Стремление сделать компьютер “предельных характеристик” существует столько же, сколько существуют и сами компьютеры. Но если поначалу предельность определялась общей надёжностью системы (грубо – числом дискретных элементов, паяных и разъёмных соединений), то на данный момент развитие упёрлось в такие фундаментальные ограничения, как скорость света и размер атомов. В самом деле, расстояние Si — Si равно 2,34 Å, так что в 7 нанометров можно уложить лишь 30 атомов кремния. И этот кремний еще надо более-менее равномерно легировать.

Технология изготовления микросхем в её нынешнем (планарном) виде доведена до совершенства и существенного прогресса здесь ждать не стоит. Но можно побороться за предельные характеристики архитектуры, за то, что Б.А.Бабаян называет “не-улучшаемой архитектурой”. Т.е. той, для которой пара компилятор/процессор способна выдавать на большинстве практических задач близкую к пиковой производительность.

Исторический экскурс

Начнём издалека, стоит осветить технологический ландшафт эпохи, без этого сложно понять логику событий, да и просто интересно.

Первые доступные микросхемы появились в начале-середине 60-х и вызвали революцию в разработке компьютеров. Если советские Эльбрусы, “антисоветские” Cray-1, Cyber-205, …. Были порождены наличием интегральных схем, то к моменту их постановки в серию уже возникли СБИС (VLSI) которые позволили разместить процессор на небольшом к-ве микросхем. Что дешевле, меньше расстояния, можно поднять частоту, меньше ест, надёжнее …

Впрочем, эволюция суперкомпьютеров в тот момент упёрлась в технологический предел – процессор, собранный на микросхемах был физически достаточно велик для того, чтобы безнаказанно поднимать тактовую частоту. А технология изготовления микросхем не была достаточно развита, чтобы разместить процессор на одном кристалле. К примеру, процессор Cray-3 состоял из сотен микросхем на арсениде галлия, которые удалось упаковать в объем, равный кубическому футу.

Интересно, что выход из технологического тупика разработчики архитектур искали в векторных конвейерных вычислителях. Это позволяло пусть и экстенсивным путём на специализированных задачах, но поднять пиковую производительность.

На другом конце спектра производительности тоже происходили интересные события. В 60-е годы появилась целая плеяда мини-компьютеров. Навскидку:

HP-2100 появился в продаже в 1966, выпущены десятки тысяч экземпляров, продукты этой серии сделали HP крупнейшим производителем миникомпьютеров и выпускались вплоть до 1990 г. Доступные языки программирования – диалекты FORTRAN-a, ALGOL, BASIC. И ассемблер, конечно, куда же без него.

DDP-516 – также 1966. Его вариант Honeywell H316 известен как “кухонный компьютер”, в его память был зашит список кулинарных рецептов, а в корпус встроена разделочная доска. Между прочим, именно H316 использовался в качестве основного компьютера для мониторинга температуры реактора АЭС Bradwell вплоть до 2000 года.Доступные языки – BASIC, FORTRAN, FORTH.

МИР – 1965, для инженерных расчетов, основной язык – АЛМИР-65

IBM-7094/7074 – 1962/1963 развитие IBM-7090 (первый транзисторный мэйнфрейм). Языки программирования – FORTRAN, COBOL

Наири 1964

Сетунь 1962

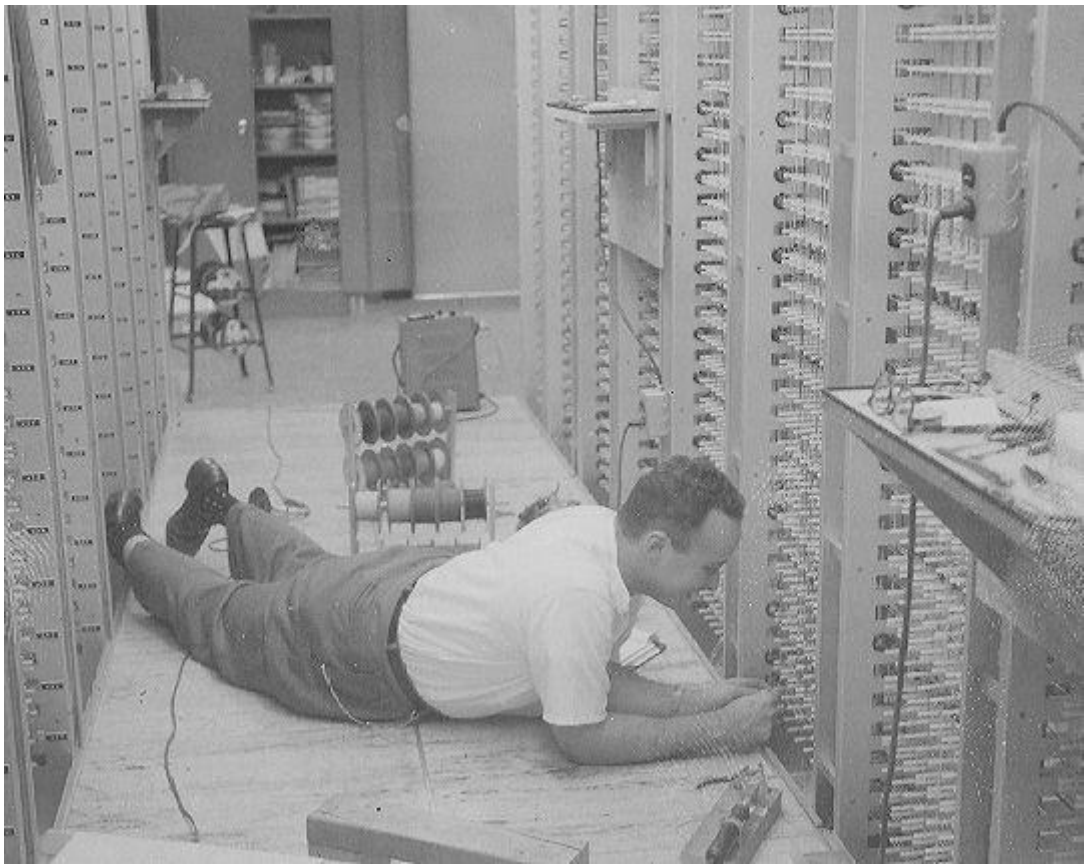
Минск 1963

PDP-1/4/…/10 1960…1969 Языки программирования FORTRAN, FOCAL, DIBOL, BASIC

PDP-11 (1970) стоит особо, он 16-разрядный и в нём появился аппаратный стек

Кроме миниатюризации шло и удешевление техники, если IBM-7090 стоил миллионы (тех ещё) долларов, то “кухонный компьютер” H316 уже около 10 000. В результате взрывным образом выросла доступность компьютеров, если число мэйнфреймов исчислялось сотнями, то мини-компьютеры продавались десятками тысяч.

Появилась физическая возможность небольшим коллективам разрабатывать и выпускать мини-компьютеры, что в свою очередь вызвало острую потребность в переносимом программном обеспечении. Ведь операционные системы в то время писались на ассемблере, языки высокого уровня не были приспособлены для системных задач. Впрочем, это уже был прогресс по сравнению с недавним прошлым, вот так, например, выглядела отладка на R1 с памятью на CRT(1960)



В то же время на более мощных компьютерах операционные системы достигли определённой зрелости. Стоит упомянуть совместный проект Bell Labs, GE и MIT под название Multics. Multics была задумана как операционная система, реализующая целую кучу революционных идей

Вытесняющая многозадачность

Официальное разделение памяти процессов на внутреннюю (оперативную) и внешнюю (файлы)

Централизованная файловая система

Сегментно – страничная виртуальная память

Динамическое связывание – разделяемые библиотеки

Многопроцессорность

Встроенная система безопасности

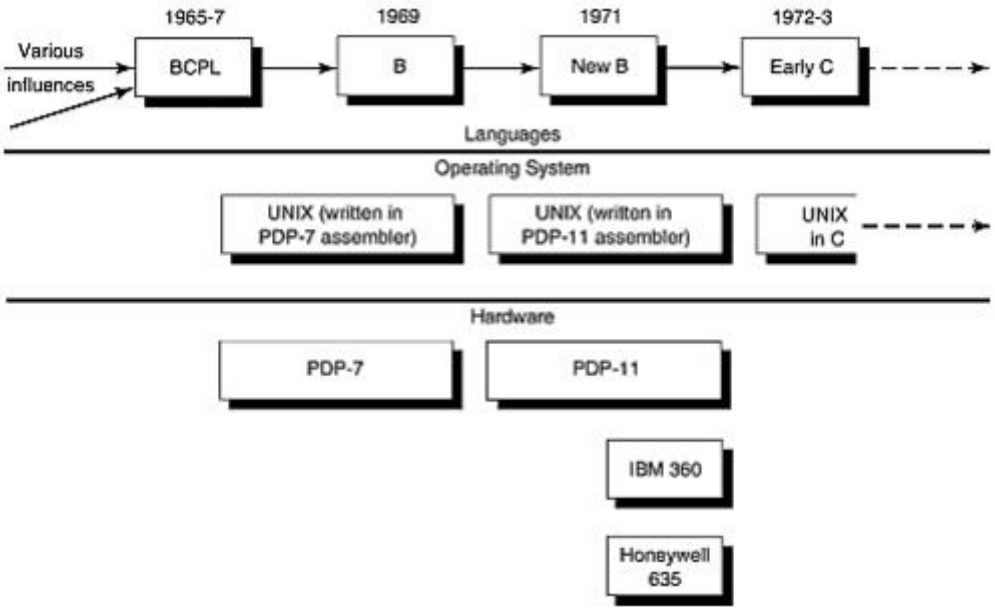
Конфигурация ОС на-лету

В основном написана на языке высокого уровня (PL/1)

…

В общем, всё то, чего мы сейчас интуитивно ожидаем от операционной системы, впервые в совокупности появилось именно в Multics.

К сожалению, данный проект, как и многие до и после него, был погребён под собственной сложностью. Произошло это в 1969 году и на остатках Multics, благодаря поначалу в основном Кену Томпсон и Деннису Ритчи из Bell Labs, появились ОС UNIX и язык программирования C.



Невозможно удержаться еще от того, чтобы привести еще одну фотографию.

Тем временем шел и невероятно быстрый прогресс в элементной базе. Первыми доступными однокристальными микропроцессорами стали 4-разрядные Intel 4004 (1971) и TMS1000 (1972) от TI. Первым 8-разрядным микропроцессором стал Intel 8008 (1972), а 16-разрядными, по видимому, HP BPC (1975) и TMS9900 от TI (1976). К первым 32-разрядным микропроцессорам можно отнести MC68000 от Motorola (1979), который, впрочем, имел 16-разрядную шину данных и АЛУ. И, наконец, первым 64-разрядным микропроцессором стал MIPS R4000 (1991). Intel i860 появился в 1989 г, но имел 32-разрядное АЛУ и 64-разрядный вычислитель с плавающей точкой (который, впрочем, умел работать с 64-разрядными целыми числами). I860 не был всё же процессором общего назначения, поэтому первенство по праву забирает именно R4000.

Итого, к середине-концу 80-х была доступна целая линейка честных 32-разрядных однокристальных микропроцессоров – Motorola MC68020 (1982), DEC-J11 (1983), Intel i386 (1985), NS32032 (1985), MIPS R2000 (1986), Fairchild Clipper (1986), SPARC V7 (1987), MIPS R3000 (1988), Motorola 88K (1988), AMD29K (1988), Intel i960 (1989).

Далее подтянулись MIPS R4000 64 (1991), HP PA-RISC 64 (1992), SPARC-V8 32(1992), PowerPC 32/64 (1993), DEC Alpha AXP 64 (1993), SPARC-V9 64 (1995)…

Войны CISC vs. RISC

К началу восьмидесятых однокристальные микропроцессоры освоились в 16-разрядной нише и наметился переход к 32 разрядам. Это уже была область компьютеров среднего класса мощности со своими лидерами, традициями, инструментами, опытом, … но также тупиками и накопленными ошибками.

В то же самое время компьютеры среднего класса стремительно дешевели, так, Interdata представила в 1973 г. Первый 32-разрядный мини-компьютер (IBM 360 совместимый) дешевле 10 000 долларов. В какой-то момент эти два процесса должны были столкнуться.

Академик И.П.Павлов наблюдал столкновение волн возбуждения и торможения в коре головного мозга, приводящее к лабораторным неврозам и назвал это “сшибкой”. Здесь эффект был отчасти похожим.

Часть разработчиков видела возможность сделать всё тоже самое, но компактнее, дешевле и мощнее. Например, DEC выпустила однокристальные процессоры T11(PDP-11) / J11(VAX). Или Texas Instruments с TMS9900 – однокристальным процессором, совместимым с популярной серией миникомпьютеров TI-990. Упомянем еще NS32K как развитие VAX. Глядя из будущего, легко понять, почему этот путь оказался тупиковым.

Другая часть проектов росла снизу вверх и имела успешные эволюционирующие продукты. К таким отнесём x86 и MC68K.

У Intel была простая и успешная серия 8086 наряду со сложной и провальной 432 серией. Может поэтому, архитектура развивалась достаточно эволюционно. 8086 превратился в дешевый 8088. Следующий шаг – 16-разрядные 80186 и более простой и дешевый 80188. 80286 16 разрядный процессор с 24-разрядной адресной шиной. 32-разрядный 80386 процессор с MMU тоже был совместим с изначальным 8086. При этом нельзя сказать, что Intel боялась экспериментировать – у них был и многопроцессорный iPCS с топологией гиперкуб и i860 с элементами VLIW и i960 с регистровыми окнами. Но об этом позже.

Аналогично поступила Motorola с удачной серией 68K, которая развивалась эволюционно – наращивали частоту, сделали полноценную 32-разрядную шину данных, АЛУ, … Причем, до этого существовала относительно удачная 8-разрядная серия 6800, которую они бросили и не побоялись сделать всё с нуля.

С третьей группой интереснее всего. Здесь отметились люди из академической среды, равно как и разработчики/исследователи из индустрии, которые увидели редкий шанс, имея за плечами и знания и практический опыт разработки процессоров, сделать новую[ые] архитектуру[ы] с нуля, на новом техническом уровне, по возможности избавившись от груза накопленных ошибок.

Необходимо учитывать также следующие обстоятельства.

Во-первых. На тот момент уже существовала свободно распространяемая (~ по цене магнитной ленты) переносимая операционная система (UNIX) с переносимым системным языком C. Причем, они еще не успели обрасти мышцами и накопить жирка, так что добавление новой архитектуры или учет её изменений не были слишком дорогостоящими. В начале UNIX развивался преимущественно энтузиастами и в 1985 году уже существовал в версии V8/BSD4.2 и был портирован с PDP-11 на VAX и Interdata 8/32 (IBM 360),

А также имелись коммерческие

386/ix от Interactive Systems Corp,

Xenix от Microsoft (x86, PDP 11, Z80, MC68K),

Idris от Whitesmiths (PDP 11, VAX, MC68K, IBM 370, Atari).

Во-вторых, произошла тихая революция в компиляторах. Условной точкой отсчета можно считать 1981 год, когда математиком Грегори Хайтином (Gregory Chaitin) был предложен [4,5] способ распределения регистров.

Речь идет о той стадии компиляции [6], когда программа преобразована во внутреннее представление в виде трёхадресного кода и до стадии кодогенерации. Трёхадресный код – фактически код для процессора с бесконечным количеством регистров. Но в конечной архитектуре количество регистров ограничено и требуется решить, какие значениях можно оставить в регистрах, а какие следует переместить в память (грубо).

До того момента эта стадия компиляции выполнялась эмпирически, с помощью набора разнообразных рецептов/приёмов, которые следовало применять в тех или иных ситуациях, качество такого кода зачастую оказывалось сомнительным. Грегори Хайтин пришел к выводу, что задача распределения регистров сводится к задаче раскраски графа. Про эту задачу известно, что она NP-полная, т.е. стоимость ее решения экспоненциально зависит от числа вершин (исходных регистров, которые надо распределить). К счастью, в данном случае это не приговор, поскольку была также предложена приемлемая эвристика. В результате удалось создать компилятор (для языка PL/1), причем качество генерируемого им кода было сопоставимо с написанным вручную на ассемблере.

Грегори Хайтин участвовал в проекте IBM 801 который не получил широкого распространения, но опыт и знания, полученные в нём были использованы в дальнейшем в архитектуре POWER. Проект начат в 1975 г. Как инициативная разработка. Была собрана и проанализирована статистика по огромному количеству реальных программ (частоты использования инструкций, тайминги, работа с памятью …) и сформулированы требования к перспективной архитектуре.

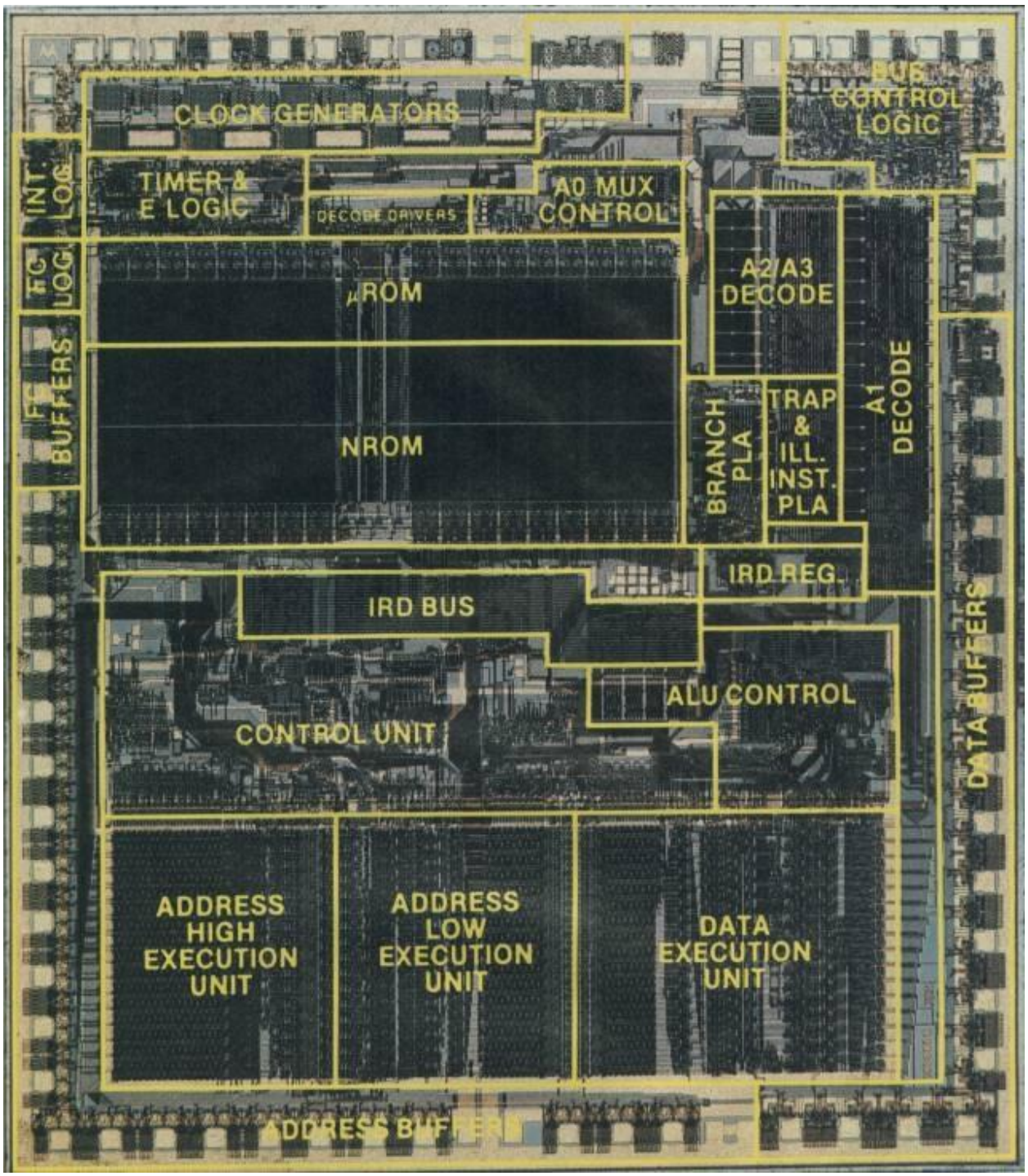
Надо сказать, что в то время идея перспективной архитектуры просто висела в воздухе и занимались ею не только в IBM, стоит упомянуть Berkeley RISC и Stanford MIPS проекты. Консенсус, сложившийся на тот момент был таков:

Микрокод – зло. Компьютеры с традиционной, т.н. CISC архитектурой традиционно имели набор инструкций, отличный от внутреннего представления этого кода. Причин несколько. Разработчики стремились к как можно более компактному коду, ведь память была не просто дорогой, её было совсем немного и хотелось поместить в неё как можно больше кода. Кроме того, если разработчики при анализе реально работающего кода (написанного преимущественно вручную на ассемблере) видели постоянно повторяющиеся паттерны инструкций, они стремились в следующей итерации архитектуры внести такие паттерны в архитектуру в виде новых инструкций [7].

В результате возник такой механизм как микрокод, который преобразовывал внешнее представление во внутреннее, пригодное для исполнения.

Эндрю Танненбаум пишет [3], что опкоды CISC инструкций напоминают ему коды Хаффмана, применяющиеся при сжатии данных. Пожалуй, это не удивительно. Микрокод можно/стоит воспринимать как аппаратный распаковщик для минимизации входного потока кода.

Объем логики, требовавшейся для работы микрокода, был не так уж и мал. Например, в MC68000 он занимал до 20% от площади микросхемы, т.е. его обслуживало ~ 15 000 из 68 000 транзисторов (да, процессоры серии 68000 состояли из чуть более чем 68000 транзисторов).



Идея, которую заложили в новый класс архитектур (RISC) заключалась в том, чтобы выровнять внутреннее и внешнее представление кода, избавиться от микрокода, а высвободившиеся транзисторы отдать под что-нибудь полезное, например, сделать побольше регистров.

Да, исполняемый код от этого распухнет, но к тому времени стоимость памяти уже не была столь критична, а объем адресуемого 32-разрядными процессорами пространства был достаточен для размещения кода любого объёма.

Глядя на это из современности, отказ от микрокода видится затеей сомнительной и конъюнктурной. Да, микрокод 68000 занимал 20%, но в транзисторах это величина примерно постоянная, а общий размер процессора нет. Для 68020 с его 190 000 транзисторов это уже 8%, а для 68040 - ~1%.

Чтобы донести распухший (на десятки процентов) RISC код до процессора, его нужно пропустить через интерфейс памяти, а это достаточно дорогой ресурс, именно пропускная способность памяти зачастую является узким местом. И не забудем про разработчиков процессоров, которые при вынесенной наружу внутренней архитектуре потеряли определённую гибкость.

Нужно больше регистров. С этим не поспоришь. Регистры являются сверхоперативной памятью – значения в них уже готовы для загрузки в исполнительные устройства процессора. Иногда их называют кэшем уровня L0.

В CISC архитектурах число регистров общего назначения невелико

68K – 8

X86 – 8

PDP-11 – 8

IBM 360 – 16

И увеличить их число невозможно, это будет уже другая архитектура. Одно уже только использование большего числа регистров в новой, написанной с нуля архитектуре способно увеличить её производительность в разы (по сравнению с CISC) при прочих равных.

Этим и вызваны работы Грегори Хайтина по автоматическому распределению регистров. В IBM-801 поначалу было 16 штук 24-разрядных регистров, но примерно в 1980 г. Архитектуру перепроектировали под 32 32-разрядных регистра. Распределять руками 32 регистра не слишком эффективно, обычный компилятор PL\1 так же не мог эффективно справляться с ними в силу того, что был оптимизирован под 16 регистров IBM-360/370.

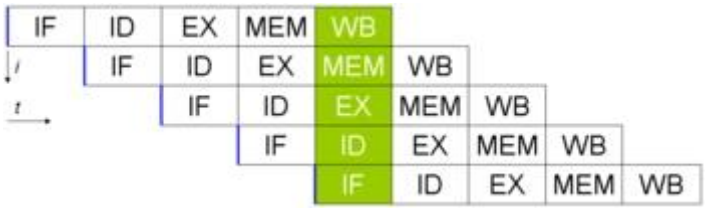
Концепция кэш-памяти в те годы была не в новинку и впервые появилась в IBM 360/86 (1968). В производительных моделях мини компьютеров она также присутствовала, но в первых однокристальных микропроцессорах её не было. Когда кэш память появилась в CISC процессорах, она в какой-то мере компенсировала малое количество регистров. Попробуем оценить эту компенсацию.

Сравнительные тесты экспериментального RISC II и 68000 показали, что первый в 1.4…4.2 раза быстрее. При этом 68000 имеет 16 регистров общего назначения, а RISC II – 22 регистра в окне вызова процедуры плюс 10 глобальных (всего 138 регистров). Известно, что 68000 не имеет кэш памяти, она в линейке появилась в процессоре 68020, это были 256 байт кэша инструкций. Кэш данных добавили в 68030, тоже 256 байт. Сравним их относительную производительность.

Известно, что 68000 выполнял в среднем 0.175 инструкций за такт. Для 68030 эта величина – 0.36 инструкций за такт. Следовательно, если привести RISC II к 68030, ускорение первого составит 0.68…2.04 раза. Что ж, результаты становятся не столь однозначными.

Интересно, что VAX 11/780, который выполняет 0.2 инструкции за такт, но содержит 8K кэш, в 0.85…2.65 раза медленнее RISC II.

Конвейер – благо. Каждый знает что такое конвейер. Идея известна с 60-х годов и к началу 80-х использование по крайней мере двухстадийного (пока одна инструкция исполняется, другая декодируется) конвейера было общепринятой практикой. Во всяком случае в 68000 был именно такой двухстадийный конвейер.



Обозначения на Фиг. 6

IF – instruction fetch

ID – instruction decode

EX – execute

MEM – memory access

WB – write back, сохранение результата

Упрощенные инструкции RISC на первый взгляд более подходят для конвейерной работы. Исполнительные устройства в RISC работают только с регистрами, если нужны данные из памяти, их предварительно требуется загрузить. Т.е. более сложные CISC инструкции после микро-кодирования распадаются на несколько внутренних (микро)инструкций, тогда как стадии конвейера касаются их всех, что менее эффективно.

В действительности, особой разницы нет, грубо, в CISC к конвейеру добавляется еще одна стадия – микро-декодирования. На производительности это явным образом не сказывается.

Для примера, в 68020 (1982 г.) появился трёхстадийный конвейер, в 68040 (1990 г) стадий стало 6 (и, кстати, кэш по 4 килобайта для данных и кода) с соответствующим приростом производительности – 1.1 (суперскаляр или ошибка ?) инструкции на такт.

Для сравнения, MIPS R3000 (1988 г.) имел 5-стадийный конвейер, не содержал внутри-процессорного кэша и имел производительность близкую к 1 инструкции за такт.

Проще инструкции – проще процессор – больше частота.

Сравним одноклассников

CISC

68030 – появился в 1987 г., 273 000 транзисторов, 3-стадийный конвейер, производительность 0.36 инструкций на такт, частоты 16…50 МГц, кэш внутренний 256 байт + 256 байт, техпроцесс .8 мкм

386DX - появился в 1985 г., 275 000 транзисторов, производительность 0.13 инструкций на такт, 16…40 Мгц, кэш внешний, техпроцесс 1.5 мкм

RISC

SPARC MB86900 (V7) – появился в 1986 г., 110 000 транзисторов, 5-стадийный конвейер, производительность 0.51 инструкций за такт, частота 14.3…33 МГц, кэш внешний, техпроцесс 1.2 мкм

R3000 - появился в 1988 г., 110 000 транзисторов, 5-стадийный конвейер, производительность 0.7…0.86 инструкций за такт, частоты 20…33 МГц, кэш внешний, техпроцесс 1.3 мкм

Производительность этих микропроцессоров сопоставима, особенно учитывая, что CISC инструкции “тяжелее” – выполняют больше работы. Насчет простоты – да, это правда, при сравнимой производительности RISC содержат на кристалле в 2.5 раза меньше логических элементов.

Итого. С технологической точки зрения, сколь-нибудь существенным элементом новизны в RISC процессорах было лишь увеличение числа регистров общего назначения. Это увеличение было возможно благодаря прогрессу в производстве СБИС а также тому, что архитектуры писались с чистого листа, без оглядки на обратную совместимость и т.п..

При разработке RISC архитектур ориентировались в первую очередь на существующие CISC процессоры и по сравнению с ними RISC выглядел весьма перспективно. Но к моменту выпуска в производство оказалось, что некоторые производители CISC смогли мобилизоваться и подтянуть производительность своих процессоров. В первую очередь это касалось молодых производителей микропроцессоров, которым не нужно было поддерживать пользователей их стремительно устаревающей техники.

Так или иначе, началась технологическая гонка, которую мы знаем как “CISC/RISC войны”. Основное технологическое преимущество RISC – регистры было нивелировано внутри-кристальным кэшем, обе стороны наращивали его объем, тактовую частоту и сложность конвейеров, интегрировали сопроцессоры с плавающей точкой … В какой-то момент микропроцессоры научились переименовывать регистры, переставлять микро-инструкции… и тут на сцену вышли суперскалярные микропроцессоры, которым стало (по большому счету) всё равно, какого типа – RISC или CISC у них набор внешних инструкций. С этого момента понятие RISC – не более, чем название раскрученного брэнда, которое используется преимущественно по инерции (либо маркетологами).

Главными проигравшими в этой борьбе стали те технологические компании, что не смогли или не успели включиться в соревнование и постепенно потеряли свою аудиторию. Главными выигравшими – те, кто смог выдержать 10-летнюю гонку. И, конечно, пользователи, которые с осторожным оптимизмом оплатили всё это великолепие.

Superscalar vs. VLIW

К началу 90-х годов сложилась довольно интересная ситуация. Коммерчески доступными на тот момент были следующие 32-разрядные микропроцессоры:

CISC

Motorola 68040 – 25 и 33 Мгц, производительность 0.7….1.1 (целочисленных) инструкций на такт, 1.17 млн транзисторов

Intel 486DX – 25 и 33 Мгц, производительность 0.338 инструкций (0.388 DX2, 0.7 DX4) на такт, 1.18 млн транзисторов

RISC

SPARC V8 – 33…40 МГц, , производительность 0.6 инструкций на такт, 1 млн транзисторов

MIPS R3400, R3500 – 25…40 МГц, 0.7 инструкций на такт

IBM POWER, 25 МГц, 0.75 инструкций на такт

HP PA-7000, 66 МГц, 0.58 млн транзисторов, 0.66 инструкций на такт

Intel i960 – 16…33 МГц, по-видимому первый микропроцессор с суперскалярным ядром, но без сопроцессора и MMU

AMD 29K – 16…40 МГц

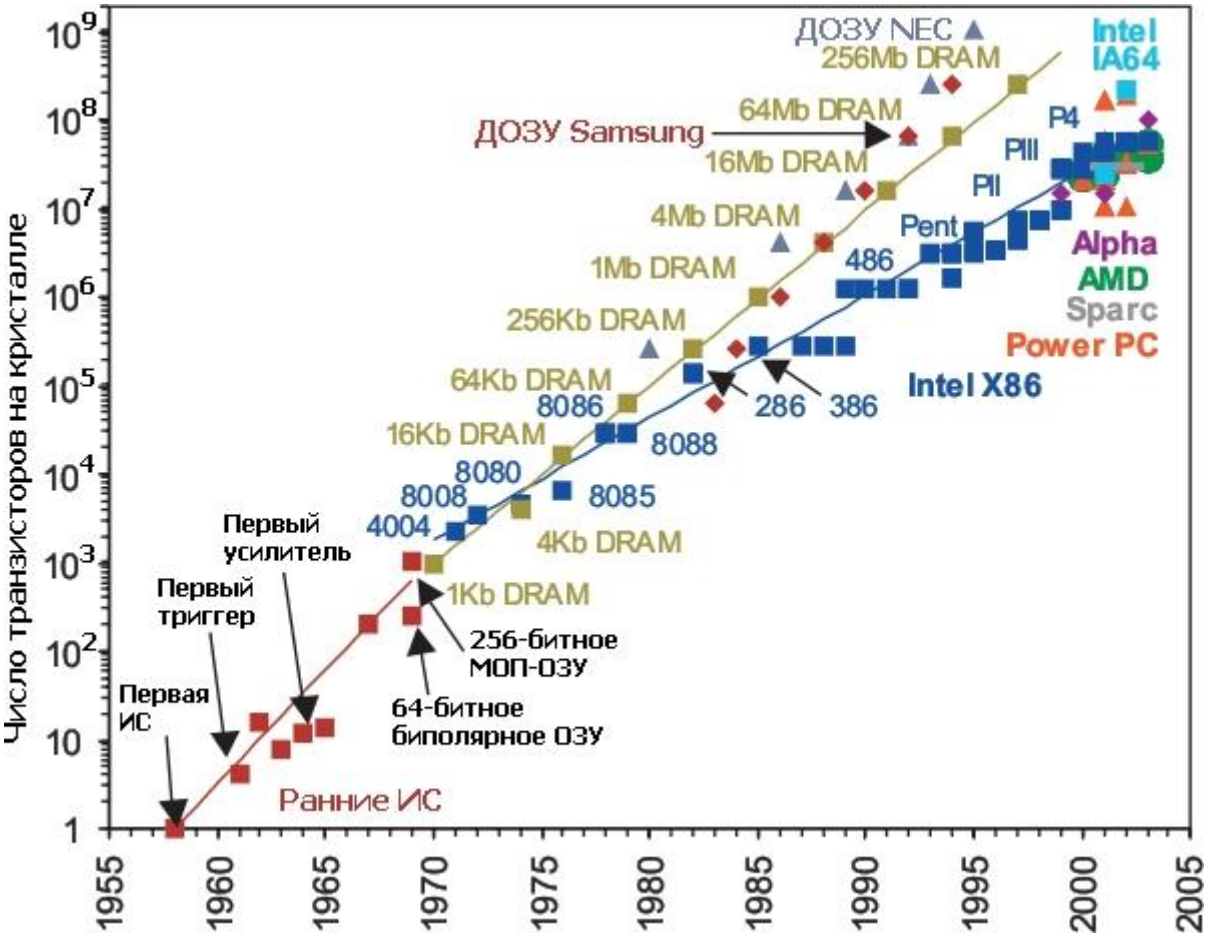
Motorola 88100 – 25…33 МГц

К 1990-1991 гг среди 32-разрядных микропроцессоров существовал довольно широкий выбор, причем производительность отличалась не на порядки. Немного особняком стоит i960, который предназначался для встраиваемых систем, впрочем, был вариант и с MMU и с сопроцессором. ARM не упоминается т.к. на тот момент находился в зачаточном состоянии.

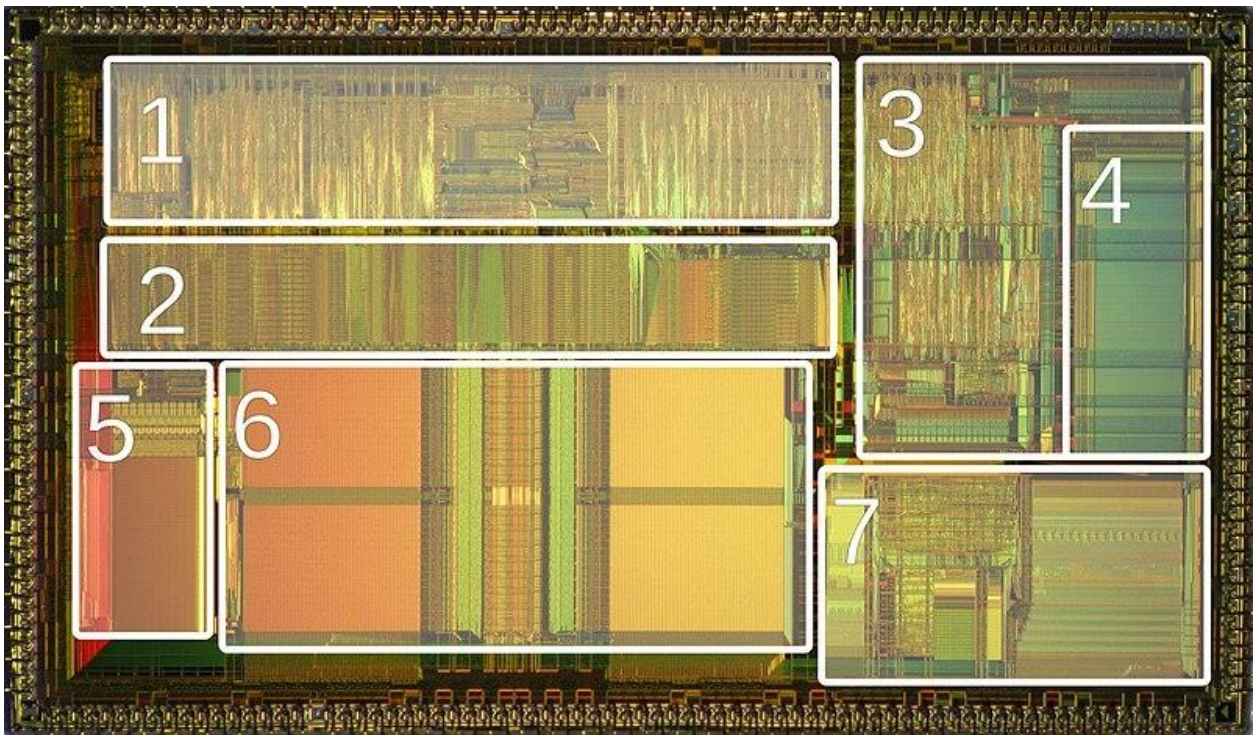
Вариантов в каком направлении развиваться было несколько

Ждать у моря погоды. Поскольку в тот момент активно действовало эмпирическое правило, известное как “закон Мура”, достаточно было лишь адаптировать свою архитектуру к новым тех.процессам, чтобы её производительность росла.

Эволюционный путь. Поскольку росло не только быстродействие транзисторов, но, в первую очередь, уменьшался их размер, появлялась возможность на той же площади кристалла разместить побольше этих самых транзисторов. См Фиг. 7



На что можно было потратить эти транзисторы? В первую очередь на кэш память. А также на расширение внутренних интерфейсов, общую оптимизацию …



На Фиг. 8 показана микрофотография 80486DX2, из общих 1.2 млн транзисторов почти половина приходится на кэш (под номером 6).

Отдельно стоит остановиться на вопросе, почему нельзя бесконтрольно увеличивать число регистров в архитектуре.

Поскольку одна из основных идей RISC процессоров это большое по сравнению с CISC число регистров, почему бы еще не увеличить их количество при выпуске новой версии архитектуры (как это в дальнейшем произошло, например, при переходе x86 => x86-64)?

Большое количество регистров полезно при выполнении линейного участка кода, когда регистры работают как хранилище промежуточных значений и как кэш нулевого уровня. Однако, всё усложняется, если требуется вызвать функцию. Компилятор ведь может и не знать, что делает эта функция и как она распоряжается регистрами (1, 2).

Раз так, перед вызовом содержимое регистров требуется сохранить.

Когда регистров мало, как в PDP-11, то и проблемы никакой нет, значения параметров вызова передаются через стек (cdecl). Когда их становится больше, появляется соблазн передать параметры вызова через регистры (что регламентируется ABI архитектуры).

Регистровые машины (CISC + наследники Stanford MIPS: MIPS, PA-RISC, PowerPC, 88K). Регистры делятся на две категории – volatile: ответственность за сохранение их содержимого лежит на вызывающей стороне и non-volatile – те, о ком должна заботиться вызываемая функция.Вот, например, как обстоят дела с с 32-разрядным компилятором OS X для PowerPC, где регистров по 32 – целочисленных и с плавающей точкой:

Volatile: GPR0, GPR2:GPR10, GPR12, FPR0:FPR13, всего 11 + 14

Non-volatile: GPR1, GPR11(*), GPR13:GPR31, FPR14:FPR31, всего 21 + 18 GPR11(*) — non-volatile для листовых функций (из которых нет других вызовов)

Соответственно, чем больше регистров в наличии, тем больше забот с сохранением их содержимого при вызове функций.

32 регистра общего назначения как раз является компромиссным значением для соблюдения баланса между производительностью вычисления выражений и лёгкостью вызова функций.

Процессоры с регистровыми окнами (Berkeley RISC: i960, SPARC, AMD29K). У этого класса выполняемая функция имеет доступ только к регистрам своего т.н. “окна”. Окно состоит из трёх частей: in – параметры вызова текущей функции, local – внутренние данные и out – параметры вызова дочерней функции. При вызове окно сдвигается и out часть вызывающей стороны становится in частью вызванной.

Сами регистры при этом организованы как кольцевой буфер. По мере роста стека окон, данные выталкиваются в память - (SPILL). Если стек регистров недо-заполнен, данные подгружаются из памяти (FILL).

У представителей этого семейства обычно много регистров, например, в AMD29K 64 глобальных и 128 локальных (стековых) регистров. В архитектуре SPARC 8 глобальных регистров и произвольное число локальных, кратное размеру окна (24). Типично 5 окон – всего 128 регистров. Изначально предполагалось, что дешевые процессоры в линейке будут содержать мало окон, а дорогие – много. Собственно, буква S в SPARC означает Scalable.

В SPARC окно фиксированного размера, в AMD29K – нет, тем не менее, использование окон большого размера при вызове функций будет приводить к тому, что поток данных в/из памяти вырастет и сведёт на нет весь выигрыш от большого количества регистров в контексте функции.

Переход на 64-разрядную архитектуру. Отличный способ найти применение всем тем дополнительным транзисторам, которые прогрессирующая технология позволяет разместить на кристалле.

Предыдущий переход с 16 разрядов на 32 дал значительный прирост производительности т.к. реальные данные с которыми приходится иметь дело далеко не всегда укладываются в 16 разрядов, эмуляция 32-разрядных операций дорога, кроме того постоянно приходилось следить за потенциальным переполнением.

Переход 32 => 64 не столь очевидно полезен, 32 разрядов вполне достаточно в большинстве случаев, а нагрузка на память возрастает. Пропускная способность памяти между тем – одно из основных узких мест в производительности.

Единственный бесспорный плюс от 64 разрядных архитектур – отсутствие ограничений на размер виртуального адресного пространства, ранее ограниченного 2 Гб (31 разряд + столько же для ОС). Впрочем, для массовых пользователей это преимущество не было очевидно вплоть до конца нулевых годов.

А вот в области высокопроизводительных вычислительных систем (HPC) преимущества 64 – разрядных архитектур были бесспорны. Так или иначе, все производители микропроцессоров начали осваивать эту нишу.

Первым стал MIPS R4000 – появился в 1991 году, 100 МГц ядро, 1.3 млн транзисторов (8К+8К кэш), 2.3 млн для 16К+16К кэша (при 0.11 млн транзисторов для R3000 без кэша)

SPARC V9 (UltraSPARC STP 1030) – появился в 1994 г., 143 МГц, 5.2 млн транзисторов, 16К+16К кэш

HP PA-RISC 7100 – появился в 1992 г., 80 МГц, 0.8 млн транзисторов при кэше 2К + 1К

DEC ALPHA (21064-AA) – появился в 1992 г., 150 МГц, 1.68 млн транзисторов при кэше 8К + 8К. DEC бросила VAX и сделала сразу 64-разрядный RISC.

PowerPC (PPC 601) – появился в 1993 г. (полностью 64-разрядный процессор PPC 620 1997 г.), 80 МГц, 2.8 млн транзисторов при кэше 32К. Сделан альянсом Apple-IBM-Motorola на основе архитектур POWER и 68040. Motorola ради этой архитектуры бросила свои ветки 68K и 88K.

И с большим опозданием:

Itanium от Intel – появился в 2001 г., это VLIW процессор, об этом позже. Совместный проект Intel и HP.

AMD64 от AMD (Opteron) – появился в 2003 г., Intel (Xeon Nocona) – 2004 г., это развитие x86. Intel стала развивать эту ветку после того, как стало понятно что Itanium не пользуется успехом.

И, наконец, архитектурные изыски, использование параллелизма. Практически любая программа обладает некоторым параллелизмом. Два выражения, которые не зависят друг от друга по данным, могут быть вычислены параллельно.

Мы не будем рассматривать программный параллелизм, равно как и использование векторных вычислений. Первое просто вне темы данной статьи, а второе до сих пор относится к области магии. Векторные вычисления действительно могут заметно ускорить вычисления, но это требует ручной работы, вставке подсказок компилятору… Современные компиляторы пользуются для векторизации набором рецептов на разные случаи жизни подобно тому, как это происходило до того как был разработан алгоритм более-менее оптимального распределения регистров (работы Грегори Хайтина). А нас больше интересует насколько пара компилятор/процессор способна справиться с параллелизмом в обычной программе.

Если ранее мы рассматривали в основном процессоры, способные выполнять одну инструкцию за такт (скалярные), то теперь нас интересуют суперскалярные, т.е. выполняющие несколько инструкций параллельно.

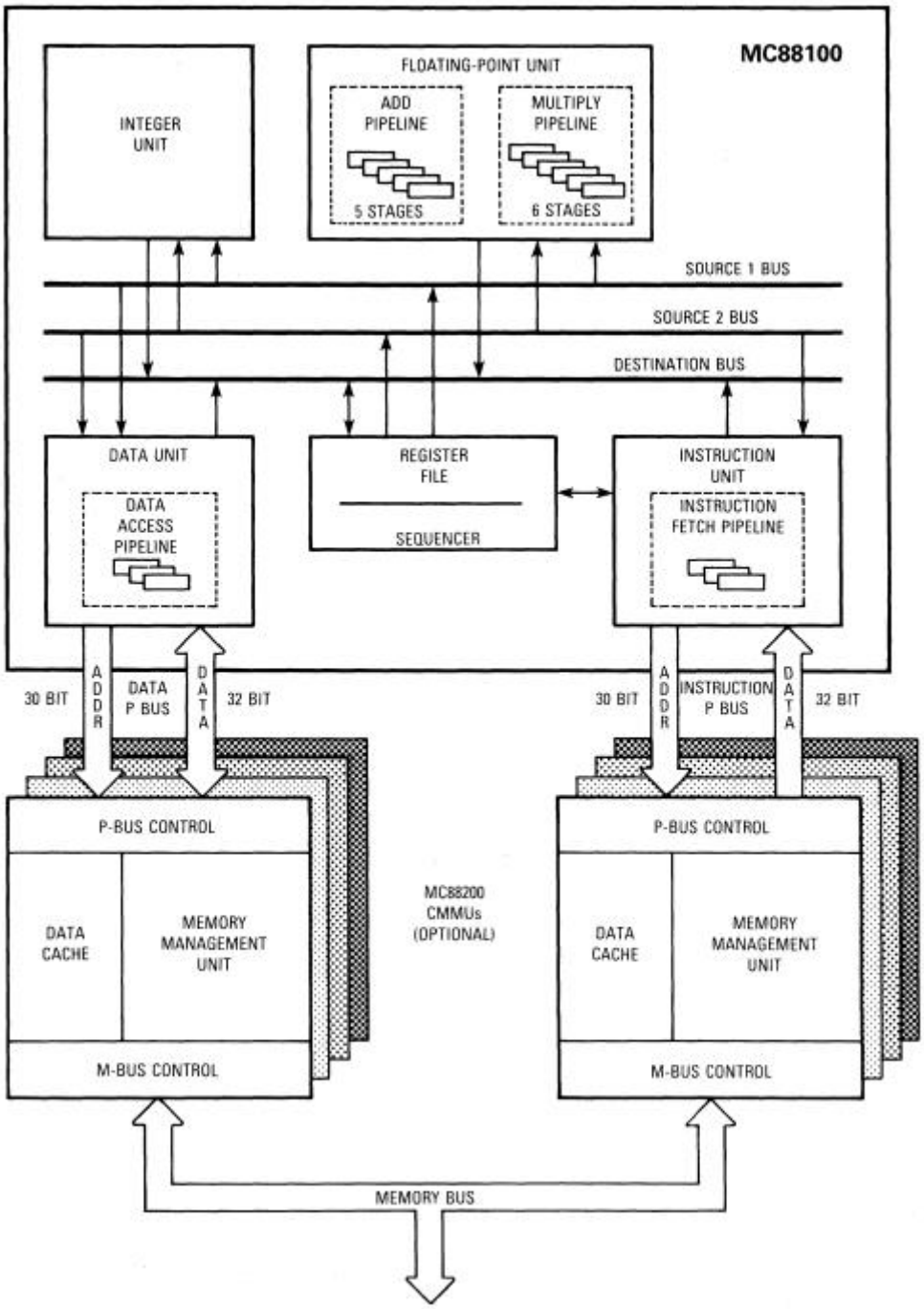
Основные суперскалярные идеи

Мы здесь не будем рассматривать “поползновения вширь” в виде векторных вычислений и/или многопроцессорные(многоядерные) вычисления.

Векторизация, с точки зрения автора, это проявление идеологического тупика, неспособность компилятора и процессора выявить и использовать параллельность в коде. Векторизация желательна в отдельных нишевых задачах, где возможна и полезна оптимизация кода вручную. В случае процессоров общего назначения это не так.

Работа в многопроцессорном режиме (SMP, NUMA, когерентность памяти …) это совершенно другая тема для разговора, очень важная, но другая. Здесь же разберём ситуацию – один процесс, один поток, один процессор (ядро).

Несколько специализированных конвейеров вместо одного общего. RISC конвейер даже в идеальном случае не способен выполнять больше одной инструкции за такт. Кроме того, не все операции могут быть выполнены за один такт, например, умножение или деление. В результате, во время исполнения длинной операции остальные стадии конвейера простаивают. Поэтому возникла мысль разбить общий конвейер на несколько меньших по функциональному предназначению. Рассмотрим на примере Motorola 88K (1988 г.)



Итак, процессор содержит пять разных конвейеров:

1) fetch/decode, который распределяет инструкции по остальным конвейерам

2) доступ к памяти

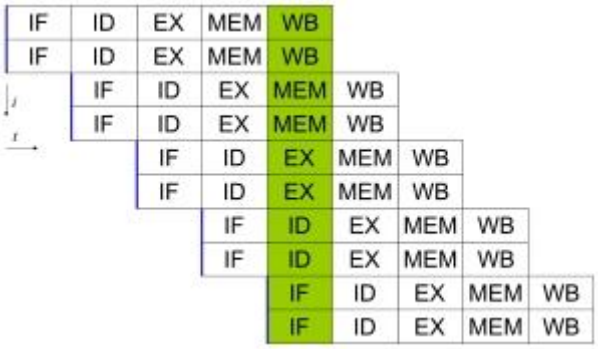
3) АЛУ (однотактный)

4,5) сопроцессор с плавающей точкой (+ целочисленные умножения и деления).

Т.е. теоретически 88100 мог выполнять до 5 инструкций одновременно.

Такую структуру можно рассматривать как единый обобщенный конвейер, без линейного прохождения всех стадий. Это, скорее, машина состояний, где порядок прохождения стадий зависит от самой инструкции. В принципе, таковым является и обычный RISC конвейер (Фиг. 6), в котором присутствуют стадии вычисления (EX) и доступа к памяти (MEM). RISC инструкция может или обращаться к памяти или выполняться, нет смысла тратить лишний такт на несуществующий этап.

Несколько основных конвейеров



Типичный пример – Pentium [9] (1993), у которого.

Параллельно работающие конвейеры U и V (аналогичные оному из 486).

Некоторые инструкции являются сочетаемыми (pairable) для U&V: MOV регистр, память или целое (immediate) в регистр или память, PUSH регистр или целое, POP регистр, LEA, NOP, INC, DEC, ADD, SUB, CMP, AND, OR, XOR, некоторые виды TEST

Некоторые инструкции являются сочетаемыми для U: ADC, SBB, SHR, SAR, SHL, SAL с целым, ROR, ROL, RCR, RCL на 1

Некоторые инструкции являются сочетаемыми для V: близкие (near) переходы, включая условные и вызов близких функций

Две последовательные инструкции выполняются параллельно, если одна из них сочетается с U, вторая с V и вторая инструкция не работает с регистром, в который пишет первая

Итого, некоторые целочисленные операции могут выполняться одновременно, что делает Pentium суперскалярным процессором. Тем не менее, такая конструкция не прижилась в том числе потому, что никак не помогает в борьбе с основным узким местом – работе с памятью.

Внеочередное выполнение инструкций (Out-of-order execution, OoO)

Исполнение инструкций по готовности требующихся функциональных устройств, а не в порядке следования в коде. Интуитивно понятная идея, которая была реализована довольно рано – еще в CDC 6600 (1964 г.) и IBM System/360 Model 91 (только для вычислений с плавающей точкой, 1967 г.).

Scoreboard – техника, применённая в CDC 6600 для OoO.

После распаковки инструкции:

1) строится граф зависимостей (в виде таблицы) от регистров и функциональных устройств (scoreboard)

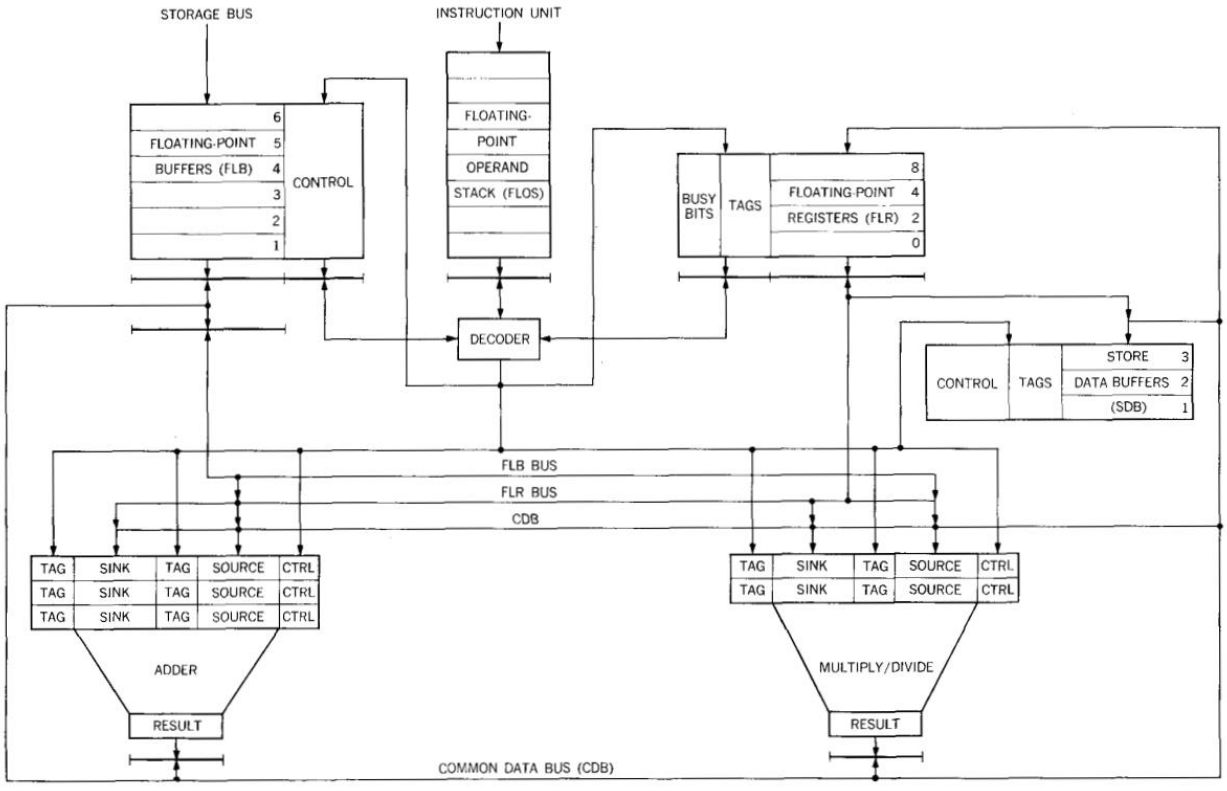
2) инструкция ожидает, когда освободятся все её зависимости

3) после исполнения, scoreboard уведомляется, что ресурсы освободились

Переименование регистров (register renaming).

Альтернативная методика от IBM. Разберём подробно, это важно.

В архитектуре IBM 360 всего 4 регистра с плавающей точкой двойной точности и это сильно ограничивало генерацию эффективного кода компилятором. Ведь код должен был быть совместимым с другими моделями. В 1965 г. Роберт Томасуло (Robert Tomasulo) предложил [8] алгоритм/архитектуру блока для работы с плавающей точкой (Фиг. 11), названную в его честь.



Floating point operation stack (FLOS) – очередь инструкций в хронологическом порядке.

Store data buffers (SDB) – устройство выгрузки содержимого регистров в память.Содержит очередь из трёх адресов.

Floating point buffers (FLB) – устройство загрузки регистров из памяти, содержит очередь из 6 адресов.

Floating point registers (FLR) -пул регистров из четырёх элементов.

Adder – сумматор, их три штуки.

Multiply/Divide – умножитель/делитель, две штуки.

Три сумматора и два умножителя называются станциями резервирования (reservation stations), у каждой из пяти станций есть два аргумента – sink & source (названия обусловлены тем, что инструкции IBM 360 были двухадресными, например, AD F0, F2 означает F0 + F2 => F0, F0 при этом называется sink (сток), а F2 – source (источник)).

CDB – общая шина данных. Все, кто могут изменить содержимое регистра, являются поставщиками CDB, все, кто могут хранить содержимое регистров, являются слушателями CDB.

Все поставщики CDB пронумерованы, так ячейки FLB имеют номера от 1 до 6, два умножителя – номера 8 и 9, три сумматора – 10…12. Значение 0 зарезервировано за FLR (пул регистров). Всего 11 поставщиков, с номерами от 1 до 12 кодируются 4-разрядным тегом (tag).

Теги есть у каждого из четырёх регистров FLR, у sink и source аргументов каждой из пяти станций и у трех буферов устройства выгрузки SDB, всего 17 тегов.

Теги есть у всех, кто слушает шину CDB. Когда поставщик собирается передать через CDB некоторое значение, например, сумматор закончил работу и готов выдать результат, он записывает в CDB и свой идентификатор (тег) и сами данные. Все слушатели (подписчики), чей тег совпал с транслируемым, записывают себе новое значение и предпринимают необходимые действия.

Например, если транслируемый тег совпал с таковым у одного из буферов блока выгрузки SDB, тот начинает запись этого значения в память по ассоциированному с этим буфером адресу, после завершения записи переходит в неактивное состояние вплоть до получения нового адреса.

Как организуется подписка на данные? Этим занимается декодер (FLOS/decoder).

Декодер берет очередную инструкцию, проверяет, доступно ли сейчас нужное ей функциональное устройство(а), если нет, пропускает такт. Например, если это инструкция суммирования, но все три сумматора заняты, остаётся только ждать

Некоторые инструкции распадаются на несколько, например, суммирование значения в регистре со значением в памяти превратится в две микро-инструкции – сначала требуется загрузить значение из памяти в один из буферов FLB, затем передать в сумматор значение регистра и буфера FLB. Передача из буфера осуществляется посредством подписки через шину CDB

Все слушатели шины CDB могут быть в двух состояниях – или содержать готовое, уже полученное значение или ожидать его получения через CDB. Как только функциональное устройство получает по подписке через CDB все необходимые аргументы (в случае сумматора и умножителя аргументов два, у устройства выгрузки SDB один), оно приступает к работе. Устройству загрузки FLB для начала загрузки буфера требуется лишь указать адрес в памяти.

Теги пула регистров FLR кроме подписки на CDB выполняют еще одну функцию. По ним декодер определяет где в данный момент (после декодирования очередной инструкции) находится значение конкретного регистра. Если тег нулевой, значение можно брать прямо из регистра FLR, при этом соответствующий аргумент функционального устройства считается готовым к использованию. В противном случае тег копируется в аргумент, который переводится в статус ожидающего CDB.

Если декодируемая инструкция меняет содержимое регистра (загрузка из памяти или sink аргумент), то декодер присваивает тегу данного регистра в FLR номер использованного функционального устройства. По завершении данной инструкции вычисленное значение будет помещено в этот регистр по подписке или утеряно, как ненужное. Если какой-либо инструкции в дальнейшем потребуется значение этого регистра, она не обязана дожидаться пока значение доедет до регистра, а сможет перехватить значение непосредственно на выходе из функционального устройства.

Пример 1.

Суммирование двух чисел из памяти в регистр, проверяем синхронизацию по F0

```
LD F0, addr1   ; значение из addr1 => F0
AD F0, addr2   ; значение из addr2 + F0 => F0
```

При разборе первой инструкции декодер

Найдёт свободный буфер (пусть 1) в блоке загрузки FLB,

Задаст ему адрес загрузки,

Запишет его тег (1) в тег регистра F0 FLR.

FLB начинает загружать буфер 1.

При разборе второй инструкции декодер

Найдёт свободный сумматор, пусть 1, его тег равен 10

Запишет тег sink аргумента сумматора 1, который возьмет из тега F0 FLR, это 1

Найдёт свободный буфер (пусть 2) в блоке загрузки FLB, задаст ему адрес загрузки, запишет его тег (2) в тег source аргумента сумматора 1

Запишет тег сумматора 1 (10) в тег регистра F0 FLR

FLB начинает загружать буфер 2.

Через несколько тактов загрузится буфер 1 FLB и запишет свой тег (1) в CDB. Его ждёт sink аргумент первого сумматора, который скопирует себе значение и перейдёт в состояние готовности. Далее загрузится буфер 2 FLB и запишет свой тег (2) в CDB. Его ждёт source аргумент первого сумматора, который скопирует себе значение и перейдёт в состояние готовности. Теперь оба аргумента сумматора 1 готовы и он начинает работать. По завершении в CDB будет записан тег (10) и значение, их ждёт регистр F0 FLR, который сохранит значение и сменит свой тэг на 0.

В F0 значение суммы, что и требовалось.

Пример 2. Здесь мы должны убедиться, что значение F2 не будет испорчено если вдруг последняя инструкция выполнится перед предпоследней

```
LD F4, addr1 ; значение из addr1 => F4
LD F2, addr2 ; значение из addr2 => F2
LD F0, addr3 ; значение из addr3 => F0
AD F0, F2    ; F2 + F0 => F0
AD F2, F4    ; F4 + F2 => F2
```

При загрузке данных будут использованы буфера FLB с тегами 1, 2, 3 и эти теги пропишутся в пул регистров FLR.

Первая инструкция суммирования заберет сумматор 1, в его sink аргумент пропишется тег 3, в source – тег 2, в тег регистра F0 попадёт 10 (сумматор 1). Вторая инструкция суммирования заберет сумматор 2, в его sink аргумент пропишется тег 2, в source – тег 1, в тег регистра F2 попадёт 11 (сумматор 2).

Т.к. данные будут приезжать в порядке F4 – F2 – F0, второе суммирование выполняется первым, но никакого конфликта нет, ведь обе инструкции подписаны на один буфер FLD.

Пример 3.

Цикл суммирования без зависимости по данным C[i] = A[i] + B[i]

```
LOOP LD   F0, A(i)      ; загружаем из массива A i-й элемент в F0
     AD   F0, B(i)      ; суммируем его с i-м элементом массива B
     STD  F0, C(i) STORE; сохраняем сумму в i-й элемент массива C
     BXH i, -1, 0, LOOP ; цикл по i в порядке убывания
```

Первые две инструкции это пример 1.

Инструкция сохранения STD подписана на результат сумматора 1 и начинает работать сразу как оно завершится.

Инструкция BXH (Branch on indeX High) уменьшает значение счетчика и направляет декодер на метку LOOP с загрузкой из массива A.

Декодеру не важно, загрузились ли данные, поэтому он продолжит работать пока у него есть свободные устройства. Он продолжит распределять инструкции второй итерации цикла несмотря на то, что еще не выполнилась первая. Он назначит загрузку элементов данных во третий и четвёртый буфера FLB, использует второй сумматор для суммирования.

То же произойдёт и с третьей итерацией цикла, на четвёртой закончатся и буфера загрузки и сумматоры.

Т.е. три итерации цикла могут исполняться параллельно. Стоит обратить внимание, тег регистра F0 постоянно меняется при декодировании инструкций LD и AD, фактически, значение попадёт в регистр только в последней итерации цикла.

Итак, подведём итоги рассмотрения работы алгоритма Томасуло.

Алгоритм успешно справляется с конфликтами чтения/записи.

Он обнаруживает зависимости по данным и позволяет исполнять инструкции параллельно и вне очереди.

Очень эффективно расширяется такое узкое место архитектуры, как маленькое число доступных регистров. Особенно это касается CISC архитектур. При 4 официальных регистрах, к ним добавилось 16 неофициальных (6 в сумматорах, 4 в умножителях и 6 на загрузке). Думается, если бы у IBM 360 изначально было 16 регистров с плавающей точкой двойной точности, такой замечательный механизм в ней вряд ли появился бы.

Позволяет масштабировать архитектуру, увеличивая число функциональных устройств. Например, вышеописанное устройство (пример 3) выполняет параллельно до трёх итераций цикла. Но если бы сумматоров было 4, а буферов загрузки 8, смогли бы исполняться по 4 итерации цикла одновременно (без перекомпиляции).

Поддерживает истинную асинхронность. На момент создания IBM 360/91 не применялась кэш память, но даже в схеме с кэшированием и непредсказуемыми временами доступа к данным, алгоритм успешно работает, ему не требуются дополнительные задержки для синхронизации. От функциональных устройств так же не требуется фиксированное число тактов для работы. Пусть умножение в общем случае занимает 5 тактов, но умножение на 1, 0 или степень 2 вполне можно выполнить и за такт. В данном случае это ускорение будет использовано максимально эффективным образом.

Данный алгоритм можно использовать и для архитектур с пулом внутренних регистров. Т.е. станции резервирования и функциональные устройства содержат не вычисленные значения регистров (или тег), а а индекс регистра во внутреннем пуле (и тег). Этот вариант называется pointer based register renaming, в отличие от вышеописанной value based схемы.

Неудивительно, что алгоритм Томасуло был подхвачен повсеместно (когда пришло время, Ex: DEC 21064 (1992), Pentium Pro (1995), MIPS R10000 (1996)) и в том или ином виде используется до сих пор. Кстати, спустя 30 лет (в 1997 г.) Роберт Томасуло за свой алгоритм получил премию Эккерта – Мокли.

Предсказание переходов

Представим ситуацию, когда устройство fetch/decode натыкается на инструкцию безусловного перехода. Чтобы не останавливать работу, было бы полезно заранее знать, что это операция перехода и начать готовиться – как минимум загружать код. Для этого надо знать адрес перехода, по возможности, до того, как сама инструкция перехода декодирована. В этом случае (в идеале) можно обойтись вообще без задержек.

Не менее горячо желание не простаивать и в случае условного перехода, даже если условие перехода на данный момент не известно (не вычислено). Что тут можно сделать?

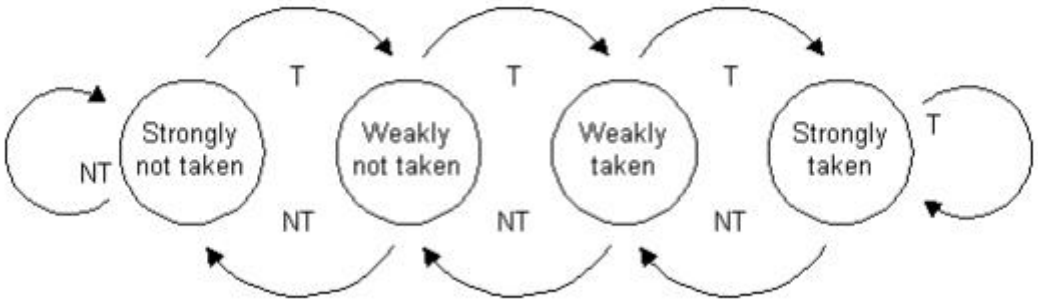
Если мы оказались в этом месте первый раз, то ничего. Предсказания делаются только на основе предыдущего поведения программы. Но в каких случаях эти предсказания имеют приемлемый шанс сбыться?

Во первых, это циклы. Ранее мы видели как IBM/360 использует для организации циклов специальную инструкцию (срисованную из FORTRAN-a), по которой можно точно сказать будет ли новая итерация или последует выход из цикла. В более современных системах циклы организованы менее очевидно, но предсказать их всё же нетрудно. Например, с помощью хранения истории переходов в сдвиговом регистре (BHR, branch history register). Такой регистр после каждого перехода сдвигает своё содержимое вверх на разряд а в младший разряд помещает информацию об условном переходе – 1 если он произошел (taken) или 0 если нет (not taken). Если такой регистр содержит одни единицы, логично предположить, что и на этот раз переход произойдет. Косвенно, о том что мы имеем дело с циклом, можно судить по переходу по адресу назад от текущего.

Во вторых, значительная часть условных операторов всегда (или почти всегда) делают один и тот же выбор. Например, проверка кода ошибки. Поскольку ошибки происходят относительно редко, обработка этих ошибок делается тоже не очень часто. Если же мы имеем дело со сложным потоком управления и/или конечным автоматом, на высокую точность предсказаний рассчитывать не стоит.

В третьих, возврат из функции. Он производится специальной инструкцией по адресу, где мы уже были и даже небольшой кэш в виде стека адресов возврата способен делать предсказания с очень хорошей точностью. Этот механизм называется RAS (Return Address Stack) и, например, в случае Alpha 21264 состоит(ял) из 12 элементов, обеспечивая правильное угадывание адреса в 85% случаев.

Итак, успешное предсказание перехода ожидается только для циклов и (более-менее) статических условных операторов (возврат из функции – отдельный случай, который обрабатывается отдельно). В условиях бесконтекстного предсказания (т.е. опираясь только на историю taken/not taken, без учета адреса инструкции перехода) возможны лишь очень простые и логичные конструкции – например, чего было больше – taken или not taken.



Лучше всего себя показывают т.н. счётчики с насыщением (saturating counters). Например, пусть предсказание определяется знаком счетчика (SC) – если больше 0, то taken, меньше 0 – not taken.

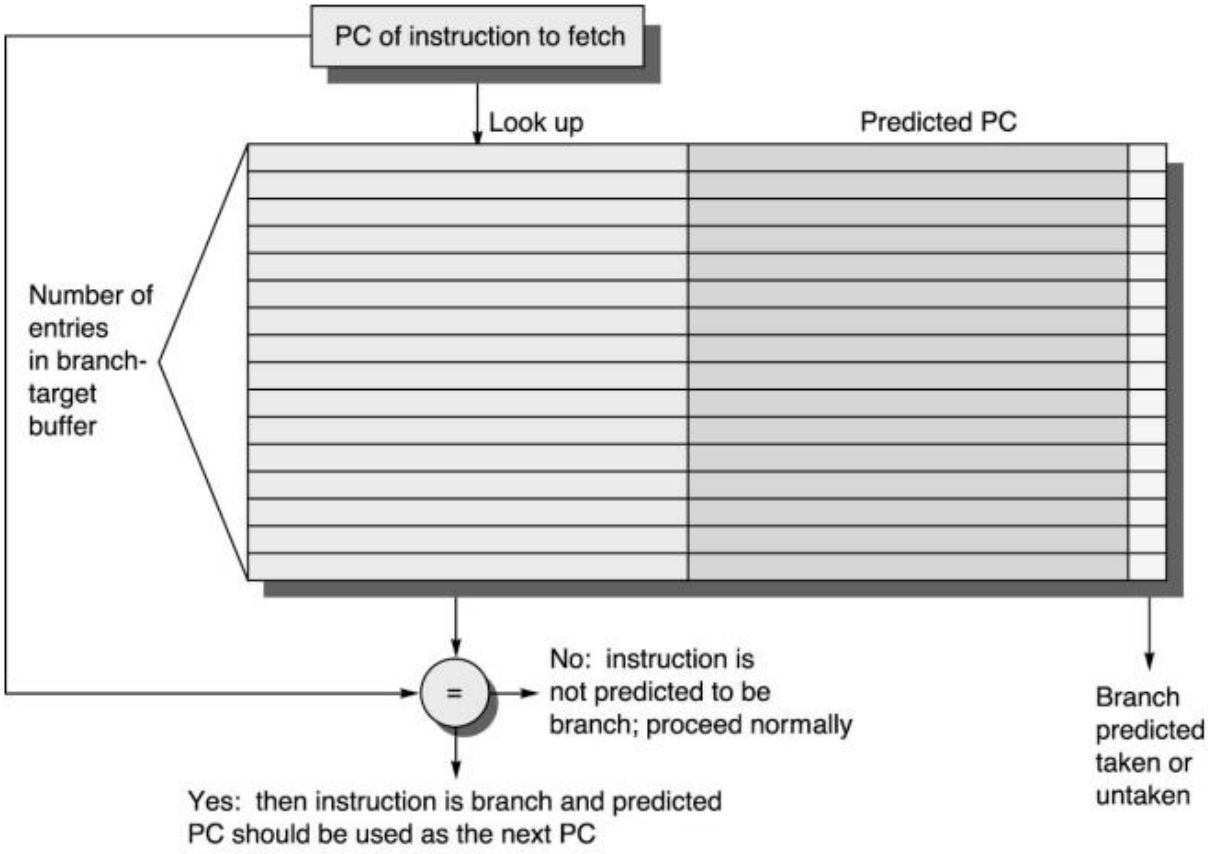
Значение счетчика находится в интервале -3/2 <= SC <= 3/2.

Значение меняется с шагом 1, если фактический переход был taken, то единица прибавляется (в пределах интервала), если not taken, единица отнимается.

Из всех счетчиков с насыщением, как ни странно, лучше всего работают именно показанные на Фиг. 12 двухразрядные т.е. с четырьмя состояниями.

Понятно, что такая простая бесконтекстная конструкция не может работать эффективно, допустим, в случае цикла с двумя ложными проверками условных операторов, переход на следующую итерацию не будет предсказан.

Выход заключается в хранении контекста условных переходов, привязанных к адресу этих переходов. Осталось понять, где всё это хранить. Было бы логично разместить контекст рядом с кодом – дополнительными тегами кэша инструкций первого уровня, но вот, например, адреса переходов (которые потребуются в случае предсказания taken) там не поместятся – слишком большие. Поэтому, используют небольшую ассоциативную память (BTB, branch target buffer).



BTB использует значение хэш функции от адреса как ключ для хранения и поиска, сам адрес как тег для сравнения. Кроме тега и предсказанного адреса в элементе BTB содержится информация для предсказателя переходов. Иногда это просто те самые два разряда счетчика с насыщением, иногда что-то более сложное, например, массив счетчиков, индексом в котором служит значение сдвигового регистра BHR. Такая двухуровневая конструкция лучше адаптируется к сложному коду.

Рассмотрим на примере PentiumPro (1995 г.) [9, стр.21].

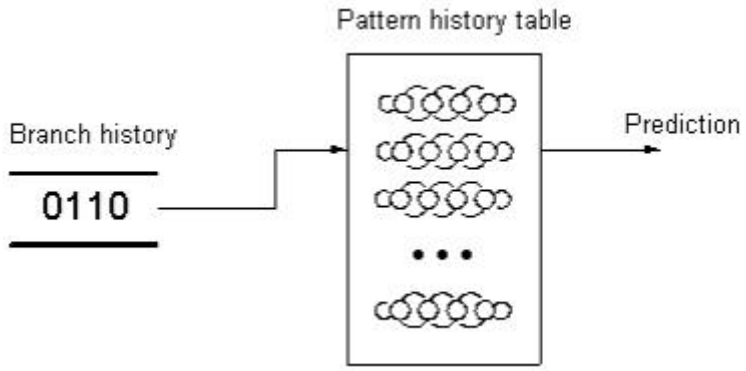
BTB состоит из 512 элементов, ассоциативность равна 16

Разряды адреса 4…7 образуют индекс корзины BTB.

Весь адрес хранится как тег для сравнения внутри корзины

При записи в таблицу, вытесняется случайный элемент

Информация предсказателя – массив из 16 двухразрядных счетчиков, индексом в котором служит 4-разрядный BHR



Условные инструкции

Поскольку условный переход – недешевое удовольствие, в некоторых архитектурах существуют т.н. условные инструкции – те, что исполняются только если выполнено определенное условие, например, выставлен определенный флаг в регистре флагов. Если же условие не выполнено, инструкции игнорируются. В таких архитектурах, как MIPS (начиная с MIPS IV), SPARC (V9) присутствует т.н. условная пересылка регистра.

А в архитектуре ARM допустимы целые условные ветки, например

```
; флаги установлены ранее
LSLEQ r0, r0, #24
ADDEQ r0, r0, #2
;…
```

Вместо

```
    ; флаги установлены ранее
    BNE over
    LSL r0, r0, #24
    ADD r0, r0, #2
Over
    ;…
```

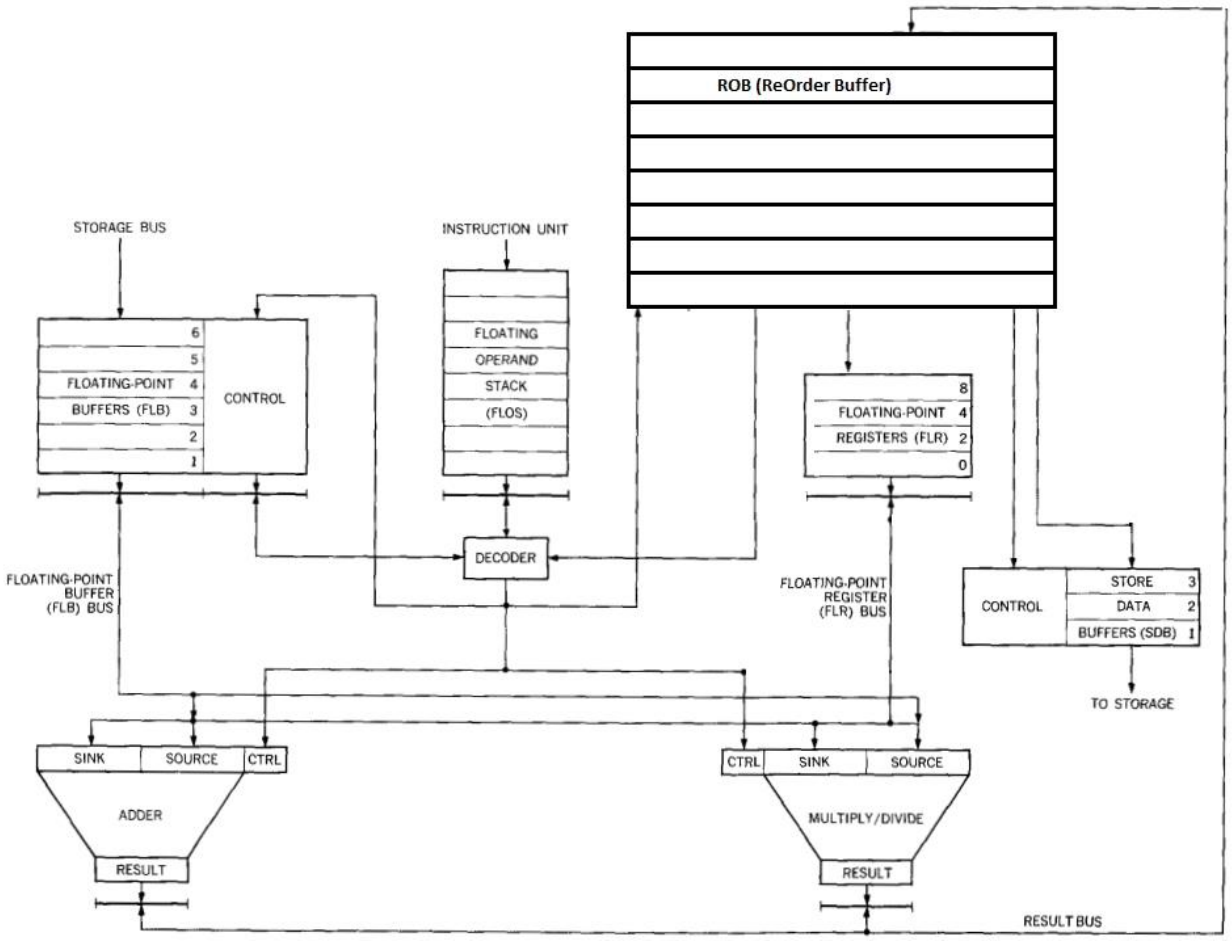
Спекулятивное (по предположению) исполнение

Пусть у нас есть предсказание перехода.Если мы ему доверяем, то возникает желание начать исполнять предсказанную ветку, во всяком случае, независимые по данным её части.Однако, предсказание может и не сбыться, в связи с чем возникает потребность исполнять код “условно” (обе ветки перехода, почему нет) и при необходимости отменять результаты его работы.

Оказывается, можно модифицировать алгоритм Томасуло и он будет удовлетворять этим требованиям. Это похоже на механизм транзакций в базах данных – к уже существующим стадиям жизненного цикла инструкций (выборка (fetch), декодирование, исполнение (execution), запись результатов(write back)) добавляется еще одна – фиксация (retirement). Если инструкция имеет эффект (запись в регистр, в память или исключение (exception)), то ее результат хранится во временном хранилище до тех пор, пока не станет ясно, пошло ли исполнение по этой ветке кода или нет. Если нет, то результаты игнорируются, если да, то происходит запись в регистр, память или выброс исключения (деление на 0 ?).

Таким хранилищем временных данных является новый блок под названием ROB (ReOrder Buffer).ROB состоит из т.н. микрокоманд (μops). Нет однозначного соответствия между входными командами и μops, так, например, команда x86: add eax, [esi+20] порождает две микрокоманды – первая читает память, вторая арифметическая – суммирует и записывает EAX (+флаг).

ROB выполняет еще одну важную функцию – он вместо контрольного блока пула регистров теперь распоряжается информацией значение какого регистра в какой ветке где физически находится в данный момент. В PentiumPro эта часть ROB называется RAT (Register Alias Table).



ROB обычно устроен как кольцевой буфер – во время декодирования/диспетчеризации резервируются и заполняются новые микрокоманды, с другой стороны по мере исполнения и фиксации они освобождаются. Если ROB полон, декодирование приостанавливается.

В чём отличия модифицированного алгоритма Томасуло от того, что рассматривалось ранее?

При декодировании дополнительным критерием продолжения работы является наличие свободных элементов ROB (в дополнение к станциям резервирования и т.д.)

Микрокоманды записываются в порядке их появления т.е. in-order, хотя исполняться могут и out-of-order, после записи указатель вершины списка актуальных микрокоманд сдвигается вперёд.

Пул регистров FLR больше не слушает шину результатов CDB, за него это делает ROB. Т.е. декодер подписывает на результат работы микрооперации не регистр, а саму микрооперацию, в ней должно быть достаточно места чтобы разместить результат своей работы..

То же самое и с модулем выгрузки в память. Если в процессе декодирования возникает инструкция выгрузки, именно она будет дожидаться результата вместо слота SDB

Каждая микрокоманда знает к какой ветке кода она относится. Всегда есть безусловная ветвь кода – та, между инструкциями которой и последней зафиксированной микрокомандой нет условных переходов. Плюс возможны условные ветви – которые соответствуют незафиксированным условным переходам. Теоретически, чтобы добраться до конкретной микрокоманды может потребоваться целый стек неисполненных условных операций, на практике этот стек очень небольшой глубины.

Указатель на хвост списка актуальных микрокоманд сдвигается при фиксации. Если он смотрит на микрокоманду из безусловной ветки, та просто фиксируется. Если же это команда условного перехода, определяется направление перехода, выбранная ветка становится безусловной, противоположная ей игнорируется. Т.е. указатель хвоста ROB пропускает микрокоманды из этой ветки (flush).

Фиксация (commit) безусловных микрокоманд заключается в записи значения в регистр или память, а также инициировании исключения, если оно возникло и записано как результат работы микрокоманды.

Фиксация микрокоманд происходит в хронологическом (in-order) порядке вне зависимости от порядка их исполнения. Собственно поэтому ROB и называется буфером переупорядочивания.

Каков размер ROB? Для некоторых из процессоров Intel/AMD это: Ivy Bridge (168), Sandy Bridge (168), Lynnfield (128), Northwood P4 (126), Yorkfield (96), Palermo (72), and Coppermine P3 (40) (отсюда).

VLIW

Суперскалярные процессоры иногда упрекают в чрезмерной сложности и это отчасти справедливо. Вызвана эта сложность наличием “великого водораздела” – системы команд. Система команд (в сущности формальность, иллюзия) изолирует внешнее представление суперскалярного процессора от его внутреннего устройства. Квинтэссенцией системы команд являются регистры общего назначения. Именно на их число ориентируется компилятор когда распределяет (allocate) регистры. Немало усилий прикладывается чтобы протащить (как канат в игольное ушко) естественный параллелизм программы через ограниченное число регистров.

С другой стороны, процессор (декодер) в меру своих возможностей старается понять что же имел ввиду компилятор и восстановить этот исходный параллелизм.

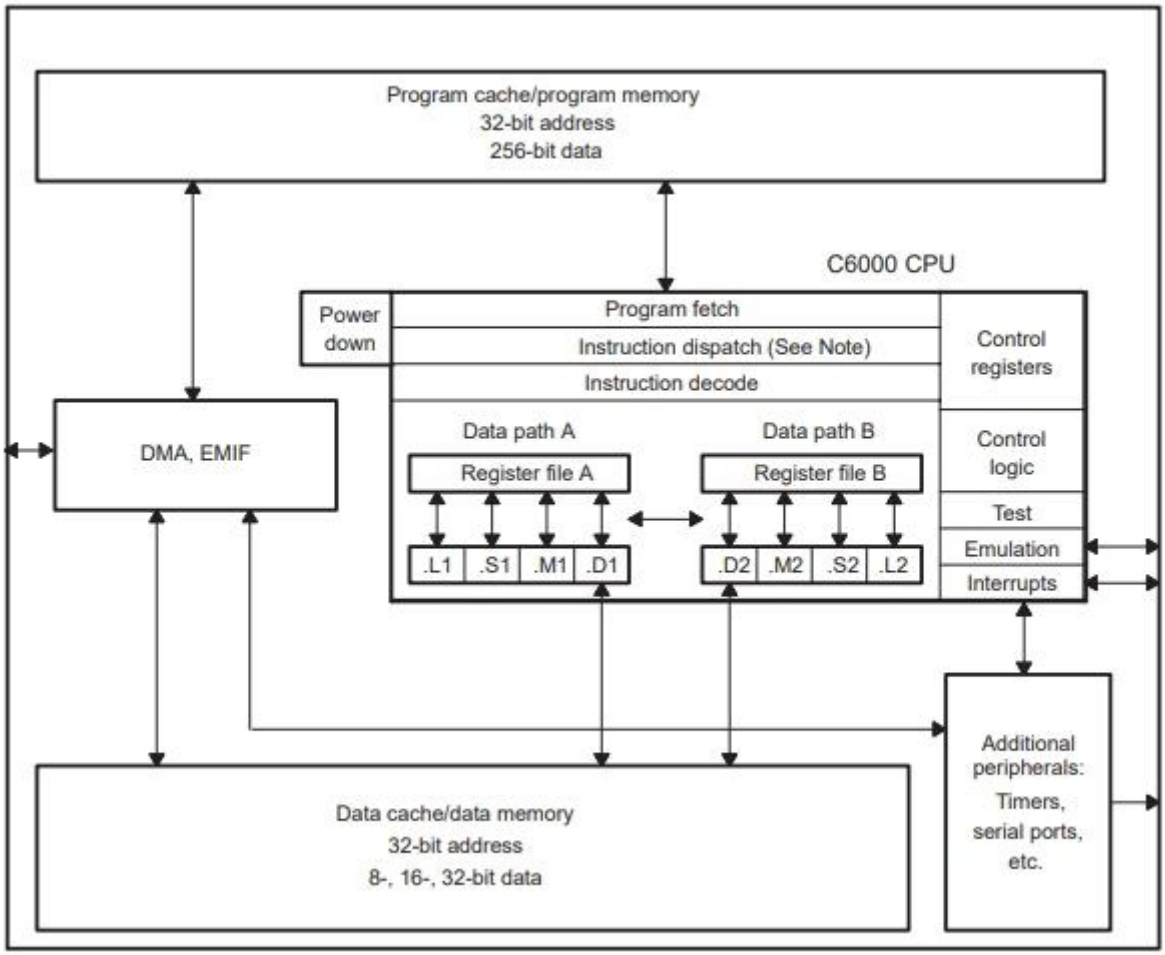
С третьей стороны, именно наличие этого водораздела позволяет, насколько это возможно, полно использовать функциональные ресурсы процессора. В результате мы имеем фантастическую совместимость сверху вниз, когда старая программа на допотопной системе команд без видимых усилий работает на новом процессоре быстрее роста тактовой частоты.

С четвертой стороны, столь всеобъемлющая совместимость не всегда и нужна, достаточно применений, когда не составляет труда перекомпилировать программы при необходимости. В частности, в обработке сигналов (DSP), суперкомпьютерных вычислениях.

Из необходимости исполнять при этом несколько команд параллельно родилась техника, известная нам как VLIW (Very Long Instruction Word). В ней на компилятор ложится ответственность за управление ресурсами процессора. И каждая иструкция фактически является пачкой микрокоманд, которые должны стартовать в один такт времени.

Реализована эта идея впервые была в области мини-суперкомпьютеров. Первыми компьютерами с широким командным словом были Multiflow TRACE (1987 г.), Cydrome (1988 г.), продукты на основе Intel i860 (1989 г.). Последний просуществовал дольше всех и гордо назывался “Cray on a Chip” (по-видимому, маркетологами), хотя, справедливости ради, у Cray Research действительно был продукт на его основе.

Рассмотрим идею на примере заслуженного сигнального процессора от Texas Instruments TMS320C6x, появившегося в 1997 г.



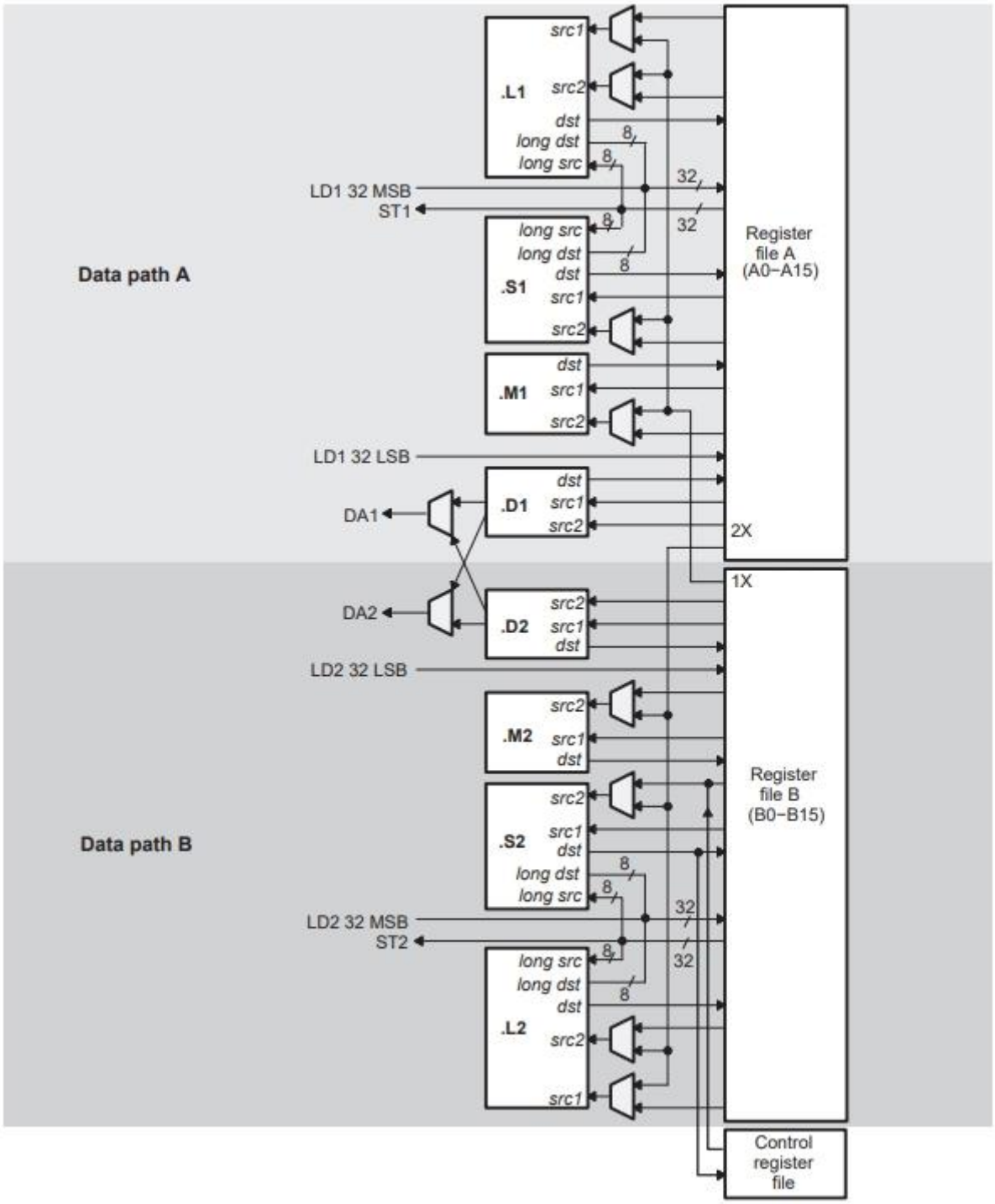
Процессор имеет два пути (data path) A & B. Каждый из них содержит свой пул регистров и функциональные модули .L, .S, .M и .D.

.L выполняет 32 и 40 разрядные арифметические операции и сравнения, преобразования DP → SP, INT → DP, INT → SP

.S выполняет некоторую арифметику, битовые операции, ветвление, преобразования SP → DP, генерацию констант, итерации получения обратной величины …

.M – умножитель

.D – вычисление адресов, загрузка из памяти и выгрузка в неё …



Явный параллелизм.

При восьми как бы независимо работающих функциональных устройствах процессор имеет возможность запускать на исполнение до восьми инструкций за такт. Слова “как бы” отражают некоторые зависимости между устройствами, например, один регистр нельзя читать более четырёх раз параллельно. Если в силу ограничений невозможно разместить в пакете содержательные инструкции, пустые места в пакета заполняются пустыми инструкциями NOP.

Одна инструкция занимает 32 разряда и код читается пачками по 8 инструкций за такт (fetch packet). Суммарно это как раз ширина шины данных а также размер линии кэша.

Младший разряд каждой инструкции – p(arallel) bit. Он сигнализирует, выполняется ли эта инструкция параллельно с правой от неё.

Пакет с Фиг. 19 будет исполняться в следующем порядке

Цикл

Инструкция

1

2

3

4

A

B

C D E

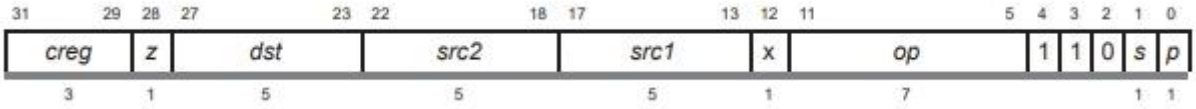
F G H

.На языке ассемблера это соответствует коду

```
        Instruction A

        Instruction B

        Instruction C
||      instruction D
||      instruction E

        Instruction F
||      instruction G
||      instruction H
```

Пара символов ‘|’ означает что данная инструкция параллельна предыдущей. И, конечно, инструкции C,D,E и F,G,H не должны использовать одинаковые функциональные устройства и нарушать другие явные и неявные ограничения.

Ветвление, условные операции.



Большинство инструкций могут быть условными. Для этого в формате инструкций присутствуют два поля – трёхразрядное creg и одноразрядное z. Z – значение для сравнения с предикатом – ноль или не ноль. Предикатом может быть один из пяти регистров (A1, A2, B0, B1, B2). Поле creg определяет, со значением какого именно предиката надо сравнивать z и сравнивать ли вообще (если creg==0, это безусловная инструкция). При ветвлении, инструкции разных веток могут идти вперемешку, эта техника называется предикативность (predication) и она заменяет спекулятивное исполнение.

Ограничения.

Никакие две инструкции не могут использовать одновременно одно функциональное устройство. Обычно функциональное устройство исполняет инструкцию несколько тактов, после чего следует цикл записи. На протяжении всех этих тактов никакая другая инструкция не может претендовать на это устройство, даже если она из другого пакета параллельных инструкций.

Запрещено параллельно писать в один регистр. Допустимы параллельные инструкции, которые пишут в один регистр при условии, что их циклы записи разнесены по времени – выполняются за разное число тактов или со сдвигом.

Разрешено не более 4 параллельных чтений из одного регистра, уже упоминалось об этом. Регистры A0, A1, B0, B1, B2 при использовании в ветвлении не учитываются при подсчетах.

Допустимо только одно чтение за цикл между регистровыми пулами (через 1X или 2X кроссы), допускается чтение разными инструкциями через кросс одного и того же регистра

Устройство .D при загрузке и выгрузке данных должно иметь в качестве аргументов только регистры из своего пула

При ветвлении разным веткам можно писать в один регистр, а использовать одно функциональное устройство – нет.

Устройства .S и .L используют общий входной порт при работе с 40-разрядными целыми, здесь тоже возможен конфликт

И т.д. и т.п.

За и против.

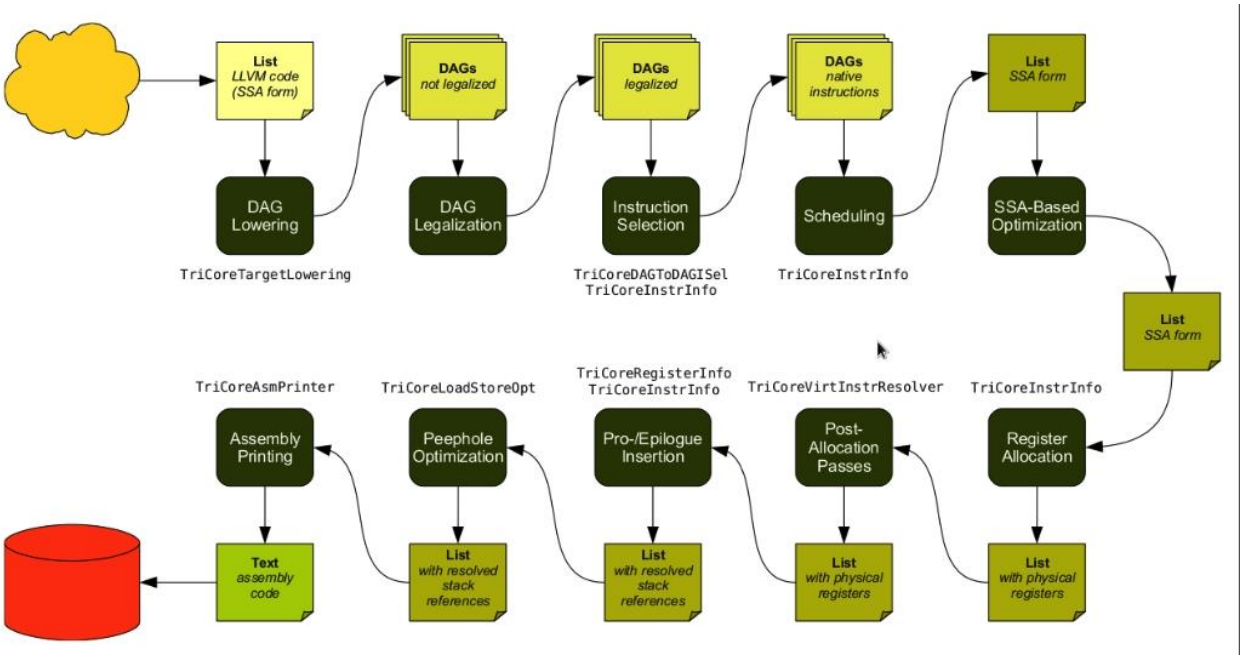
Процессоры с широким командным словом от суперскалярных отличаются отсутствием т.н. front-end'а – компилятор генерирует сразу микрокод, занимается статическим предсказанием ветвлений и статической же оптимизацией использования функциональных блоков. В них нет внеочередного исполнения инструкций, ни предсказания переходов, ни спекулятивного исполнения. В силу своей простоты VLIW процессоры потенциально способны работать на более высокой частоте и потреблять меньше энергии.

Однако статическая природа оптимизации накладывает на использование VLIW процессоров свой отпечаток. Там где суперскаляр работает по готовности, VLIW обязан рассчитывать на худший случай, например, при чтении из памяти обесценивается работа кэша. Альтернатива – рассчитываем на чтение из кэша, но в случае промаха придётся так или иначе синхронизировать исполнение – вставлять задержки. Иногда (ex: 1967ВН28) используют память, работающую на частоте процессора, чтобы он мог проявить себя. Понятно, что для процессора общего назначения это неприемлемо.

Со статическим предсказанием переходов беда не меньшая. Там где современный суперскалярный процессор легко вскрывает паттерны условных переходов и адаптируется к ним, VLIW может только подбрасывать монетку. Спекулятивное исполнение можно себе представить, но оно повлечет за собой переименование регистров и усложнение архитектуры вплоть до сопоставимых с суперскалярной. Поэтому предпочитают предикативный код и замешивание инструкций разных веток в одном потоке.

Еще одна беда – отсутствие масштабирования. Если в суперскалярном процессоре количество функциональных устройств скрыто от компилятора, то в случае VLIW компилятор должен знать фактическое устройство процессора и ориентироваться на него. Тот же самый код работает быстрее на суперскалярном процессоре с той же тактовой частотой и большим количеством (предположим) сумматоров. Но для VLIW это не так. Нужно ускорение – требуется перекомпиляция под новую архитектуру. В качестве примера – MCST разрабатывает уже седьмую версию архитектуры Эльбрус, число параллельно исполняемых инструкций при этом выросло с 23 до 50.

Нельзя не сказать также о сложности создания компиляторов для VLIW архитектур. Внутренний мир суперскалярного процессора скрыт от компилятора и задача оптимизации (очень грубо) сводится к распределению регистров. Задача это хоть и NP-полная (сводится к экспоненциально сложной задаче раскраски графа), но имеющая приемлемое полиномиальное решение.



На Фиг. 21 показаны стадии компиляции LLVM, распределение регистров – это маленький (но очень важный) блок в правом нижнем углу (отсюда).

Для VLIW всё неизмеримо сложнее. Казалось бы, по сравнению с декодером у компилятора несравненно больше ресурсов (в том числе времени), он должен выигрывать в качестве. Однако, в случае VLIW планировать нужно не только регистры, но и функциональные устройства, учитывая при этом различные ограничения. Здесь стоит пробежаться глазами по вышеприведённому списку ограничений для TMS320C6x и представить трудности компилятора. Для примера, более простая, хотя и NP-полная (т.е. экспоненциальная по сложности), задача об укладке рюкзака имеет ряд приближённых схем полностью полиномиального времени т.е. решения полиномиальной сложности, которые дают решения хуже оптимального не более чем на фиксированную величину. Но эти решения очень чувствительны к формулировке задачи. В частности, для двумерного рюкзака (multidimensional knapsack problem) таких решений уже нет.

Не исключено, что подходящая эвристика всё же существует, но стоит ожидать что она будет весьма чувствительна к малейшим изменениям в архитектуре. И при выпуске минимально изменённого процессора придётся переделывать/перевыпускать и компилятор к нему. Либо смириться с потерей производительности.

Еще раз отметим, проблемы VLIW компилятора носят фундаментальный характер, (почти) нет надежды что однажды вместо набора трюков, применяемых в частных ситуациях, появится универсальный алгоритм компиляции для данного класса архитектур.

В результате всего вышесказанного, или просто так сложились обстоятельства, но на данный момент VLIW процессоры используются преимущественно в нишевых продуктах, в основном в обработке сигналов. Из популярных процессоров этого класса можно в дополнение к TMS320C6x назвать

Analog Devices SHARC ADSP-21xxx (1994 г.)

Analog Devices TigerSHARC ADSP-TS201, снят с производства, есть аналог от компании Milandr.

Упоминавшиеся Эльбрусы от MCST.

Иногда VLIW используют как сопроцессор, совмещая с ядром общего назначения, например DaVinci от TI.

Аналогично, NeuroMatrix от Module.

…

EPIC

Стоит сказать несколько слов об этой вымершей ныне ветке процессоров от Intel (Itanium (1997…2001), Itanium II (2002)). Само название означает Explicitly Parallel Instruction Computing. Эту архитектуру нельзя назвать VLIW т.к. макро-инструкция содержит всего 3 микрокоманды. Её следовало бы отнести к классу LIW, но, с точки зрения Intel, акроним LIW имел ненужные коннотации, так что название дали в соответствии с присущей компании скромностью.

В архитектуру заложены следующие идеи:

Много регистров – 128 64-разрядных целочисленных и столько же с плавающей точкой

Механизм передачи параметров функций через “регистровые окна” с автоматической подкачкой из памяти (RSE).

Явный параллелизм в машинном коде (пачками (bundle) по 3 штуки).

In-order код, всю оптимизацию делает компилятор.

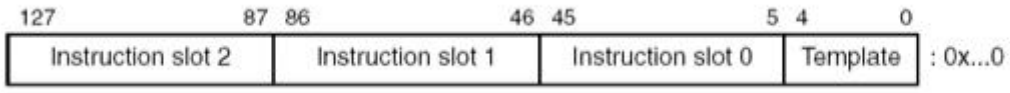
Масштабируемость по числу функциональных блоков (inherently scalable instruction set).

Предикативность (predication) в духе TMS320C6x – инструкции содержат ссылку на регистр-предикат, если в нём 0, инструкция пропускается

64 одноразрядных предикатов (регистров) вместо общего регистра флагов, который порождает ложные зависимости в данных.

Спекулятивная загрузка данных.

В EPIC интересует заявленная масштабируемость – слабое место архитектур с явным параллелизмом.



Макрокоманда (bundle) обычно состоит из трёх 41-разрядных инструкций. Иногда это одна 41 и одна 82 разрядная (с плавающей точкой двойной точности) инструкции. Кроме них в макрокоманду входит и 5-разрядный код шаблона (template).

Инструкции бывают разного типа в зависимости от задействованных функциональных устройств. Это:

M-unit – работа с памятью

I-unit – целочисленные операции с ALU (и без), работа с непосредственными операндами (immediate)

F-unit – работа с плавающей точкой

B-unit – операции ветвления

Типы инструкций

Мнемоника

Описание

Устройства

A

Целочисленные с ALU

I-unit или M-unit

I

Целочисленные без ALU

I-unit

M

Память

M-unit

F

Плавающая точка

F-unit

B

Ветвление

B-unit

X (L+X)

Расширение

I-unit и M-unit

Поскольку код – сущность одномерная, инструкции можно записать подряд, упоминая только их тип (A & I обычно смешивают в один класс – I), например:

… MMIMFBMIIMMF…

Как мы помним, в TMS320C6x каждая микро-инструкция содержала p-бит, который сообщал декодеру, может ли эта инструкция выполняться параллельно с инструкцией справа от нее. В EPIC применена похожая техника, код разделён на последовательности инструкций, которые могут исполняться параллельно, мнемонически они разделяются запятой, например:

… M,MIMFBMI,IMMF …

Здесь три группы параллельных инструкций. Инструкции объединены группами по три штуки, поэтому пример превращается в

… M,MI MFB MI,I MMF …

Вот эти трёхбуквенные комбинации и кодируются в поле template макро-инструкции. Однако, код шаблона это всего 5 разрядов, которые позволяют иметь до 32 вариантов, а имеющихся комбинаций типов намного больше (плюс запятые). Поэтому эмпирически, на основе статистики компиляций был подобран оптимальный набор шаблонов (всего 24), который кодируется следующим образом:

0 MII	1 MII,	2 MI,I	3 MI,I,
4 MX	5 MX,	6 –	7 –
8 MMI	9 MMI,	a M,MI	b M,MI,
c MFI	d MFI,	e MMF	f MMF,
10 MIB	11 MIB,	12 MBB	13 MBB,
14 –	15 –	16 BBB	17 BBB,
18 MMB	19 MMB,	1a –	1b –
1c MFB	1d MFB,	1e –	1f –

Что это даёт? Процессор (Itanium 2) состоит из т.н. исполнительных групп, каждая из которых содержит

6 ALU

2 целочисленных блока

4 мультимедиа блока

Двухпортовый L1 кэш данных

2 блока плавающей точки

3 блока ветвления

Число исполнительных групп зависит от модели процессора и компилятору оно неизвестно, считается, что она одна. Тем не менее, если таких групп несколько, при декодировании они могут быть задействованы. Например, пусть у нас последовательность параллельных инструкций, в которой 4 работают с плавающей точкой. В одной исполнительной группе 2 устройства с плавающей точкой, но декодер знает, что фактически исполнительных групп две, поэтому имеется возможность загрузить все 4 устройства FPU. Так и достигается заявленная масштабируемость процессора. Хотя, если бы компилятор сразу исходил из наличия двух групп, код скорее всего получился бы более оптимальным.

Итак, архитектура EPIC действительно масштабируется по числу функциональных устройств (но не регистров!). Т.е. Intel удалось победить одну из основных бед VLIW процессоров (впрочем, строго говоря, EPIC это не VLIW). Но статическая оптимизация и

трудности с эффективной компиляцией никуда не делись. В конце концов, при прочих равных, процессоры архитектуры EPIC оказались дороже оных из суперскалярной ветки AMD64, что и привело их к нынешнему состоянию.

Альтернативные идеи

Не секрет, что основные архитектурные идеи современных процессоров заложены в прошлом тысячелетии. Последние годы повышение производительности в основном шло за счет развития вширь – векторные вычисления, многопоточность, многопроцессорность, виртуализация … Если рассматривать архитектуру одного ядра/процессора, видно лишь постепенное устранение узких мест:

Тормозит интерфейс памяти – добавим кэша данных и кода, больше кэша микрокоманд (μops), повышаем частоту, шире интерфейсы,

Много сил уходит на декодирование – давайте кэшировать уже декодированные куски кода

Эволюция векторных операций:

SSE – SSE2 – SSE3 – SSSE3 – SSE4 – AVX – AVX2 – AVX512 - … ?

… ?

Поглядим по сторонам, наверняка стоящие идеи как минимум озвучены,

Достаточно смотреть и видеть, слушать и слышать.

EDGE

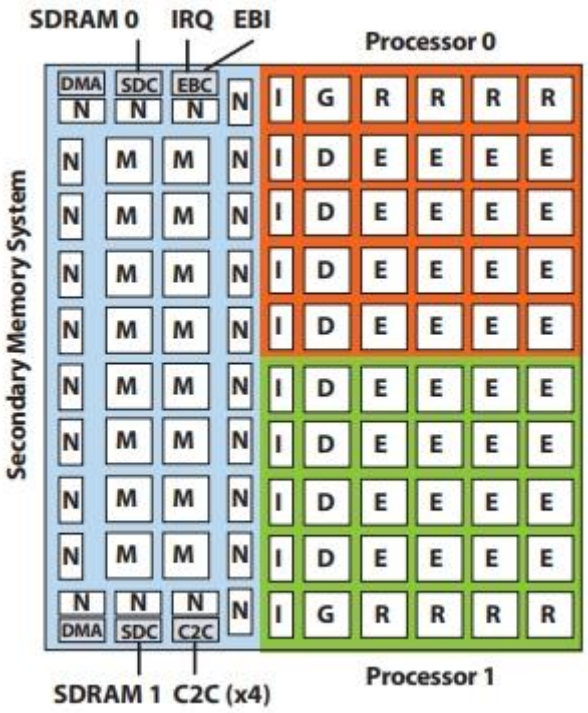
Этот акроним расшифровывается как Explicit Data Graph Execution. Под графом понимается именно граф данных. Концепция тяготеет к т.н. dataflow подходу, который альтернативен т.н. control flow подходу (Фон-Неймановские архитектуры).

EDGE архитектуры опираются на понятие базового блока из теории компиляторов. В традиционных архитектурах базовые блоки теряются при генерации кода, далее во время декодирования процессору приходится восстанавливать структуру заново (при предсказании переходов). В данном случае предлагается передавать процессору полную информацию о структуре программы. При этом ставка сделана не столько на спекулятивное исполнение разных веток, сколько на массовый параллелизм – предполагается одновременно исполнять десятки- сотни разных базовых блоков.

EDGE процессор состоит из множества микро-ядер, способных исполнять линейный (без ветвления) код, фиксировать или отбрасывать результаты. Микроядро содержит свой

собственный набор регистров, линейные участки кода обмениваются вычисленными результатами через специальные механизмы.

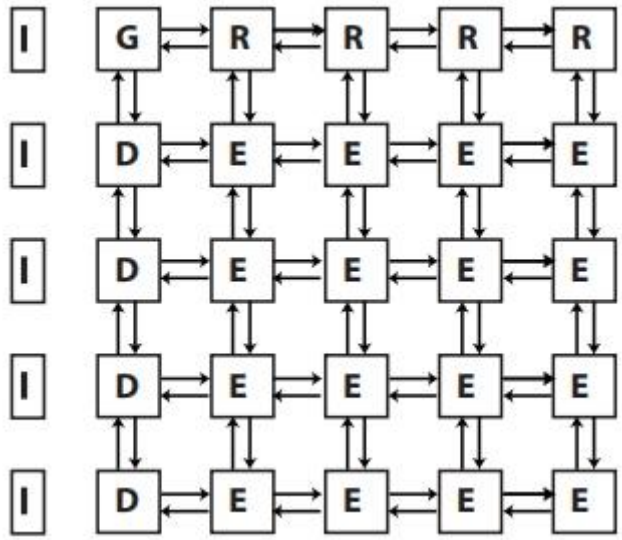
Рассмотрим на примере самой известной из архитектур этого класса TRIPS, во всяком случае она существовала “в железе”.



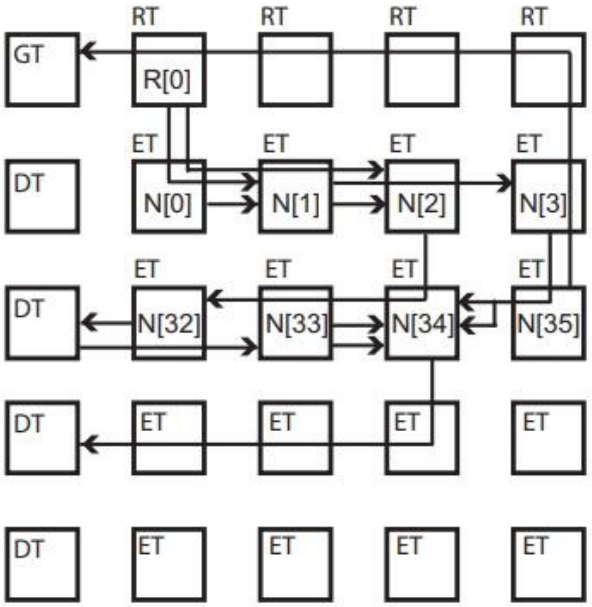
Прототип содержит на одном кристалле два процессора и интерфейсную часть (синяя), включая 1 Мб L2 кэш памяти в 16 М блоках.

Каждый из процессоров состоит из 16 микроядер (E, исполняют линейные участки кода, в каждом одно целочисленное устройство и одно с плавающей точкой.), пять блоков инструкций (I – 16 Кб I-cache в каждом), 4 блока данных (D, 8 Кб L1 D-cache в каждом), 4 блока регистров (R – 128 регистров в каждом), блок управления (G – global control tile).

Блоки связаны в плоские сетевые структуры (сети), например, сеть операндов выглядит так:



Для того, чтобы передать аргумент из левого нижнего E блока в правый верхний понадобится 6 тактов.




```
R[0]        read R4    N[1,L] N[2,L]
N[0]        movi #0    N[1,R]
N[1]        teq        N[2,P] N[3,P]
N[2]  p_f   muli #4    N[32,L]
N[3]  p_t   null       N[34,L] N[34,R]
N[32]       lw #8      N[33, L]   LSID=0
N[33]       mov        N[34,L] N[34,R]
N[34]       sw #0                 LSID=1
N[35]       callo      $func1
```

Инструкция TRIPS включает идентификатор блока E, на котором она будет исполняться, например, R[0]. А также куда отправлять полученное значение, например, N[1,L] означает что получателем является левый операнд блока N[1]. Теги p_t и p_f – означают разные ветки предикативного исполнения.

Какой конкретно блок будет исполнять инструкцию решает компилятор. Одновременно один процессор может исполнять до 4 потоков (threads).

Всё это удалось разместить на кристалле, содержащем 170 млн транзисторов.

Сравнение с Alpha 21264 (15 млн транзисторов) показало, что некоторые задачи TRIPS выполняет быстрее, некоторые медленнее, и то и другое иногда в разы. Код TRIPS, оптимизированный вручную, в 2…4 раза быстрее того, что выдаёт компилятор с максимальной оптимизацией, что неудивительно, ведь всё трудности VLIW компиляторов в данном случае можно смело возвести в квадрат. Т.е. прототип с 32 целочисленными (+ 32 floating point) блоками (против 2 + 2 у альфы) на некоторых задачах выглядит сопоставимо. Хотя и содержит в 11 раз больше транзисторов.

История с EDGE/TRIPS выглядит странно вот еще в каком отношении. В качестве одной из основных идей был заявлен отход от Фон-Неймановской архитектуры, но не предложен соответствующий теоретический подход/язык программирования. Вместо этого силы брошены на адаптацию компилятора C, вполне императивного и Фон-Неймановского. Неудивительно, что поставленная цель – создание (к 2012 году) петафлопсного процессора так и не была достигнута.

STRANDs

Сама по себе идея разбить код на зависимые по данным линейные участки (стренды) и исполнять их параллельно не нова. Она использовалась (насколько можно понять из отрывочных данных) в системе Эльбрус-3-1 (МКП) (1990 г.), не исключено, что и раньше были попытки.

Использование стрендов является альтернативой внеочередному исполнению инструкций (OoO) – в самом деле, хотя каждый стренд исполняется строго в своём порядке (in-order), они конкурируют между собой за ресурсы процессора и общий порядок заранее неизвестен.

Стренды выражают параллелизм намного более явно по сравнению с обычным суперскалярным процессором и намного более естественно по сравнению с VLIW. Причем, силы на это тратит компилятор. Стренды упрощают работу декодера и расширяют его “область видимости”. Однако же у них есть и весьма проблемное место – конкуренция за ресурсы процессора может приводить к взаимным блокировкам. Любая построенная на стрендах архитектура обязана предусмотреть средства недопущения дедлоков.

Рассмотрим относительно современный взгляд на проблему на примере выступления [10] Б.А.Бабаяна на открытом семинаре ARCCN (2016 г.), где он представил перспективную архитектуру Narch(+) от Intel.

Narch+

Эта архитектура, судя по всему, идеологически появилась из экспериментального проекта Narch (1992…1998)[12], разрабатывавшегося в МЦСТ под руководством Б.А.Бабаяна. Narch был попыткой сделать внешне SPARC-совместимый вариант Эльбруса-3 (не МКП). Рабочим языком программирования планировался Эль-96. Машина

работала на эмуляторе, в “железе” не существовала. Одна из основных особенностей – 4-канальное широкое командное слово.

Narch+ сильно отличается от своего предшественника. Это по-прежнему система с явным параллелизмом, но организованным не через широкое командное слово, другими методами. Внешняя совместимость сделана с x86.

Есть и общие черты, например, объектная модель на основе тегированной памяти (capability mechanism), но здесь разбирается только то, что касается (пост?)суперскалярности.

Аналоги стрендов здесь называются потоками (streams).

Потоки исполняются т.н. worker-ами, каждый из которых имеет свой счетчик команд (IP).

Инструкции загружаются “как есть”, без декодирования.

Таких worker-ов может быть порядка 16 на кластер. Кластер содержит общий для всех worker-ов пул регистров (~500 штук), исполнительные устройства, станции резервирования…

Кластеры образуют процессор, в котором их может быть ~60.

Потоки порождаются в двух случаях – либо это начало базового блока, либо внутри базового блока компилятор (когда видит независимый по данным от текущего набор инструкций) вставляет специальную инструкцию FORK. Т.е. поток всегда привязан к какому-то базовому блоку.

Алгоритм разбиения базового блока на потоки таков:

Компилятор приписывает каждой инструкции базового блока номер PVN (Priority Value Number) – число тактов до конца базового блока при фиксированной стоимости инструкций.

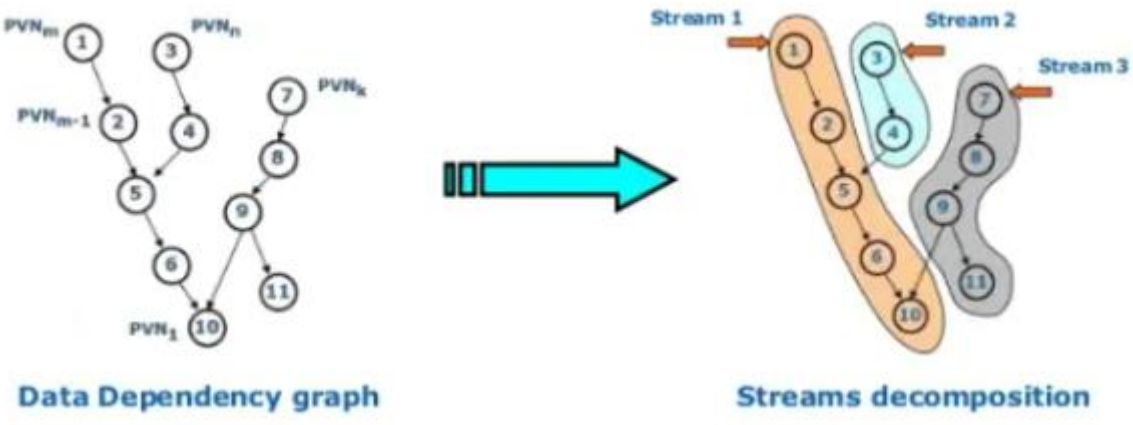
Компилятор сортирует инструкции по PVN.

Создаёт стартовый поток.

Проходит инструкции в порядке убывания, при этом

Если текущая нераспределённая инструкция зависит по данным от распределённой, дописывает её в тот же поток

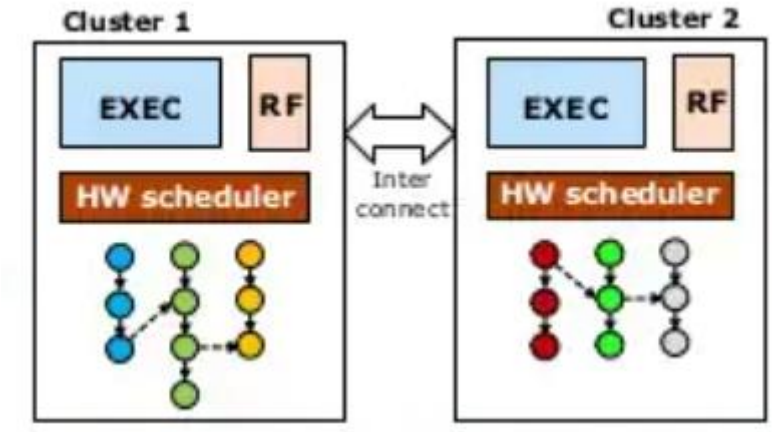
Если зависимостей нет, создаёт новый поток



- Компилятор следит за тем, чтобы одновременно работающих потоков было не слишком много
-
- Если базовый блок велик, компилятор разбивает его на кадры (intervals).
-
- Делается это с помощью механизма CL/DL.
-
- DL (Definition Line) и CL (Control Line)- линии, разделяющие инструкции базового блока.
-
- DL(i), i-номер кадра, соответствует некоторому моменту времени, когда, по мнению компилятора, все активные потоки переходят в новый кадр. После того, как все потоки пересекут DL(i), из буферов инструкций (в потоках) можно удалить всё, касающееся предыдущего кадра.
-
- CL(i) – аналогичен DL(i), это барьер, достигнув которого, все потоки останавливаются и ожидают до тех пор, пока соответствующий DL не будет пересечен всеми, кому это предназначено компилятором. Или просто продолжают работу, если это событие уже наступило. Кроме того, достижение потоком линии CL(i) – отличный сигнал, что пора загружать инструкции следующего кадра. Данный механизм синхронизирует потоки, не позволяет им сильно отставать или забегать вперёд.
-
- DL и CL – физическии это метки на инструкциях
-
- Раз в кластере около 500 регистров, размер кадра (в инструкциях) делается равным числу регистров. Даже если каждая инструкция будет выдавать результат в свой регистр, их хватит на всех и взаимная блокировка в таких условиях

невозможна. В этом в общем то и смысл кадров – компилятор разграничивает инструкции так, чтобы тупики были невозможны теоретически.

-
- Предполагается, что в обычных условиях CL/DL позволяет прозрачно и без задержек выполнять базовые блоки любого размера без ограничения по ресурсам и риска попасть в дедлок.
-
- Допускается спекулятивное исполнение базовых блоков, для этого необходим механизм фиксации/устаревания инструкций (аналог ROB), причем общий для всех потоков.
-
- На случай небольших веток присутствует предикативность (predication) в духе TMS320C6x / Itanium
-
- Допускается использование одной функцией до четырёх кластеров, либо для работы с разными ветками при ветвлении, либо с итерациями цикла, если это позволяют зависимости по данным.
-
- Особым случаем потоков являются прологи и эпилоги функций при их вызове. Что интересно, и вызывающей и вызываемой сторонам есть чем заняться при вызове функции и возврате из нее, кроме того, обе стороны могут делать это в разных потоках, не мешая друг другу.
-
- При определённых условиях, например, независимых по данным итерациям циклов, функция может занять больше кластеров, чем четыре, вплоть до того, чтобы использовать их все.
-
- Совместное исполнение функции на нескольких кластерах подразумевает распределённое хранение регистров по кластерам. А также механизм поддержки когерентности, аналогичный оному для кэш-памяти (MESI, …).



Логично предположить, что поддержка когерентности распространяется не на все ~500 регистров, а только лишь на их часть (с индексами 0…N), тогда как остальные регистры остаются локальными для данного кластера. Технически ограничение N задаётся либо архитектурой, либо определяется компилятором для каждой функции.

Локальные регистры (предположительно) не сохраняются при вызове функций, вся конструкция в этом случае становится похожей на распределённый RSE (Register Stack Engine) Итаниума.

Что же, архитектура Narch(+) умудряется(лась, на данный момент следов развития проекта нет) оставаться регистровой машиной со всеми плюсами эффективной компиляции, разделяя при этом концепцию EDGE с её естественной OoO, но без ужасов dataflow. Во всяком случае, так видится автору.

А что если

Из вышеописанного автор выделяет две перспективные идеи. Если внимательный и дотошный читатель видит что-то еще, пусть не постесняется написать об этом.

Разметка* кода

Слово “разметка” здесь используется в том же смысле в каком HTML – язык разметки текста.

Что же является узким местом суперскалярных процессоров? Вот Intel жалуется в статье 10-летней давности:

В чем состоит проблема? В сложности задачи, которую берет на себя процессор. Суперскаляр на ходу распараллеливает последовательный код. Но этот процесс распараллеливания слишком трудоемок даже для нынешних процессорных мощностей, и именно он ограничивает производительность машины. Делать это преобразование быстрее определенного количества команд за такт невозможно. Можно сделать больше, но при этом упадет тактовая частота – такой подход, очевидно, бессмысленен. Разумный предел количества команд в настоящее время изучен со всех сторон и пересмотру не подлежит.

Что ограничивает скорость декодирования?

Отчасти, необходимость предсказывать переходы, чтобы избежать простоев при передаче управления в неподготовленный участок кода. Для этого необходимо забежать вперёд как можно дальше и протестировать адрес каждой инструкции, что, например, в случае x86 с её системой команд, весьма нетривиально без декодирования.

Даже для архитектур с более простой упаковкой инструкций сильно далеко забежать не удастся, ведь размер ROB (буфер упорядочения) ограничен (обычно единицами сотен элементов, ex: Sandy Bridge – 168). Кроме того, ROB может содержать инструкции нескольких веток кода.

Размер ROB в свою очередь ограничен количеством внутренних регистров. Внутренние регистры – довольно дорогой ресурс, что неудивительно, учитывая многопортовый доступ к ним (P4 Prescott: 256 штук (32-разрядных или 128 64-разрядных), 12 портов на чтение и 6 на запись). И увеличивать число регистров бессмысленно без расширения параллельного доступа к ним, а это сильно усложняет всю конструкцию.

Что здесь можно предпринять?

Мы привыкли воспринимать код как в основном одномерный поток инструкций. В таком потоке инструкции управления (control flow) перемежаются инструкциями обработки данных (data-flow). Вызов функции это обобщенная инструкция – безусловная передача управления с обязательным возвратом обратно (+- exception).

Разделение единого потока инструкций на два – control-flow & data-flow выглядит довольно многообещающим:

Набор инструкций разделяется на два подмножества

Инструкции ветвления, безусловного перехода, исполнения линейного участка, вызова и возврата из функции …

Работа с памятью и регистрами

Компилятор для каждой функции создаёт единый поток управления и по потоку обработки на каждый линейный участок кода

Соответственно, есть два типа декодеров – каждый на своё подмножество системы команд. Из общих соображений, они проще единого и, к тому же, могут работать параллельно.

Data-flow декодеров может быть несколько, так два декодера в состоянии параллельно обрабатывать обе ветки конструкции IF-THEN-ELSE

Рассмотрим на примере первой попавшейся под руку процедуры.

```
Typedef struct bitKey_s {
   …
   Uint64_t vals_[4];
```

```
} bitKey_t;
Const uint64_t smasks8[8] = {…};

Void setLowBits(bitKey_t pk, int idx) {
   For (int lidx = idx; lidx >= 0; lidx -= 64) {
      Unsigned ix = lidx >> 6;
      If (idx == lidx) {
         Unsigned llidx = (lidx % 64);
         Pk->vals_[ix] |= (smasks8[7] >> (63 – llidx));
         Pk->vals_[ix] -= (1ULL << llidx);
      } else
         Pk->vals_[ix] |= smasks8[lidx & 7];
   }
}
```

Эта функция (setLowBits) скомпилирована MSVC и получившийся ассемблерный код (x86-64) разобран на части так, как с точки зрения автора это следовало бы сделать компилятору. X86 здесь только для примера.

Инструкции control flow

```
      __run df_setLowBits_1
      Js    setLowBits_4
; for (…) {
      __run df_setLowBits_2
setLowBits_1:
      __run df_setLowBits_3
; if (idx == lidx)
      Jne   setLowBits_2
      __run df_setLowBits_4
      Jmp   setLowBits_3
; else
setLowBits_2:
      __run df_setLowBits_5
setLowBits_3:
```

```
        __run df_setLowBits_6
; }
        Jns   setLowBits_1
        __run df_setLowBits_7
setLowBits_4:
        __run df_setLowBits_8
        Ret
```

Инструкции data flow

```
Df_setLowBits_1:
    Push        rbx
    Movsxd      r11,edx
    Mov         rbx,rcx
    Mov         eax,r11d
    Test        edx,edx
    __stop

Df_setLowBits_2:
    Mov         qword ptr [rsp+10h],rbp
    Mov         r10,r11
    Mov         qword ptr [rsp+18h],rsi
    Mov         ebp,1
    Mov         qword ptr [rsp+20h],rdi
    Mov         rsi,8080808080808080h
    Lea         rdi,[smasks8]
    __stop

Df_setLowBits_3:
    Mov         r8d,eax
    Sar         r8d,6
    Cmp         r11d,eax
    __stop

Df_setLowBits_4:
```

```
    Lea         r9,[rbx+r88]
    Mov         ecx,3Fh
    Mov         edx,eax
    And         edx,3Fh
    Sub         ecx,edx
    Shrx        r8,rsi,rcx
    Or          r8,qword ptr [r9]
    Mov         ecx,edx
    Shlx        rdx,rbp,rcx
    Sub         r8,rdx
    Mov         qword ptr [r9],r8
    __stop

Df_setLowBits_5:
    Mov         rcx,r10
    Lea         rdx,[rbx+r88]
    And         ecx,7
    Mov         rcx,qword ptr [rdi+rcx*8]
    Or          qword ptr [rdx],rcx
    __stop

Df_setLowBits_6:
    Sub         r10,40h
    Sub         eax,40h
    __stop

Df_setLowBits_7:
    Mov         rdi,qword ptr [rsp+20h]
    Mov         rsi,qword ptr [rsp+18h]
    Mov         rbp,qword ptr [rsp+10h]
    __stop

Df_setLowBits_8:
    Pop         rbx
```

__stop

Добавлены две инструкции - __run в потоке управления запускает исполнение линейного участка, __stop в потоке исполнения останавливает этот участок, после чего поток управления может продолжить работу. С точки зрения потока управления, __run – это единая макро-инструкция.

На первый взгляд стало логичней чем было. Во всяком случае, предсказывать переходы в таком виде намного проще.

Регистровая машина без ограничений.

Еще одна цитата из статьи уважаемого @saul (Intel):

Главная проблема суперскаляра – в неприспособленности входного кода к его нуждам. Имеется параллельный алгоритм работы ядра и параллельно организованная аппаратная часть. Однако между ними, по середине, находится некая бюрократическая организация – последовательная система команд. Компилятор преобразует программу в последовательную систему команд; от той последовательности, в какой он это сделает, будет зависеть общая скорость процесса. Но компилятор не знает в точности, как работает машина. Поэтому, вообще говоря, работа компилятора сегодня – это шаманство.

Автору очень нравится эта статья, общей атмосферой безысходности она напоминает “Что-то страшное грядёт (Something wicked this way comes)” Рэя Брэдбери.

Регистровая машина без ограничений.

Еще одна цитата из статьи уважаемого @saul (Intel):

Главная проблема суперскаляра – в неприспособленности входного кода к его нуждам. Имеется параллельный алгоритм работы ядра и параллельно организованная аппаратная часть. Однако между ними, по середине, находится некая бюрократическая организация – последовательная система команд. Компилятор преобразует программу в последовательную систему команд; от той последовательности, в какой он это сделает, будет зависеть общая скорость процесса. Но компилятор не знает в точности, как работает машина. Поэтому, вообще говоря, работа компилятора сегодня – это шаманство.

Автору очень нравится эта статья, общей атмосферой безысходности она напоминает “Что-то страшное грядёт (Something wicked this way comes)” Рэя Брэдбери.

Итак, компилятор проталкивает алгоритм с его естественным параллелизмом через ограничения архитектуры. С другого конца суперскалярный процессор пытается восстановить задумки компилятора. Тупик сложности, связанный с этой задачей был

осознан довольно давно, в результате и пошли разговоры о конце Фон-Неймановской архитектуры.

Тут следует сделать уточнения:

Во первых, одномерность (последовательность) системы команд не является непременным атрибутом кода, создаваемого компилятором, а существует преимущественно в головах в виде привычной парадигмы.

Во вторых, именно в случае суперскалярного процессора основным ограничением архитектуры является использование регистров общего назначения (их количество и архитектурные обременения). Для преодоления этого ограничения компилятор использует алгоритм на основе раскраски графа. На стороне процессора обратную работу выполняет алгоритм переименования регистров (Томасуло) с ROB.

Всех этих сложностей можно было бы избежать, если бы у компилятора была возможность передать процессору без изменений весь найденный им параллелизм. Ну или, если бы компилятор знал и имел возможность использовать настоящее число регистров в процессоре.

Обе идеи в том или ином виде были реализованы.

Про настоящее число регистров.

AMD29K[13].

Первые процессоры этой линейки не были суперскалярными.

Это архитектура класса Berkeley RISC, т.е. с регистровыми окнами.

Пул локальных регистров – 128 штук, образующих кольцевой буфер

Общее количество локальных регистров, доступных функции (есть еще глобальные) переменно (определяется компилятором), вплоть до всех 128 штук, хотя память именно под параметры ограничена 16 регистрами. Это отличает архитектуру от, скажем, SPARC, где окна были фиксированного размера 8+8+8 = 24 штуки (входные параметры + внутренние переменные + параметры вызова следующей функции, которые после сдвига окна становились её входными параметрами).

При сдвиге окна регистров их часть (за пределами окна) может сбрасываться в память и\или загружаться из памяти, это делалось программно (компилятором) в прологах и эпилогах функций.

Существовал также и честный стек в памяти, т.е. компилятор решал, какие локальные данные поместить в быстрый регистровый стек, а какие оставить в обычном.

Теоретически, в данной архитектуре число доступных при необходимости регистров настолько велико, что процессор фактически не нуждается в переименовании регистров, их при распределении и так должно хватить на всё. Однако, платить за это пришлось бы избыточным обменом с памятью. Поэтому суперскалярные процессоры этой архитектуры использовали переименование регистров с ROB из 10 элементов.

Narch(+)Линейные участки кода исполняются “кадрами” по ~500 штук.Доступных регистров общего назначения тоже около 500.Т.е. без всякого переименования регистров заведомо хватит для хранения результатов каждой инструкции.

Про передачу параллелизма от компилятора процессору.

Некоторые примеры были описаны выше.

EDGE/TRIPS, см. Фиг. 24, 25. Процессор TRIPS содержит 16 вычислительных устройств, образующих сеть (4x4) с передачей операндов между узлами сети. При компиляции выражение “выкладывается” на эту вычислительную сеть с сохранением связей графа выражения. Регистры в TRIPS тоже есть, но они используются скорее как средство коммуникации.

EDGE/Multiclet. Линейные участки кода в этой архитектуре называются параграфами. Параграф состоит из команд, операндами которых являются ссылки на результат работы одной из вышестоящих команд. Параграф может быть любого размера, но расстояние между командой и источником её операнда не может превышать 64 команды. Обмен данными между параграфами изначально был только через память, позднее появился более быстрый механизм.

Напоминает Narch+, где, фактически, каждой инструкции соответствовал регистр для хранения её результата.

С другой точки зрения, это TRIPS с человеческим лицом.

Итак. Если бы мы собрались улучшить взаимопонимание компилятора и процессора, с чего следовало бы начать? С фиксации факта, что внутреннее представление программы

компилятором – трёхадресный код, фактически, код для процессора с неограниченным количеством регистров. Т.е. идеальная архитектура не должна содержать ограничений на число регистров.

Следствием является утверждение, что идеальная архитектура принадлежит классу архитектур с регистровыми окнами, ведь если нет ограничений на регистры, невозможно оптимально разделить их на volatile и non-volatile.(*на самом деле это дискуссионный вопрос).

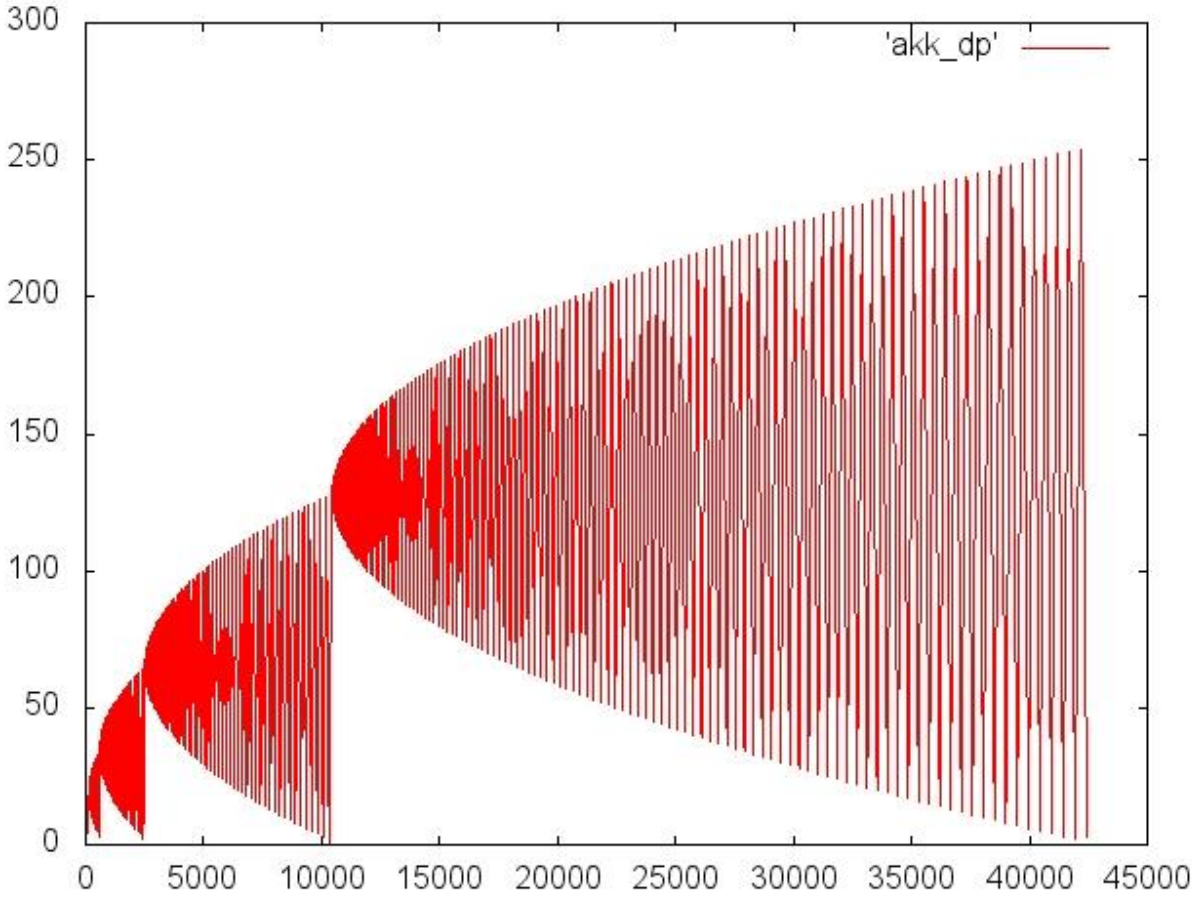
Но ведь чуть раньше говорилось, что большое количество регистров мешает вызову функций и чем больше регистров, тем больше затраты на сохранение их содержимого. И что 32 регистра – это тот самый баланс, где достигается оптимум производительности.

Что же, “в действительности всё не так, как на самом деле”(C). В первую очередь это (32 регистра как оптимум) касается архитектур без регистровых окон, тех, что произошли от ветки Stanford MIPS. Но вот для AMD29K (и для Itanium, кстати) компилятор мог самостоятельно назначить размер окна регистров для функции.

Регистровое окно состоит из трёх частей – входные параметры, регистры контекста выполнения и будущие аргументы функций, вызываемых из этой. В промежутках между вызовами регистры под будущие аргументы можно использовать как временные. И здесь получается та же дилемма, маленькое окно – неэффективный код, большое окно – большой обмен с памятью. И, несмотря на то, что компилятор выделил функции (к примеру) 20 регистров из 128 фактически имеющихся, остальные использовать нельзя даже временно, например, в вычислительном цикле. Пул регистров это ведь кольцевой буфер, в них (регистрах за пределами окна) чьи-то данные.

Регистровые окна предназначены для ситуаций, аналогичных следующей: в длинной функции с кучей циклов и вычислений мы в итерации делаем вызовы функций с умеренным количеством параметров и умренным контекстом исполнения. В результате возможно пережить недлинную цепочку вызовов и вернуться назад, причем исходный контекст исполнения всё еще будет в регистрах. А значит есть шанс обойтись вообще без обращения к памяти! Обычные регистровые процессоры решают эту проблему с помощью кэш памяти.

Понятно, что если в задаче поток управления устроен в духе функции Аккермана (Фиг. 29), то не поможет ни кэш, ни регистровый пул любого размера, сохранение контекста функций в любом случае будет идти через память. К счастью, это экстремальный случай.



Так какую проблему хочется решить из тех, что недоступны для существующих архитектур (с регистровыми окнами или без оных) ? Их даже две.

Дать возможность функции в промежутках между вызовами подфункций использовать любое (в пределах разумного) количество регистров в том числе для временных данных.

Дать возможность функции во время вызова подфункций сохранять свой контекст в том (любом) количестве регистров (…), которого эту функцию в данный момент сочтёт достойной компилятор.

Тезисы:

Разумным считается количество программно доступных регистров, которое умещается на минимальной странице физической памяти (на двух?). Т.е. для 4K – 512 (1024?).

Количество физических регистров заметно меньше (пусть 64) и оно неизвестно компилятору. Этот параметр доступен для масштабирования.

Регистровый стек аналогично AMD29K & Itanium имеет своё отражение в память.

Есть также и обычный стек в памяти, где компилятор размещает большие или не требующие быстрого доступа данные.

Программно доступные регистры имеют своё отражение в кэш память L1. Для работы требуется полное кэширование трёх (пяти?) физических страниц – текущей, (двух?) предыдущей(их) и (двух?) следующей(их).

Физические регистры также отражаются в кэш L1, но значения в них доступны непосредственно, без загрузки.

Значения из физических регистров записываются в кэш при сдвиге окна если они при этом перестают в него попадать.

Доступ к физическим регистрам быстрее, чем к прочим..

При сдвиге окна назад (возврат из функции), содержимое регистров, ставших физическими загружается в регистры либо сразу либо по требованию (lazy load)

Нумерация регистров в каждой функции идёт от начала окна (Berkeley RISC).

В Berkeley RISC сдвиг окна происходит так, чтобы output часть окна – аргументы вызываемой функции стали input частью окна после вызова. В случае, когда output-секция полностью или частично располагалась в физических регистрах, она так в них и останется, загружать ничего не придётся. Сдвиг назад аналогичен. Вообще, когда размер окна меньше числа физических регистров, возникает выбор, как расположить его. Два крайних варианта – избыточные регистры остаются ниже окна (предположим что стек растёт вверх) и, возможно, это облегчит возврат из функции. Либо они размещены выше окна и это ускорит следующий вызов. Решать компилятору.

Кодирование номеров регистров в инструкциях производится, например, с помощью целых переменного размера, в большинстве случаев размер инструкций даже не вырастет.

Решает ли такая схема сформулированные выше проблемы?

Функция может использовать любое количество регистров (в пределах разумного) до тех пор, пока не потребуется вызвать другую функцию. Окно регистров ограничено снизу, но сверху предела нет (…). Выше окна расположена незанятая пока память, её нельзя испортить. Те регистры, которые представлены физически, имеют преимущество по скорости перед остальными, но их не так уж и мало. Если же физических регистров не хватило, значит компилятор решил что это всё равно дешевле, чем обращаться к памяти напрямую.

При вызове функции компилятор должен решить, что именно он считает необходимым сохранит в регистрах для вызывающей стороны. Если сохранять слишком мало, то, вероятно, часть данных придётся записать в память (основной стек). Или по возвращении вычислить заново. Если оставить слишком много, неизбежны издержки, связанные с загрузкой данных из кэша в физические регистры по возвращении. Как бы то ни было, для следующего каждого вызова (внутри одной вызывающей функции) компилятор может определить оптимальный размер окна для сохранения временных данных именно здесь.

Резонный вопрос, а чем это отличается от обычной работы регистрового процессора, там ведь так же при вызове функции данные сохраняются в память, фактически в кэш ?Разница в отсутствии избыточных инструкций для загрузки/выгрузки данных, код более компактный, не нужно декодировать лишнего, работает не медленнее.

А чем это лучше существующих архитектур с регистровыми окнами ? Гибкостью. Вместо создания пула регистров на всё время вызова функции, компилятор свободно использует оптимальное количество регистров в каждый момент времени, сохраняя при дочерних вызовах только действительно необходимое.