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

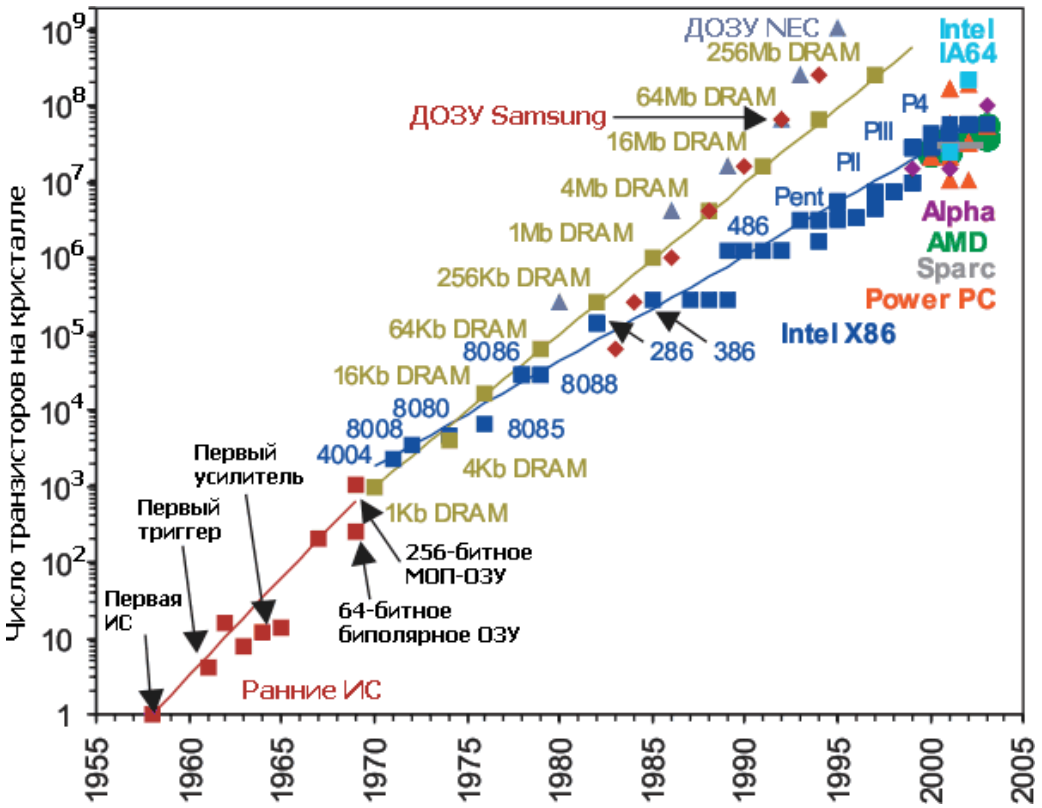
RISC

- 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 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.



Фиг.1 Зависимость числа транзисторов микропроцессора от времени (отсюда)

На что можно было потратить эти транзисторы? В первую очередь на кэш память. А также на расширение внутренних интерфейсов, общую оптимизацию … Критерием оптимизации может быть не только уменьшение количества использованных вентилей, но и производительность системы. Например, многопортовые регистры (позволяющие одновременно читать регистр разными инструкциями и писать в него) сложнее обычных однопортовых, но заметно увеличивают производительность.

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, содержимое регистров требуется сохранить. Подробнее об этом в главе (2.4.) про регистровые окна.

Когда регистров мало, то и проблемы никакой нет. Но чем больше регистров, тем больше усилий требуется чтобы сохранить их состояние.
32 регистра общего назначения - консенсус, компромиссное значением для соблюдения баланса между производительностью вычисления выражений и лёгкостью

вызова функций.

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 68К и 88К.

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.

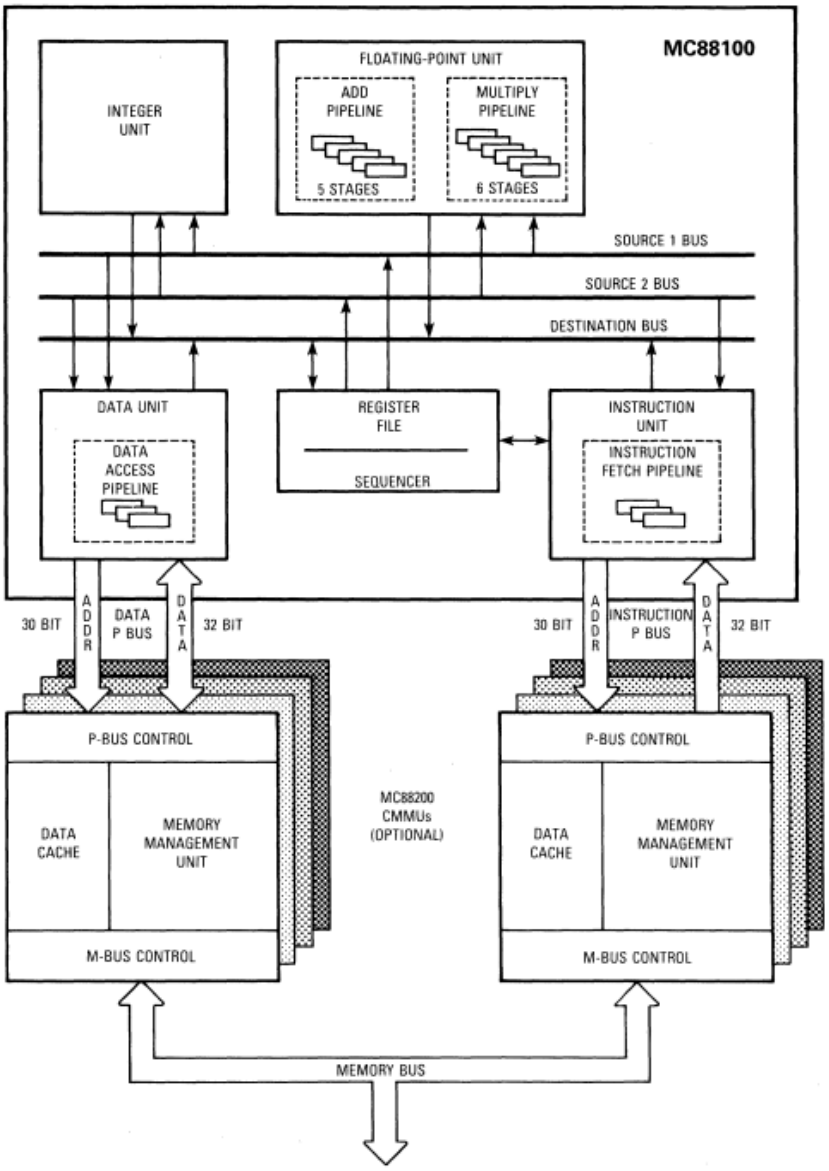
И, наконец, архитектурные изыски, **использование параллелизма**.
Практически любая программа обладает некоторым параллелизмом. Два выражения, которые не зависят друг от друга по данным, могут быть вычислены параллельно.
Мы не будем рассматривать программный параллелизм, равно как и использование векторных 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:

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.)

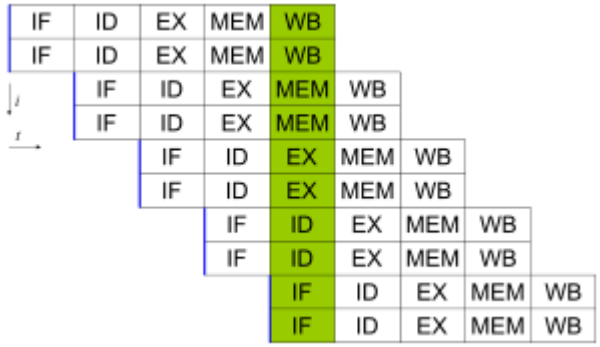


Фиг.2 Функциональная схема 88100 (отсюда)

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, 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.



Фиг.3 Два независимых конвейера. (отсюда)

Типичный пример - Pentium (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 от регистров и функциональных устройств (scoreboard)

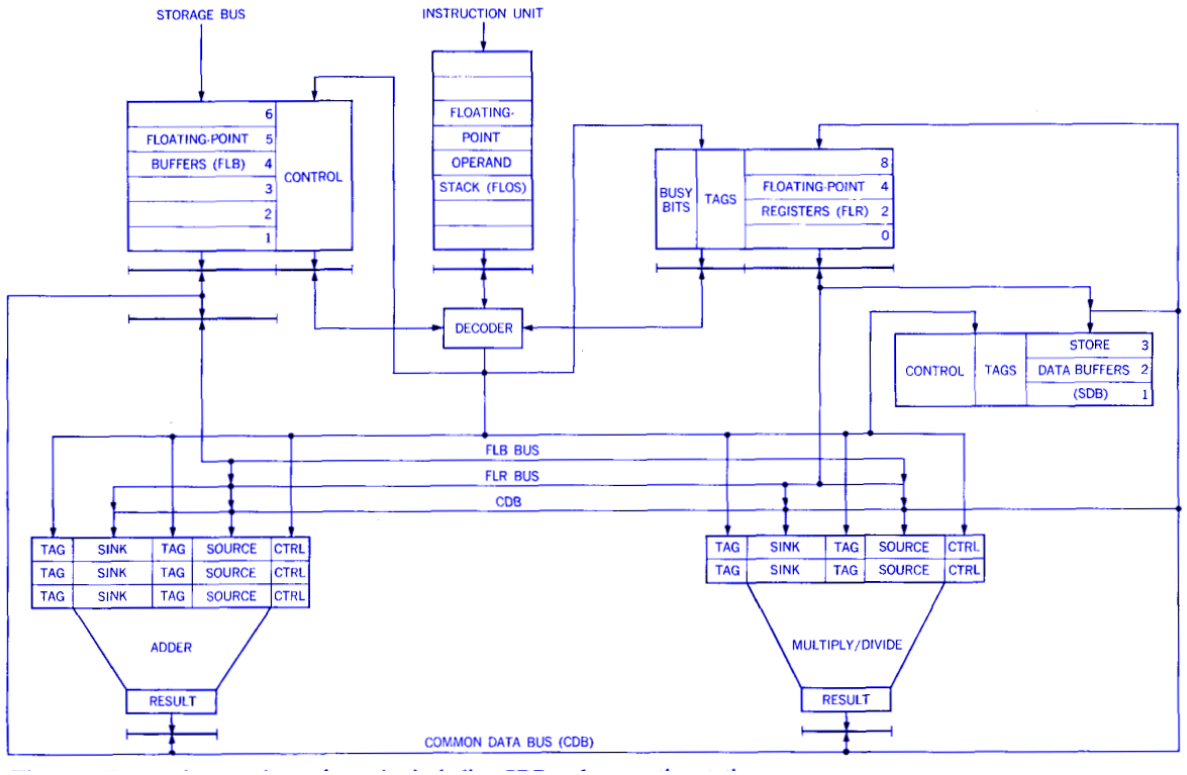
2) инструкция ожидает, когда освободятся все её зависимости
3) после исполнения, scoreboard уведомляется, что ресурсы освободились

Переименование регистров (register renaming).

Альтернативная методика от IBM. Разберём подробно, это важно.

В архитектуре IBM 360 (1964 г.) всего 4 регистра с плавающей точкой двойной точности и это сильно ограничивало генерацию эффективного кода компилятором. Серия разрабатывалась для применения преимущественно в коммерческих целях, но быстро выяснилось, что клиенты применяют их и в инженерных расчетах, где эти 4 регистра стали узким местом. Можно добавить умножителей/делителей, но они будут простаивать, нужный объем данных через 4 регистра просто не протолкнуть. Ведь код должен был быть совместимым с другими моделями и новые регистры добавить в архитектуру на лету невозможно.

В 1965 г. Роберт Томасуло (Robert Tomasulo) предложил алгоритм/архитектуру блока для работы с плавающей точкой (Фиг.4), названную в его честь. Этот алгоритм (его еще называют техникой переименования регистров, когда фактически используется большее количество регистров, чем видно снаружи) устраняет узкое место и позволяет прозрачно масштабировать архитектуру - на дешевых машинах мало устройств и низкая производительность, на дорогих машинах - наоборот.



Фиг.4 Работа с плавающей точкой в IBM 360/91 (отсюда)

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 потребуется значение этого регистра, она не обязана дожидаться пока значение доедет до регистра, а сможет перехватить значение непосредственно на выходе из функционального устройства.

Примеры:

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.

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.

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)           ; 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.
- Позволяет масштабировать архитектуру, увеличивая число функциональных устройств. Например, мы видели, что 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. Например, 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 - просто изменив регистр - счетчик команд.

Не менее горячо наше 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-а), 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.

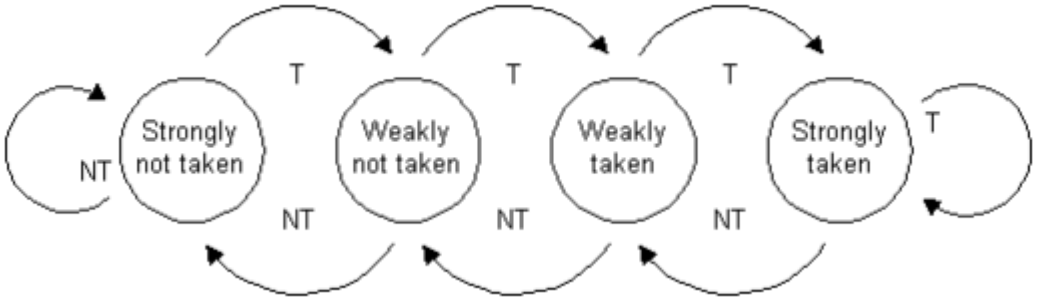
Косвенно, о том что мы имеем дело с циклом, можно судить по переходу по адресу назад от текущего.

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 перехода 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 -1.5 >= SC <= 1.5 (четыре состояния: -1.5 ... -0.5 ... 0.5 ... 1.5).
Значение меняется с шагом 1, если фактический переход был taken, то единица прибавляется (в пределах интервала), если not taken, единица отнимается.



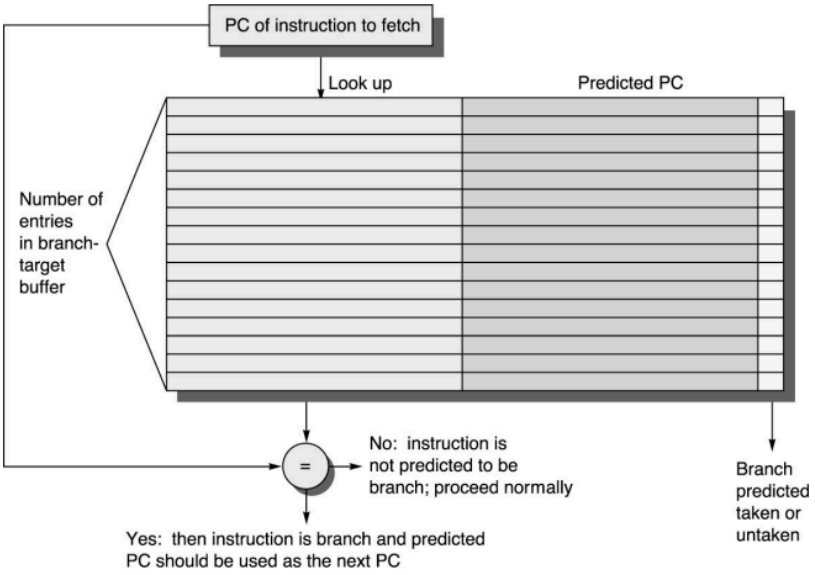
Фиг.5 Конечный автомат счетчика с насыщением, отсюда

Из всех счетчиков с насыщением, как ни странно, лучше всего работают именно показанные на Фиг.5 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).

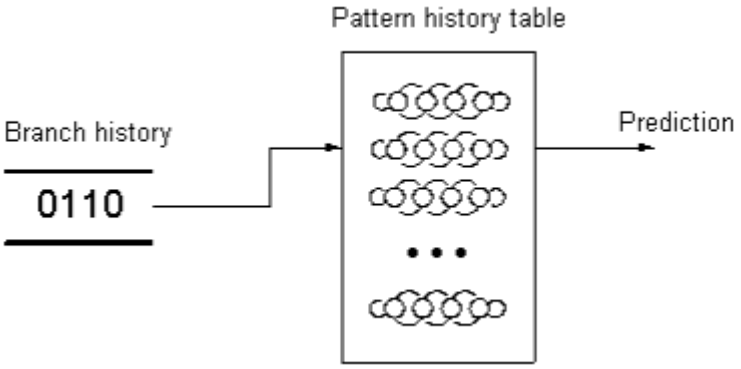
BTB использует значение хэш функции от адреса как ключ для хранения и поиска, сам адрес как тег для сравнения. Кроме тега и предсказанного адреса в элементе BTB содержится информация для предсказателя переходов. Иногда это просто те самые два разряда счетчика с насыщением, иногда что-то более сложное,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



Фиг.6 Буфер предсказаний переходов. (отсюда)

Рассмотрим на примере PentiumPro (1995 г.) [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



Фиг.7 двухуровневый предсказатель перехода PentiumPro [стр.13]

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

вместо

```
    ; флаги установлены ранее
    BNE over          ; переход, если Z != 0
    LSL r0, r0, #24   ; если Z = 0
    ADD r0, r0, #2
over
    ;...
```

получаем

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

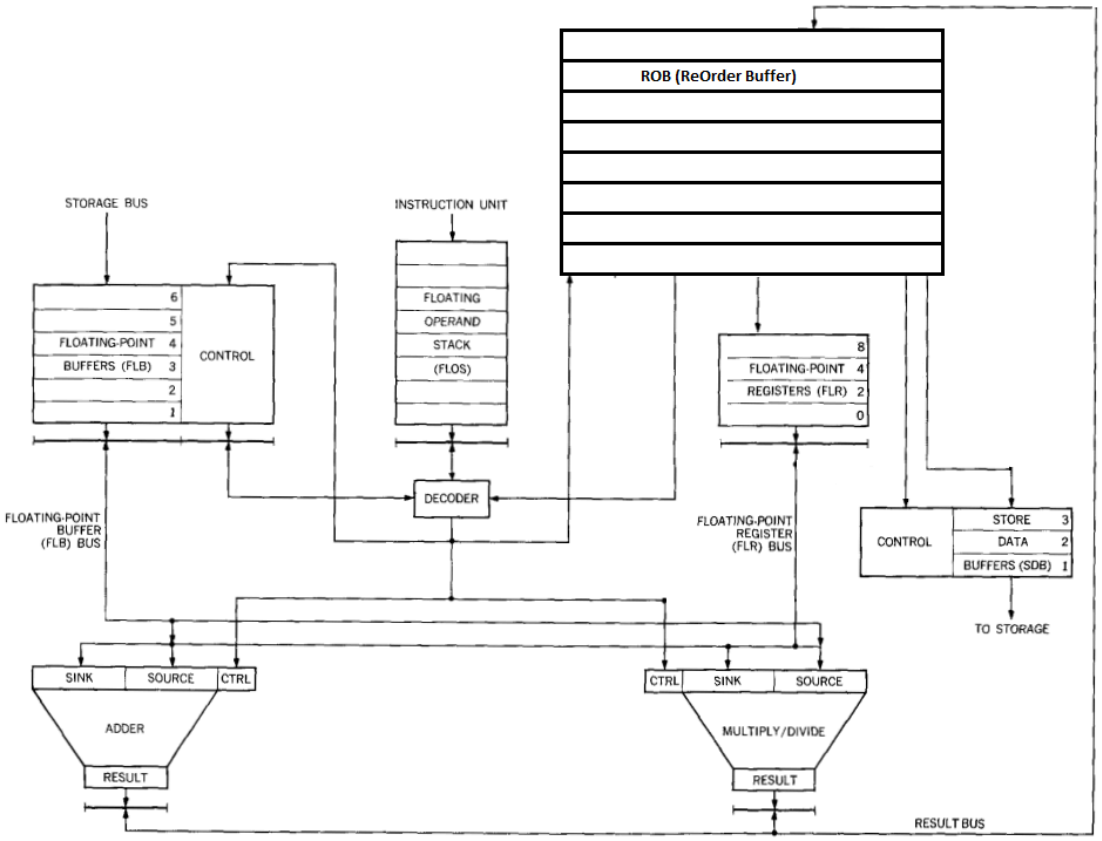
Спекулятивное (по предположению) исполнение

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 сможет 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 (FLR на Фиг.4) 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).



Фиг.8 Схема Томасуло с буфером переупорядочивания.

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 мы рассматривали 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.

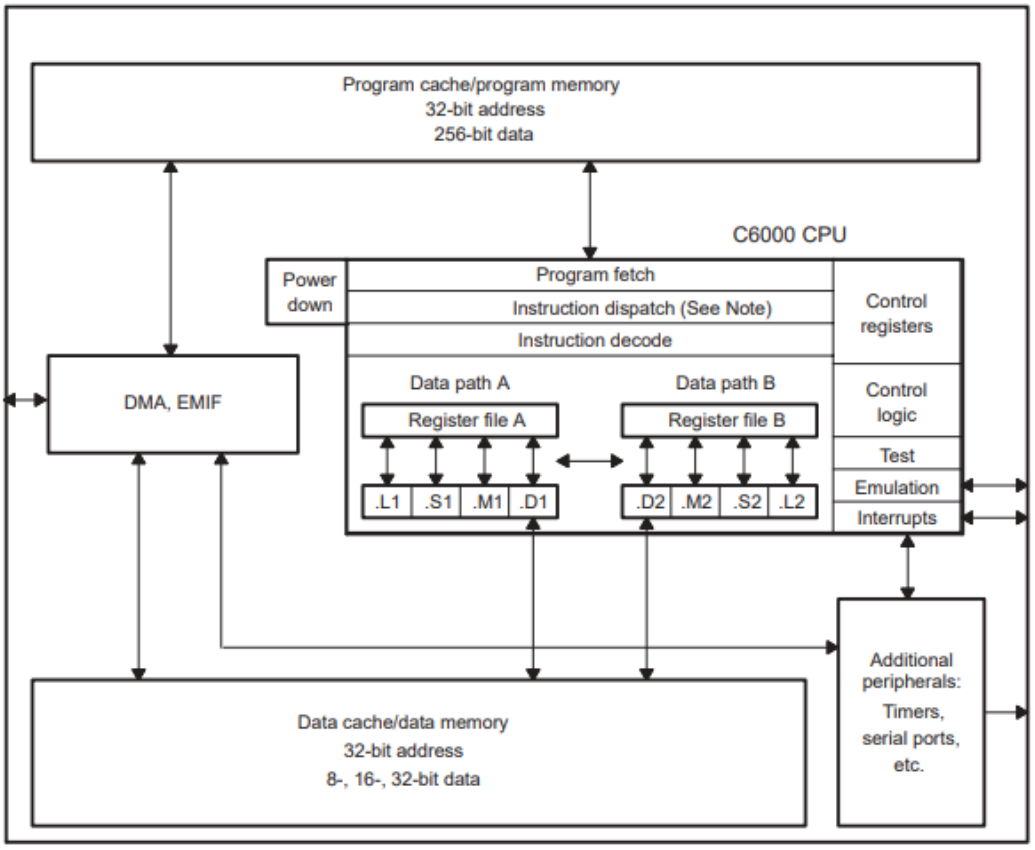
С третьей стороны, именно наличие этого водораздела позволяет, насколько это возможно, полно использовать функциональные ресурсы процессора. В результате мы имеем фантастическую совместимость сверху вниз, когда старая программа на допотопной системе команд без видимых усилий работает на новом процессоре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

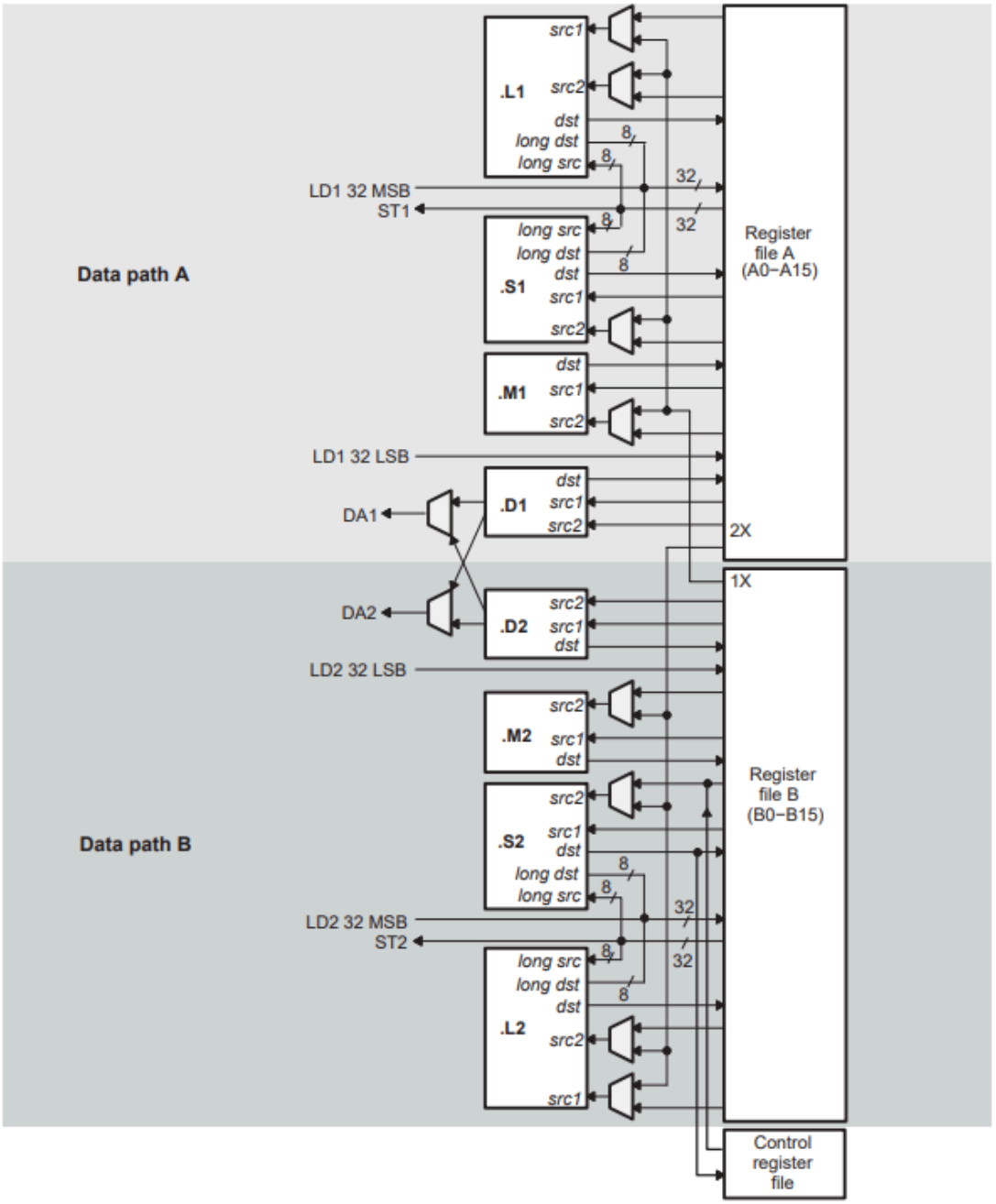
Рассмотрим идею на примере заслуженного сигнального процессора от Texas Instruments TMS320C6x, появившегося в 1997 г.



Фиг. 9 Блок-схема DSP TMS320C67x (отсюда)

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(Double Precision floating point) → SP (Single Precision), 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 ...

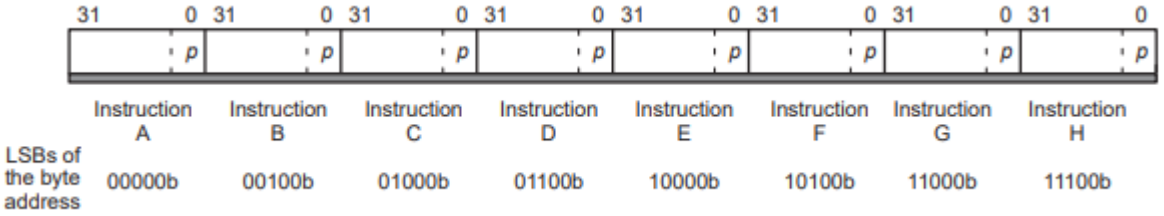


Фиг.10 Схема работы функциональных модулей TMS320C67x (отсюда)

Явный параллелизм.

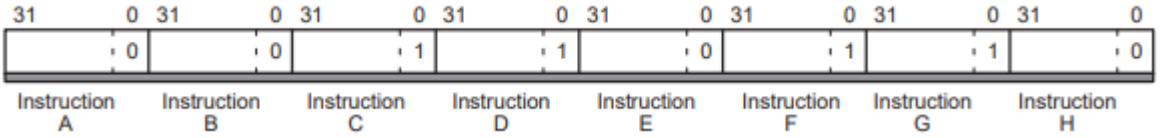
При восьми как бы независимо работающих функциональных устройствах процессор имеет возможность запускать на исполнение до восьми инструкций за такт. Слова “как бы” отражают некоторые зависимости между устройствами, например, один регистр нельзя читать более четырёх раз параллельно. Если в силу ограничений 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.



Фиг.11 пакет параллельных инструкций. LSB - младшие разряды. (отсюда)

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



Фиг.12 Пример частично-параллельного пакета (отсюда)

Пакет с Фиг.12 будет исполняться в следующем порядке

Цикл	Инструкция
1 2 3 4	A B C D E F G H

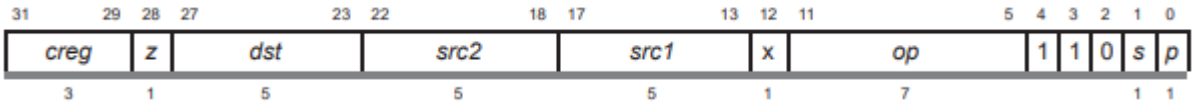
.

На ассемблере параллельные инструкции разделяются двумя символами ||

```
      instruction A

      instruction B

      instruction C
||    instruction D
||    instruction E

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

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.



Фиг.13 структура инструкции ADD

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, см. Фиг 10), 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.

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, это означает, что стоимость решения растёт как экспонента от числа объектов, на практике это означает невозможность решения), математическая задача об укладке рюкзака имеет ряд приближённых схем полностью полиномиального времени (например, кубическое решение, где стоимость зависит как число элементов в третьей степени) 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, уже для двумерного (и более) 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

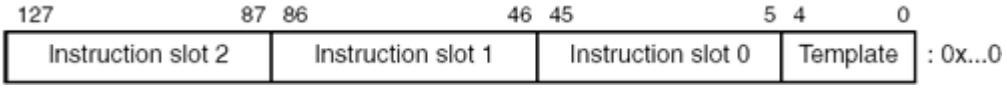
Стоит сказать несколько слов об этой вымершей ныне ветке процессоров от Intel (Itanium (1997...2001), Itanium II (2002)). Само название означает Explicitly Parallel Instruction Computing. Эту архитектуру нельзя назвать VLIW т.к. макро-инструкция содержит всего 3 микрокоманды. Её следовало бы отнести к классу LIW, но, Intel это, вероятно, показалось недостаточно благозвучным, 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.



Фиг.14 формат макрокоманды, отсюда

Макрокоманда (bundle) обычно состоит из трёх 41-разрядных инструкций. Иногда это одна 41 и одна 82 разрядные (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 –

Фиг.15 Коды шаблонов (template) макрокоманды (отсюда)

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.