

Лекция 18

Производительность ЦП

Модель фон Неймана

- Процессор – единственный (нет многоядерности, нескольких процессоров на плате)
- Процессор выполняет инструкции последовательно, т. е. не приступает к выполнению следующей, пока не завершил выполнение предыдущей
- То есть все эффекты выполнения предыдущей инструкции фиксируются перед выполнением следующей – на современных процессорах с точки зрения программиста это правило выполняется

Современные компьютеры

- Многоядерные, многопроцессорные
- Каждое ядро ведет себя по фон-Неймановски
- Для повышения скорости работы одного ядра применяются:
 - Конвейеризация
 - Предсказание переходов
 - Суперскалярность
 - Спекулятивные вычисления (speculative execution)
 - Перестановка вычислений (out of order)
 - Переименование регистров
 - Кэширование

Конвейеризация

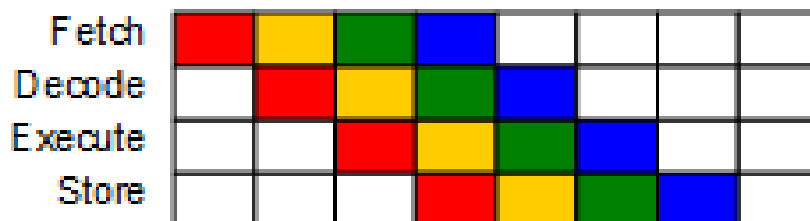
- Шаги исполнения инструкции модельного процессора:
 - Загрузка инструкции из памяти
 - Декодирование инструкции
 - Загрузка аргументов инструкции на устройство (сумматор, умножитель, FPU) для выполнения
 - Выполнение операции
 - Сохранение результатов операции

Функциональные устройства ЦП

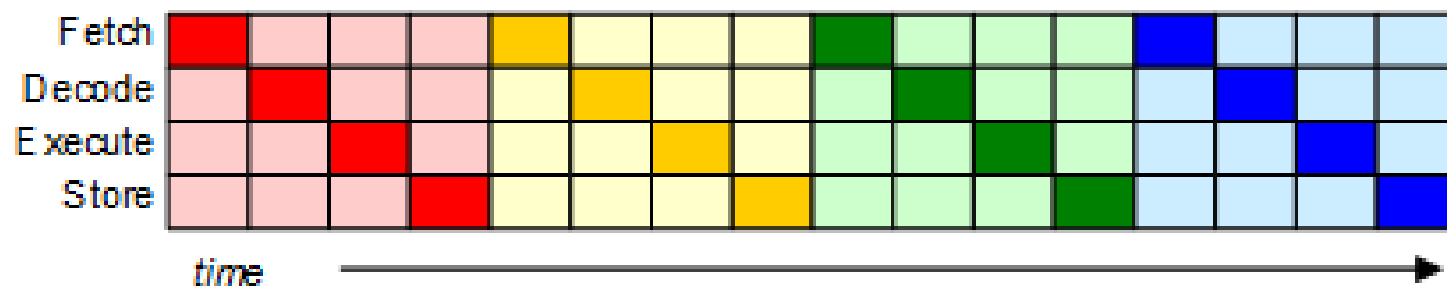
- Устройство управления памятью
- Декодер инструкций
- Регистровый файл (совокупность регистров)
- Сумматор, сдвиговый регистр, умножитель...
 - АЛУ (арифметико-логическое устройство)

Pipelining

pipelined



not pipelined



- “обычный” процессор – 16 тактов
- Конвейерный процессор – 7 тактов

Длина конвейера

- Чем проще операции, выполняемые на одном шаге конвейера, тем выше можно поднять скорость работы (тактовую частоту)
- Но конвейер удлиняется
- Процессоры Intel:
 - Pentium – 5 stages
 - Pentium 4 (Prescott) – 31 (!) stages
 - Современные – 14 stages
- Семейство ARM: 8 – 15 stages

Конфликты (hazards – помехи?)

- Различные hazards не позволяют конвейеру достичь максимальной пропускной способности
- Structural hazards:
 - Два шага конвейера требуют одно функц. устройство (напр. доступ к ОЗУ)
- Data hazards:
 - Данные еще не готовы (RAW – read after write)
- Control hazards:
 - Неизвестно, выполнится ли условие
 - Косвенный переход

Особые ситуации

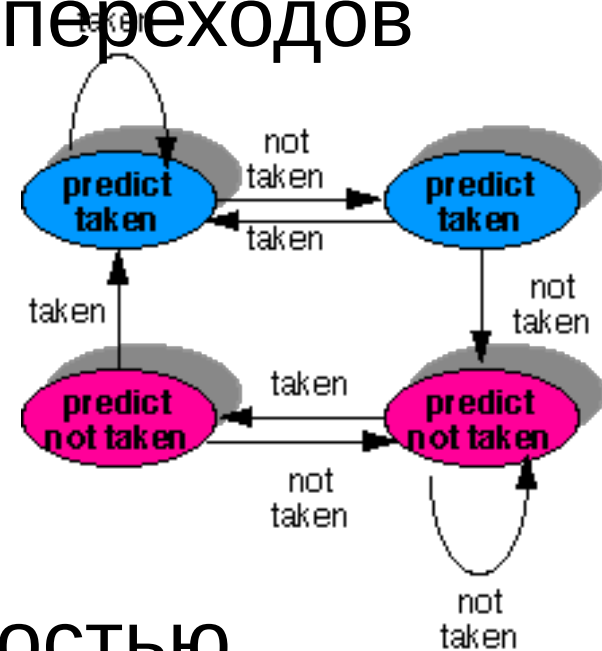
- Pipeline stall (простой) – конвейер не может продолжить работу из-за ожидания долгой операции (например, чтение из памяти)
- Pipeline flush (очистка) – конвейер должен быть очищен из-за перехода в другую точку программы (условный переход, косвенный вызов/переход, системный вызов)
- Чем длиннее конвейер, тем большие задержки вызывает pipeline flush

Предсказание перехода (branch prediction)

- При декодировании инструкции перехода попытаться угадать, будет ли переход
 - Если угадали, конвейер продолжит нормальную работу
 - Если не угадали – pipeline flush
- Инструкции выполняются спекулятивно, то есть запись в регистры или память откладывается до момента, когда с инструкцией перехода не станет ясно
- Предсказание адреса перехода (branch target prediction) – по адресу инструкции (до декодирования) предсказать адрес следующей инструкции

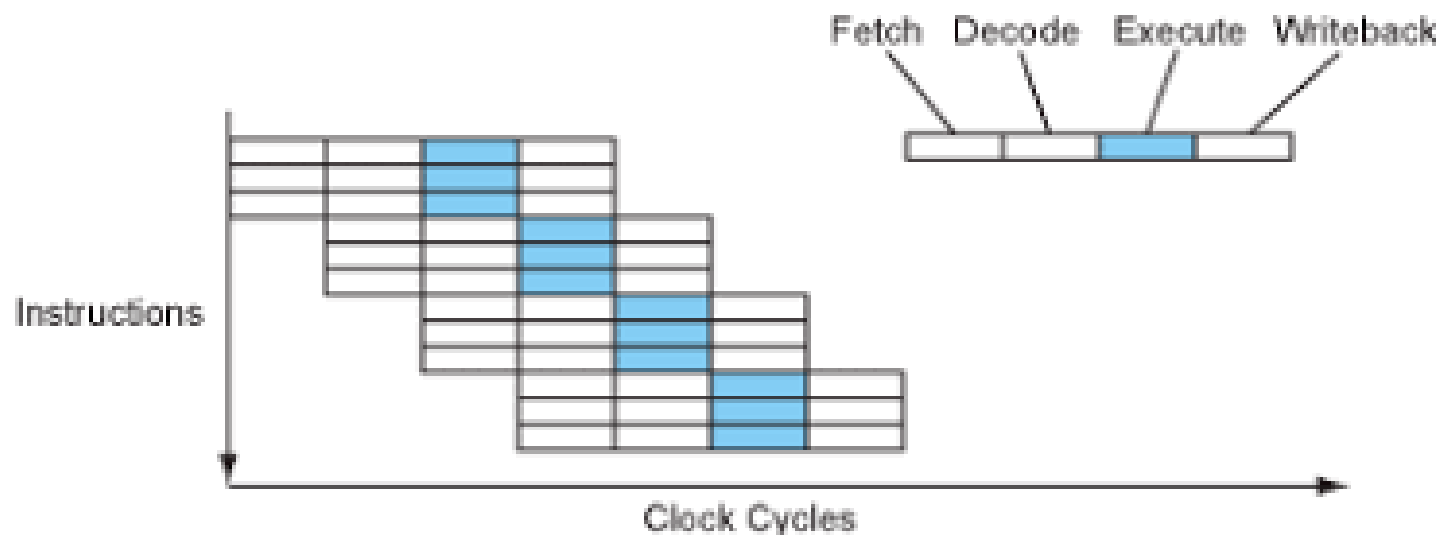
Способы предсказания

- Статический (варианты: все переходит, никогда не переходит, переходит только назад)
- Счетчик с насыщением истории переходов
- Адаптивный двухуровневый
- Нейросети (?)
-
- Современные процессоры предсказывают переходы с точностью 95%



Суперскалярность

- Процессор может выполнять более одной инструкции за такт
 - Увеличиваем число конвейеров
 - Увеличиваем число функциональных устройств



Функциональные устройства

- Intel Skylake:
 - ALU – 4 – add, and, or, test, mov, ...
 - DIV – 1 – div, idiv, ...
 - SHIFT – 2 – sal, shl, ...
 - SLOW INT – 1 – mul, imul, rcl
 - VEC ALU – 3 – vpand, vpor, ...
 - VEC ADD – 2 – vadd, vsub, ...
 - ...
- (теоретически) – до 4 инструкций сложения одновременно, но вычисление адреса – требует ALU

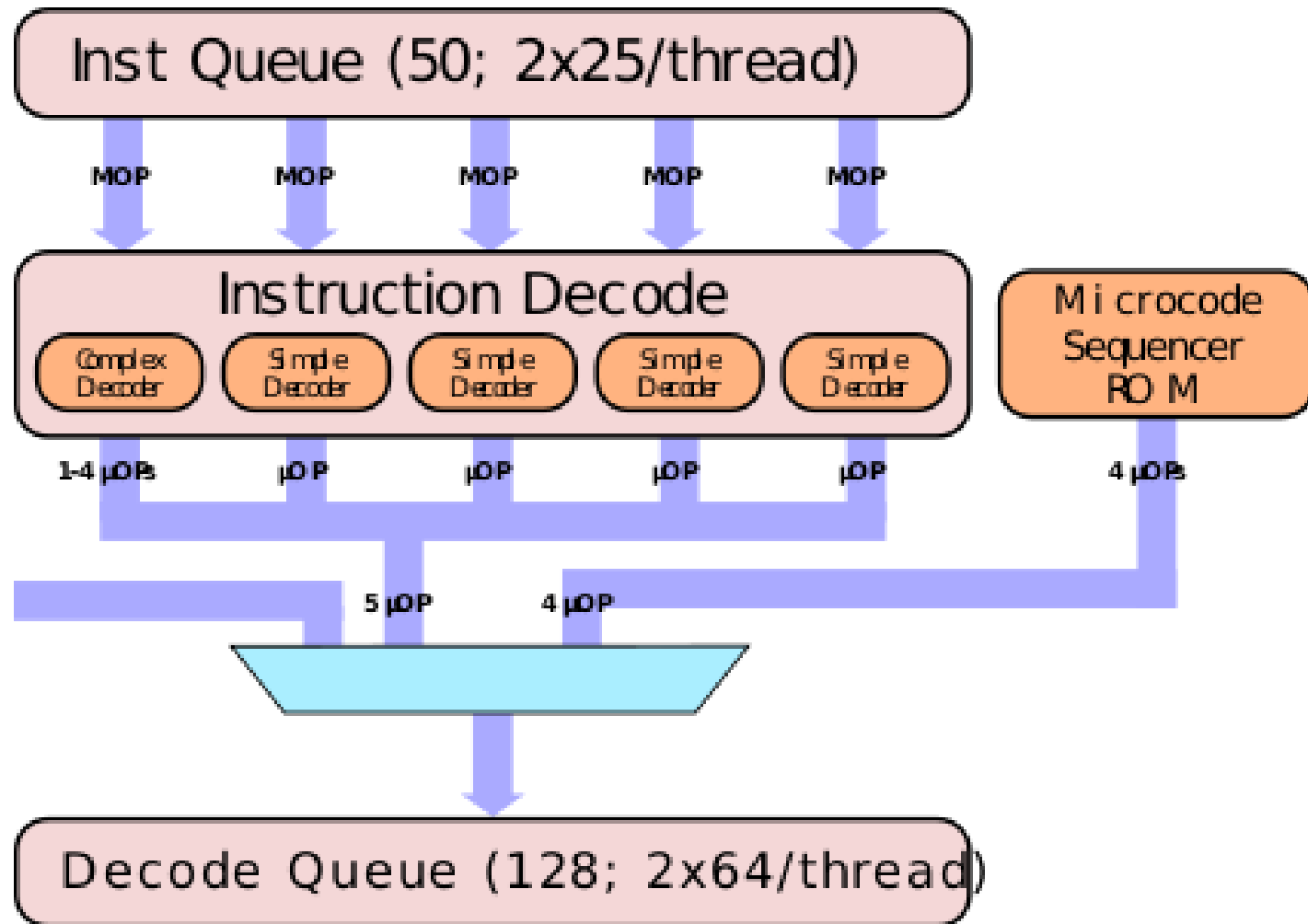
Переименование регистров

- При суперскалярности появляются новые data hazards:
 - Write After Read (WAR) на регистрах, например:
mov (%esi), %eax
add %eax, %ebx
mov \$10, %eax // anti-dependence
add \$20, %eax
mov %eax, %ecx
 - Write After Write (WAW) – должен сохраняться последний результат
- ISA (instruction set architecture) registers переименовываются в hardware registers при выполнении инструкции

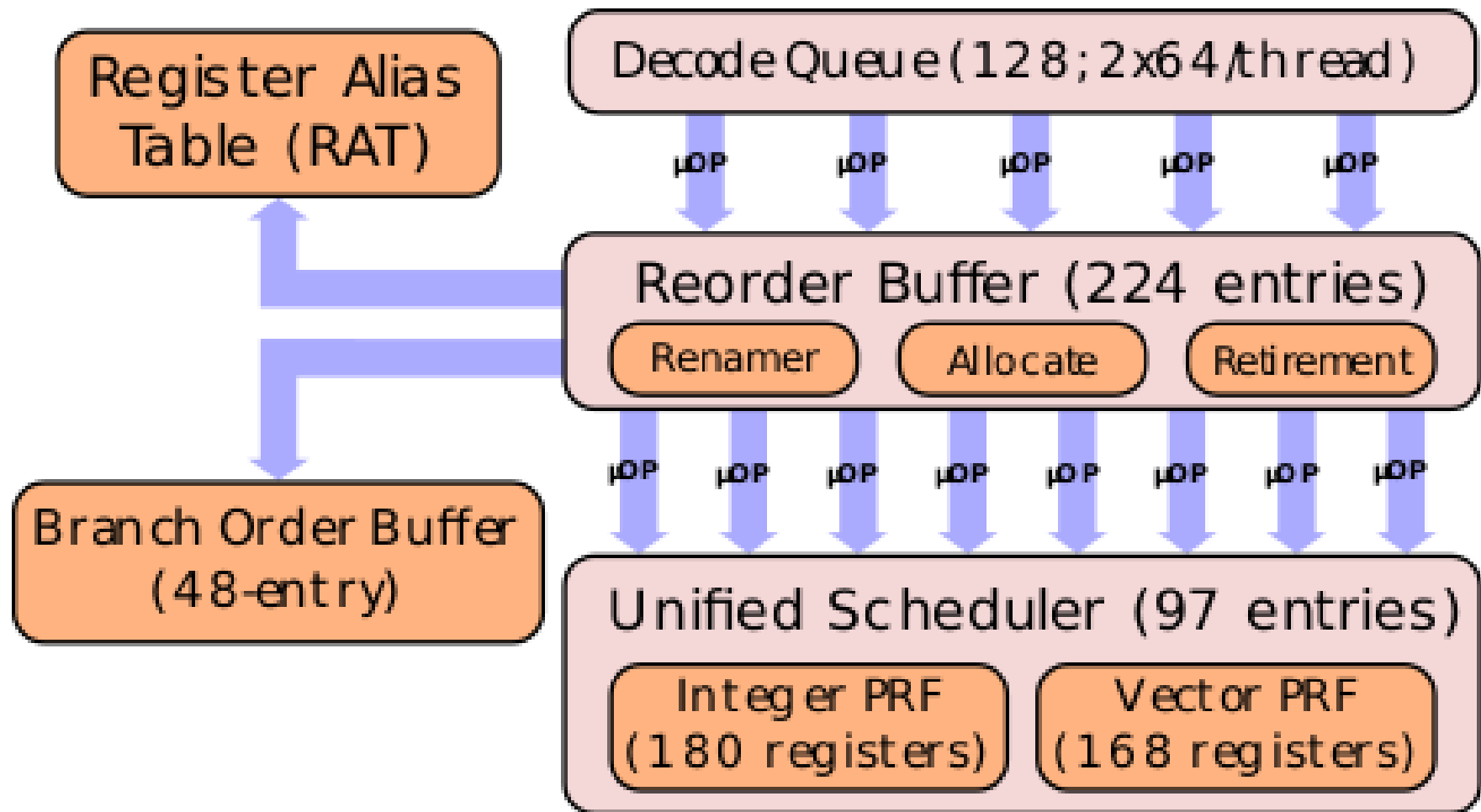
Out-of-order execution

- Развитие идеи pipeline, отказываемся от фиксированного порядка стадий, но переходим к выполнению инструкции по готовности
- Конвейер делится на front-end и execution engine
- Инструкция декодируется в микроинструкции (μOP) во front-end
- Они поступают в execution engine где выполняются по готовности
- Доступ к памяти обрабатывается в memory engine

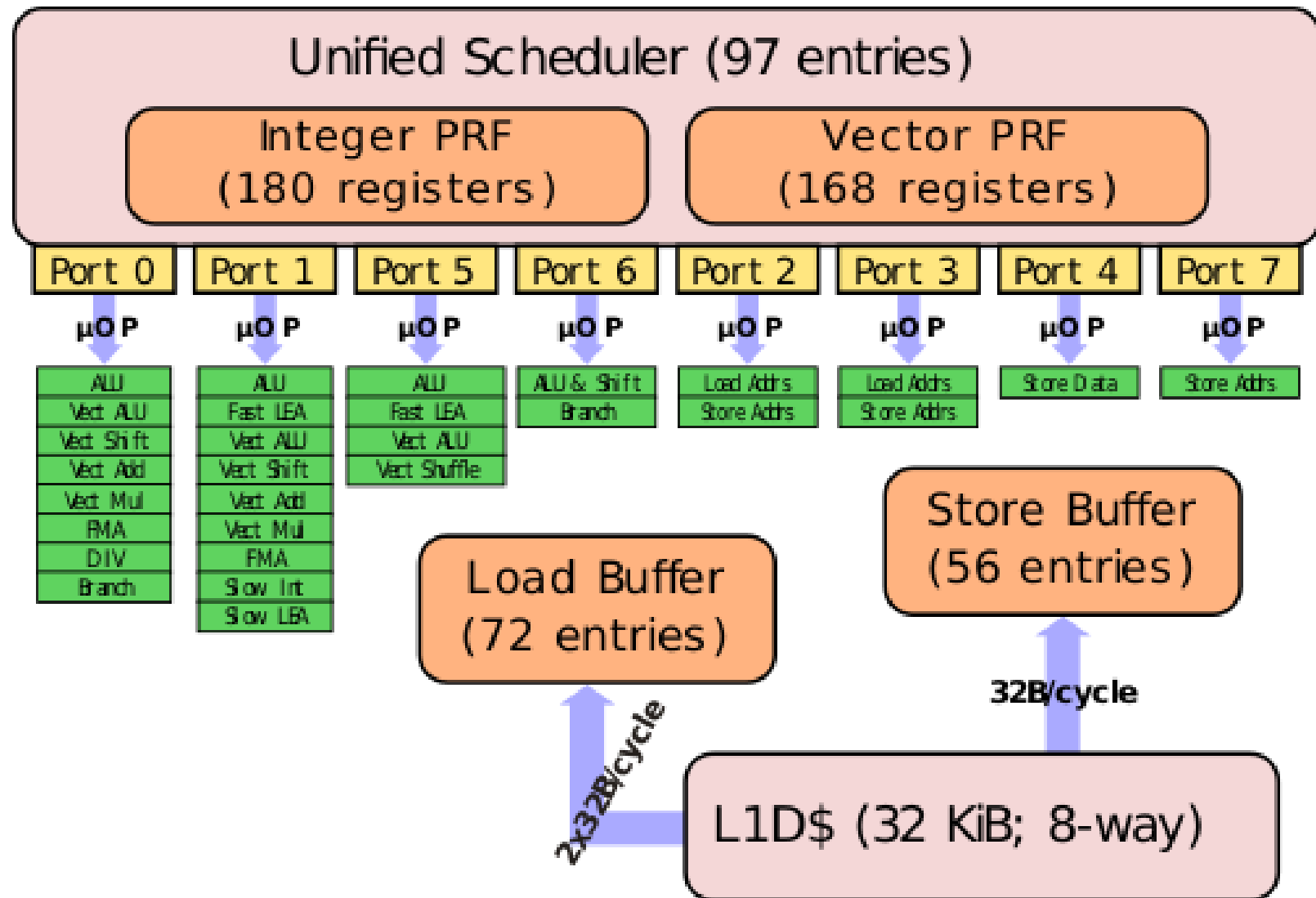
Skylake front-end



Skylake execution engine



Execution engine



Спекулятивные вычисления

- Инструкция выполняется спекулятивно, если на момент ее выполнения еще не известно, должна ли она выполняться вообще
cmp %eax, %ebx
jne out
mov \$10, %ecx // нужно ли выполнять?
- Если обнаружилось, что инструкцию не требуется выполнять, нужно состояние процессора откатить к состоянию “до” выполнения
- Спекулятивные вычисления позволяют поддерживать загрузженность функциональных устройств и снижают число очисток конвейера

Спекулятивные вычисления

- Практически все инструкции выполняются спекулятивно:
mov (%eax), %edx
add \$1, %ecx
- Инструкция mov может завершиться ошибкой недопустимого адреса (page fault)
- В этом случае значение регистра %ecx не должно измениться! Add выполняется спекулятивно.

Meltdown bug

- Взаимодействие подсистемы виртуальной памяти, кеша и спекулятивных вычислений позволяет получить по *побочному каналу* значение ячейки памяти, к которой в обычном режиме нет доступа
- Побочный канал: в теоретически защищенной системе утечка данных происходит средствами, не предусмотренными теоретической моделью:
 - Отслеживание времени выполнения (timing attack)
 - Отслеживание потребления энергии
 - Отслеживание паразитных электромагнитных излучений

Timing attack

- Измеряя время выполнения можно определить данные, над которыми проводилась операция
- Пример: проверка пароля с помощью `strcmp`
 - `strcmp` – завершает выполнение при первом несовпадении
 - Если можем замерить время и определить число операций, то можно подобрать строку, перебирая символы, начиная с первого

Проверка нахождения значения в кеше

- Напрямую невозможно определить, находится ли ячейка памяти в кеше
- Но можно замерить время загрузки значения ячейки в регистр. Если оно “невелико”, ячейка была в кеше
- Схема реализации:
rdtsc // читаем внутренний счетчик тактов
mov (%eax), %ebx // обращаемся к (%eax)
rdtsc
// вычисляем время обращения, сравниваем

“Закрытая” виртуальная память

- В таблице страниц бит “U” отвечает за доступность страницы в пользовательском режиме
- На самом деле таблицы страниц содержат отображение всей памяти процесса, всей **физической памяти и виртуальной памяти ядра**, но большая часть виртуальной памяти закрыта для процесса
- Но бит U проверяется отдельно от обращения к памяти
- **Обращение к памяти с недопустимым битом U выполняется и считанное значение попадает в кеш!**

Идея meltdown

```
// пусть %ebx будет содержать 0
mov $secretaddr, %eax    // адрес в ядре
div %ebx                 // div – медленная
// деление на 0 даст исключение, но
// последующие инструкции будут
// исполняться спекулятивно
mov (%eax), %ebx         // в обход бита U
and $15, %ebx            // младшие 4 бита
shl $6, %ebx             // cache line = 64
mov mydata(%ebx), %ebx
```

Идея meltdown

- Спекулятивное выполнение `mov mydata(%ebx), %ecx` приведет к тому, что ячейка памяти с адресом, содержащим 4 бита секретного значения в памяти ядра, окажется в кеше
- Замеряя время обращения к кешу можно понять, чему был равен `%ebx`
- Отсюда можно извлечь значение 4 битов по заданному адресу в памяти ядра

Further reading

- Pro Intel/AMD arch:
<http://www.agner.org/optimize/microarchitecture.pdf>
- Pro meltdown:
<https://meltdownattack.com/>