любая считываемая из памяти информация попадает не только в процессор, но и в кэш. И если эта же информация (тот же адрес в памяти) нужна снова, сначала процессор проверяет: а нет ли её в кэше? Если есть — информация берётся оттуда, и обращения к памяти не происходит вовсе. Аналогично с записью: информация, если её объём влезает в кэш — пишется именно туда, и только потом, когда процессор закончил операцию записи, и занялся выполнением других команд, данные, записанные в кэш,

параллельно с работой процессорного ядра «потихоньку выгружаются» в ОЗУ.

Разумеется, объём данных, прочитанных и записанных за всё время работы программы — намного больше объёма кэша. Поэтому некоторые из них приходится время от времени удалять, чтобы в кэш могли поместиться новые, более актуальные. Самый простой из известных механизмов обеспечения данного процесса — отслеживание времени последнего обращения к данным, находящимся в кэше. Так, если нам необходимо поместить новые данные в кэш, а он уже «забит под завязку», контроллер, управляющий кэшем, смотрит: к какому фрагменту кэша не происходило обращения дольше всего? Именно этот фрагмент и является первым кандидатом на «вылет», а на его место записываются новые данные, с которыми нужно работать сейчас. Вот так, в общих чертах, работает механизм кэширования в процессорах. Разумеется, приведенное выше объяснение весьма примитивно, на самом деле всё ещё сложнее, но, надеемся, общее представление о том, зачем процессору нужен кэш и как он работает, вы получить смогли.

А для того чтобы было понятно, насколько важен кэш, приведем простой пример: скорость обмена данными процессора Pentium 4 со своим кэшем более чем в 10 раз (!) превосходит скорость его работы с памятью. Фактически, в полную силу современные процессоры способны работать только с кэшем: как только они сталкиваются с необходимостью прочитать данные из памяти — все их хваленые мегагерцы начинают просто «греть воздух». Опять-таки, простой пример: выполнение простейшей инструкции процессором происходит за один такт, то есть за секунду он может выполнить такое количество простых инструкций, какова его частота (на самом деле еще больше, но это оставим на потом...). А вот время ожидания данных из памяти может в худшем случае составить более 200 тактов! Что делает процессор, пока он ждет нужных данных? А ничего он не делает. Просто стоит и ждет...

Многоуровневое кэширование

Специфика конструирования современных процессорных ядер привела к тому, что систему кэширования в подавляющем большинстве CPU приходится делать многоуровневой. Кэш первого уровня (самый «близкий» к ядру) традиционно разделяется на две (как правило, равные) половины: кэш инструкций (L1I) и кэш данных (L1D). Это разделение предусматривается так называемой «гарвардской архитектурой» процессора, которая по состоянию на сегодня является самой популярной теоретической разработкой для построения современных CPU. В L1I, соответственно, аккумулируются только команды (с ним работает декодер, см. ниже), а в L1D — только данные (они впоследствии, как правило, попадают во внутренние регистры процессора). «Над L1» стоит кэш второго уровня — L2. Он, как правило, больше по объёму, и является уже «смешанным» — там располагаются и команды, и данные. L3 (кэш третьего уровня), как правило, полностью повторяет структуру L2, и в современных x86 CPU встречается редко. Чаще всего, L3 — это плод компромисса: за счёт использования более медленной и узкой шины, его можно сделать очень большим, но при этом скорость L3 всё равно остаётся более высокой, чем скорость памяти (хотя и не такой высокой, как у L2-кэша). Тем не менее, алгоритм работы с многоуровневым кэшем в общих чертах не отличается от алгоритма работы с одноуровневым, просто добавляются лишние итерации: сначала информация ищется в L1, если её там нет — в L2, потом — в L3, и уже потом, если ни на одном уровне кэша она не найдена — идёт обращение к основной памяти (ОЗУ).

Декодер

На самом деле, исполнительные блоки всех современных десктопных x86-процессоров... вовсе не работают с кодом в стандарте x86. У каждого процессора есть своя, «внутренняя» система команд, не имеющая ничего общего с теми командами (тем самым «кодом»), которые поступают извне. В общем случае, команды, исполняемые ядром — намного проще,

«примитивнее», чем команды стандарта x86. Именно для того, чтобы процессор «внешне выглядел» как x86 CPU, и существует такой блок как декодер: он отвечает за преобразование «внешнего» x86-кода во «внутренние» команды, исполняемые ядром (при этом достаточно часто одна команда x86-кода преобразуется в несколько более простых «внутренних»). Декодер является очень важной частью современного процессора: от его быстродействия зависит то, насколько постоянным будет поток команд, поступающих на исполняющие блоки. Ведь они неспособны работать с кодом x86, поэтому то, будут они что-то делать, или простаивать — во многом зависит от скорости работы декодера. Достаточно необычный способ ускорить процесс декодирования команд реализовала в процессорах архитектуры NetBurst компания Intel — см. ниже про Trace cache.

Исполняющие (функциональные) устройства

Пройдя через все уровни кэша и декодер, команды наконец-то попадают на те блоки, ради которых вся эта катавасия и устраивалась: *исполняющие* устройства. По сути, именно исполняющие устройства и являются *единственно необходимым элементом* процессора. Можно обойтись без кэша — скорость снизится, но программы работать будут. Можно обойтись без декодера — исполняющие устройства станут сложнее, но работать процессор будет. В конце концов, ранние процессоры архитектуры x86 (i8086, i80186, 286, 386, 486, Am5x86) — как-то без декодера обходились. Без исполняющих устройств обойтись невозможно, ибо именно они исполняют код программы. В самом первом приближении они традиционно делятся на две больших группы: арифметико-логические устройства (ALU) и блок вычислений с плавающей точкой (FPU).

Арифметико-логические устройства

ALU традиционно отвечают за два типа операций: арифметические действия (сложение, вычитание, умножение, деление) с целыми числами, логические операции с опять-таки целыми числами (логическое «и», логическое «или», «исключающее или», и тому подобные). Что, собственно, и следует из их названия. Блоков ALU в современных процессорах, как правило, несколько. Для чего — вы поймёте позже, прочитав раздел «Суперскалярность и внеочередное исполнение команд». Понятно, что ALU может исполнить только те команды, которые предназначены для него. Распределением команд, поступающих с декодера, по различным исполняющим устройствам, занимается специальный блок, но это уже, как говорится, «слишком сложные материи», и их вряд ли имеет смысл разьяснять в материале, который посвящен лишь поверхностному ознакомлению с основными принципами работы современных x86 CPU.

Блок вычислений с плавающей запятой*

FPU занимается выполнением команд, работающих с числами с плавающей запятой, кроме того, традиционно на него «вешают всех собак» в виде всяческих дополнительных наборов команд (MMX, 3DNow!, SSE, SSE2, SSE3...) — независимо от того, работают они с числами с плавающей запятой, или с целыми. Как и в случае с ALU, отдельных блоков в FPU может быть несколько, и они способны работать параллельно.

* — согласно традиций русской математической школы, мы называем FPU «блоком вычислений с плавающей **запятой**», хотя буквально его название (Floating **P**oint Unit) переводится как «...с плавающей точкой» — согласно американскому стандарту написания таких чисел.

Регистры процессора

Регистры — по сути, те же ячейки памяти, но «территориально» они расположены прямо в процессорном ядре. Разумеется, скорость работы с регистрами во много раз превосходит как скорость работы с ячейками памяти, расположенными в основном ОЗУ (тут вообще на порядки...), так и с кэшами любого уровня. Поэтому большинство команд архитектуры x86 предусматривают осуществление действий именно над содержимым регистров, а не над содержимым памяти. Однако общий объём регистров процессора, как правило, очень мал — он не сравним даже с объёмом кэшей первого уровня. Поэтому де-факто код программы (не на языке высокого уровня, а именно бинарный, «машинный») часто содержит следующую последовательность операций: загрузить в один из регистров процессора информацию из ОЗУ, загрузить в другой регистр другую информацию (тоже из ОЗУ), произвести некое действие над содержимым этих регистров, поместив результат в третий — а потом снова выгрузить результат из регистра в основную память.

Процессор в подробностях

Особенности кэшей

Частота работы кэша и его шина

Во всех современных x86 CPU все уровни кэша работают на той же частоте, что и процессорное ядро, но это вовсе не всегда было так (данный вопрос уже поднимался выше). Однако скорость работы с кэшем зависит не только от частоты, но и от ширины шины, с помощью которой он соединён с процессорным ядром. Как вы (надеюсь) помните из ранее прочитанного, скорость передачи данных является, по сути, произведением частоты работы шины (количества тактов в секунду) на количество байт, которые передаются по шине за один такт. Количество передаваемых за такт байтов можно увеличивать за счёт введения DDR и QDR (Double Data Rate и Quad Data Rate) протоколов — или просто за счёт увеличения ширины шины. В случае с кэшем более популярен второй вариант — не в последнюю очередь из-за «пикантных особенностей» DDR/QDR, описанных выше. Разумеется, минимально разумной шириной шины кэша является ширина внешней шины самого процессора, то есть, по состоянию на сегодняшний день — 64 бита. Именно так, в духе здорового минимализма, и поступает компания AMD: в её процессорах ширина шины L1 \longleftrightarrow L2 равна 64 битам, но при этом она двунаправленная, то есть, способна работать одновременно на передачу и приём информации. В духе «здорового гигантизма» в очередной раз поступила компания Intel: в её процессорах, начиная с Pentium III «Сорперmine», шина L1 \longleftrightarrow L2 имеет ширину... 256 бит! По принципу «кашу маслом не испортишь», как говорится. Правда, шина эта однонаправленная, то есть в один момент времени работает либо только на передачу, либо только на приём. Споры о том, какой из подходов лучше (двунаправленная шина, но более узкая, или однонаправленная широкая) — продолжаются до сих пор... впрочем, равно как и множество других споров относительно технических решений, применяемых двумя основными конкурентами на рынке x86 CPU.

Эксклюзивный и не эксклюзивный кэш

Концепции эксклюзивного и не эксклюзивного кэширования очень просты: в случае не эксклюзивного кэша, информация на всех уровнях кэширования может дублироваться. Таким образом, L2 может содержать в себе данные, которые уже находятся в L1I и L1D, а L3 (если он есть) может содержать в себе полную копию всего содержимого L2 (и, соответственно, L1I и L1D). Эксклюзивный кэш, в отличие от не эксклюзивного,

предусматривает чёткое разграничение: если информация содержится на каком-то уровне кэша — то на всех остальных она отсутствует. Плюс эксклюзивного кэша очевиден: общий размер кэшируемой информации в данном случае равен суммарному объёму кэшей всех уровней — в отличие от не эксклюзивного кэша, где размер кэшируемой информации (в худшем случае) равен объёму самого большого уровня кэша. Минус эксклюзивного кэша менее очевиден, но он есть: необходим специальный механизм, который следит за собственно «эксклюзивностью» (так, например, при удалении информации из L1-кэша, перед этим автоматически инициируется процесс её копирования в L2).

Не эксклюзивный кэш традиционно использует компания Intel, эксклюзивный (с момента появления процессоров Athlon на ядре Thunderbird) — компания AMD. В целом, мы наблюдаем здесь классическое противостояние между объёмом и скоростью: за счёт эксклюзивности, при одинаковых объёмах L1/L2 у AMD общий размер кэшируемой информации получается больше — но за счёт неё же он работает медленней (задержки, вызванные наличием механизма обеспечения эксклюзивности). Следует, наверное, заметить, что недостатки не эксклюзивного кэша компания Intel в последнее время компенсирует просто, тупо, но весомо: наращивая его объёмы. Для топовых процессоров данной компании стал уже почти что нормой L2-кэш объёмом 2 МБ — и AMD с её 128 КБ L1C+L1D и максимум 1 МБ L2 пока «не переплюнуть» эти 2 МБ даже за счёт эксклюзивности.

Кроме того, увеличивать общий объём кэшируемой информации за счёт введения эксклюзивной архитектуры кэша имеет смысл только в том случае, когда выигрыш в объёме получается достаточно большим. Для компании AMD это актуально т.к. у её сегодняшних CPU суммарный объём L1D+L1I равен 128 КБ. Процессорам Intel, у которых объём L1D равен максимум 32 КБ, а L1I иногда имеет совсем другую структуру (см. про Trace cache), введение эксклюзивной архитектуры дало бы намного меньше пользы.

*А ещё есть такое распространённое заблуждение, что архитектура кэша у CPU компании Intel «инклюзивная». На самом деле — нет. Именно НЕ эксклюзивная. Инклюзивная архитектура предусматривает, что на «нижнем» уровне кэша **не может** находиться ничего, чего нет на более «верхнем». Не эксклюзивная архитектура всего лишь **допускает** дублирование данных на разных уровнях.*

Trace cache

Концепция Trace cache, состоит в том, чтобы сохранять в кэше инструкций первого уровня (L1I) не те команды, которые считаны из памяти, а уже декодированные последовательности (см. декодер). Таким образом, если некая x86-команда исполняется повторно, и она всё ещё находится в L1I, декодеру процессора не нужно снова преобразовывать её в последовательность команд «внутреннего кода», так как L1I содержит данную последовательность в уже декодированном виде. Концепция Trace cache очень удачно вписывается в общую концепцию архитектуры Intel NetBurst, ориентированную на создание процессоров с очень высокой частотой работы ядра. Однако полезность Trace cache для [относительно] менее высокочастотных CPU до сих пор находится под вопросом, так как сложность организации Trace cache становится сопоставима с задачей конструирования обычного быстрого декодера. Поэтому, отдавая должное оригинальности идеи, мы всё же сказали бы, что универсальным решением «на все случаи жизни» Trace cache считать нельзя.

Суперскалярность и внеочередное исполнение команд

Основная черта всех современных процессоров состоит в том, что они способны запускать на исполнение не только ту команду, которую (согласно коду программы) следует исполнить в данный момент времени, но и другие, следующие после неё. Приведём простой (канонический) пример. Пусть нам следует исполнить следующую последовательность

команд:

- 1) $A = B + C$
- 2) $Z = X + Y$
- 3) $K = A + Z$

Легко заметить, что команды (1) и (2) совершенно независимы друг от друга — они не пересекаются ни по исходным данным (переменные B и C в первом случае, X и Y во втором), ни по месту размещения результата (переменная A в первом случае и Z во втором). Стало быть, если на данный момент у нас есть свободные исполняющие блоки в количестве более одного, данные команды можно распределить по ним, и выполнить одновременно, а не последовательно*. Таким образом, если принять время исполнения каждой команды равным N тактов процессора, то в классическом случае исполнение всей последовательности заняло бы $N*3$ тактов, а в случае с параллельным исполнением — всего $N*2$ тактов (так как команду (3) нельзя выполнить, не дождавись результата исполнения двух предыдущих).

* — разумеется, степень параллелизма не бесконечна: команды могут быть выполнены параллельно только в том случае, когда на данный момент времени есть в наличии соответствующее количество свободных от работы блоков (ФУ), причём именно таких, которые «понимают» рассматриваемые команды. Самый простой пример: блок, относящийся к ALU, физически неспособен исполнить инструкцию, предназначенную для FPU. Обратное также верно.

На самом деле всё ещё сложнее. Так, если у нас имеется следующая последовательность:

- 1) $A = B + C$
- 2) $K = A + M$
- 3) $Z = X + Y$

То очередь исполнения команд процессором будет изменена! Так как команды (1) и (3) независимы друг от друга (ни по исходным данным, ни по месту размещения результата), они могут быть выполнены параллельно — и будут выполнены параллельно. А вот команда (2) будет выполнена после них (третьей) — поскольку для того, чтобы результат вычислений был корректен, необходимо, чтобы перед этим была выполнена команда (1). Именно поэтому обсуждаемый в данном разделе механизм и называется «внеочередным исполнением команд» (Out-of-Order Execution, или сокращённо «OoO»): в тех случаях, когда очерёдность выполнения никак не может сказаться на результате, команды отправляются на исполнение не в той последовательности, в которой они располагаются в коде программы, а в той, которая позволяет достичь максимального быстродействия.

Теперь вам должно стать окончательно понятно, зачем современным CPU такое количество однотипных исполняющих блоков: они обеспечивают возможность параллельного выполнения нескольких команд, которые в случае с «классическим» подходом к проектированию процессора пришлось бы выполнять в той последовательности, в которой они содержатся в исходном коде, одну за другой.

Процессоры, оснащённые механизмом параллельного исполнения нескольких подряд идущих команд, принято называть «суперскалярными». Однако не все суперскалярные процессоры поддерживают внеочередное исполнение. Так, в первом примере нам достаточно «простой суперскалярности» (выполнения двух последовательных команд одновременно) — а вот во втором примере без перестановки команд местами уже не обойтись, если мы хотим получить максимальное быстродействие. Все современные x86 CPU обладают обоими качествами: являются суперскалярными, и поддерживают внеочередное исполнение команд. В то же время, были в истории x86 и «простые суперскаляры», OoO не поддерживающие. Например, классическим десктопным x86-суперскаляром без OoO был Intel Pentium [MMX].

Справедливости ради, стоит заметить, что никаких заслуг в разработке концепций

суперскалярности и OoO — нет ни у Intel, ни у AMD, ни у какого-либо иного (в том числе из ныне почивших) производителя x86 CPU. Первый суперскалярный компьютер, поддерживающий OoO, был разработан Сеймуром Креем (Seymour Cray) ещё в 60-х годах XX века. Для сравнения: Intel свой первый суперскалярный процессор (Pentium) выпустила в 1993 году, первый суперскаляр с OoO (Pentium Pro) — в 1995 году; первый суперскаляр с OoO от AMD (K5) увидел свет в 1996 году. Комментарии, как говорится, излишни...

Предварительное (опережающее) декодирование и кэширование

Предсказание ветвлений

В любой более-менее сложной программе присутствуют команды условного перехода: «Если некое условие истинно — перейти к исполнению одного участка кода, если нет — другого». С точки зрения скорости выполнения кода программы современным процессором, поддерживающим внеочередное исполнение, любая команда условного перехода — воистину бич божий. Ведь до тех пор, пока не станет известно, какой участок кода после условного перехода окажется «актуальным» — его невозможно начать декодировать и исполнять (см. внеочередное исполнение). Для того чтобы как-то примирить концепцию внеочередного исполнения с командами условного перехода, предназначается специальный блок: блок предсказания ветвлений. Как понятно из его названия, занимается он, по сути, «пророчествами»: пытается предсказать, на какой участок кода укажет команда условного перехода, ещё до того, как она будет исполнена. В соответствии с указаниями «штатного внутриядерного пророка», процессором производятся вполне реальные действия: «напророченный» участок кода загружается в кэш (если он там отсутствует), и даже начинается декодирование и выполнение его команд. Причём среди выполняемых команд также могут содержаться инструкции условного перехода, и их результаты тоже предсказываются, что порождает целую цепочку из *пока не проверенных* предсказаний! Разумеется, если блок предсказания ветвлений ошибся, вся проделанная в соответствии с его предсказаниями работа просто аннулируется.

На самом деле, алгоритмы, по которым работает блок предсказания ветвлений, вовсе не являются шедеврами искусственного интеллекта. Преимущественно они просты... и тупы. Ибо чаще всего команда условного перехода встречается в циклах: некий счётчик принимает значение X , и после каждого прохождения цикла значение счётчика уменьшается на единицу. Соответственно, до тех пор, пока значение счётчика больше нуля — осуществляется переход на начало цикла, а после того, как он становится равным нулю — исполнение продолжается дальше. Блок предсказания ветвлений просто анализирует результат выполнения команды условного перехода, и считает, что если N раз подряд результатом стал переход на определённый адрес — то и в $N+1$ случае будет осуществлён переход туда же. Однако, несмотря на весь примитивизм, данная схема работает просто замечательно: например, в случае, если счётчик принимает значение 100, а «порог срабатывания» предсказателя ветвлений (N) равен двум переходам подряд на один и тот же адрес — легко заметить, что 97 переходов из 98 будут предсказаны правильно!

Разумеется, несмотря на достаточно высокую **эффективность** простых алгоритмов, механизмы предсказания ветвлений в современных CPU всё равно постоянно совершенствуются и усложняются — но тут уже речь идёт о борьбе за единицы процентов: например, за то, чтобы повысить эффективность работы блока предсказания ветвлений с 95 процентов до 97, или даже с 97% до 99...

Предвыборка данных

Блок предвыборки данных (Prefetch) очень похож по принципу своего действия на блок предсказания ветвлений — с той только разницей, что в данном случае речь идёт не о коде, а о данных. Общий принцип действия такой же: если встроенная схема анализа доступа к данным в ОЗУ решает, что к некоему участку памяти, ещё не загруженному в кэш, скоро будет осуществлён доступ — она даёт команду на загрузку данного участка памяти в кэш ещё до того, как он понадобится исполняемой программе. «Умно» (результативно) работающий блок предвыборки позволяет существенно сократить время доступа к нужным данным, и, соответственно, повысить скорость исполнения программы. К слову: грамотный Prefetch очень хорошо компенсирует высокую латентность подсистемы памяти, подгружая нужные данные в кэш, и тем самым, нивелируя задержки при доступе к ним, если бы они находились не в кэше, а в основном ОЗУ.

Однако, разумеется, в случае ошибки блока предвыборки данных, неизбежны негативные последствия: загружая де-факто «ненужные» данные в кэш, Prefetch вытесняет из него другие (быть может, как раз нужные). Кроме того, за счёт «предвосхищения» операции считывания, создаётся дополнительная нагрузка на контроллер памяти (де-факто, в случае ошибки — совершенно бесполезная).

Алгоритмы Prefetch, как и алгоритмы блока предсказания ветвлений, тоже не блещут интеллектуальностью: как правило, данный блок стремится отследить, не считывается ли информация из памяти с определённым «шагом» (по адресам), и на основании этого анализа пытается предсказать, с какого адреса будут считываться данные в процессе дальнейшей работы программы. Впрочем, как и в случае с блоком предсказания ветвлений, простота алгоритма вовсе не означает низкую **эффективность**: в среднем, блок предвыборки данных чаще «попадает», чем ошибается (и это, как и в предыдущем случае, прежде всего связано с тем, что «массированное» чтение данных из памяти, как правило происходит в процессе исполнения различных циклов).

Заключение

*Я — тот кролик, который не может начать жевать траву до тех пор, пока не поймёт во всех деталях, как происходит процесс фотосинтеза!
(изложение личной позиции одним из близких знакомых автора)*

Вполне возможно, те чувства, которые у вас возникли после прочтения данной статьи, можно описать примерно следующим образом: «Вместо того чтобы на пальцах объяснить, какой процессор лучше — взяли и загрузили мне мозги кучей специфической информации, в которой ещё разбираться и разбираться, и конца-края не видно!» Вполне нормальная реакция: поверьте, мы вас хорошо понимаем. Скажем даже больше (и пусть с головы упадёт корона!): если вы думаете, что мы сами можем ответить на этот простецкий вопрос («какой процессор лучше?») — то вы очень сильно заблуждаетесь. Не можем. Для одних задач лучше один, для других — другой, а тут ещё цена разная, доступность, симпатии конкретного пользователя к определённым маркам... Не имеет задача однозначного решения. Если бы имела — наверняка кто-то бы его нашёл, и стал бы самым знаменитым обозревателем за всю историю независимых тестовых лабораторий.

Хотелось бы подчеркнуть ещё раз: *даже полностью усвоив и осмыслив всю информацию, изложенную в данном материале — вы по-прежнему не сможете предсказать, какой из двух процессоров будет быстрее в ваших задачах, глядя только на их характеристики.* Во-первых — потому, что далеко не все характеристики процессоров здесь рассмотрены. Во-вторых — потому, что есть и такие параметры CPU, которые в числовом виде могут быть представлены только с очень большой «натяжкой». Так для кого же (и для чего) всё это

написано? В основном — для тех самых «кроликов», которые непременно желают знать, что происходит внутри тех устройств, которыми они пользуются ежедневно. Зачем? Может, они просто лучше себя чувствуют, когда знают, что вокруг них происходит? :)

В ближайших планах на расширение FAQ:

1. Раздел, посвящённый многопроцессорным системам: объяснение понятия SMP, технология Hyper-Threading, N-процессорность, N-ядерность.
2. Раздел, посвящённый физическим характеристикам CPU: типы корпусов, сокет, энергопотребление, и т.п.

Станислав Гарматюк (nawhi@ixbt.com)

Опубликовано — 9 февраля 2006 г.