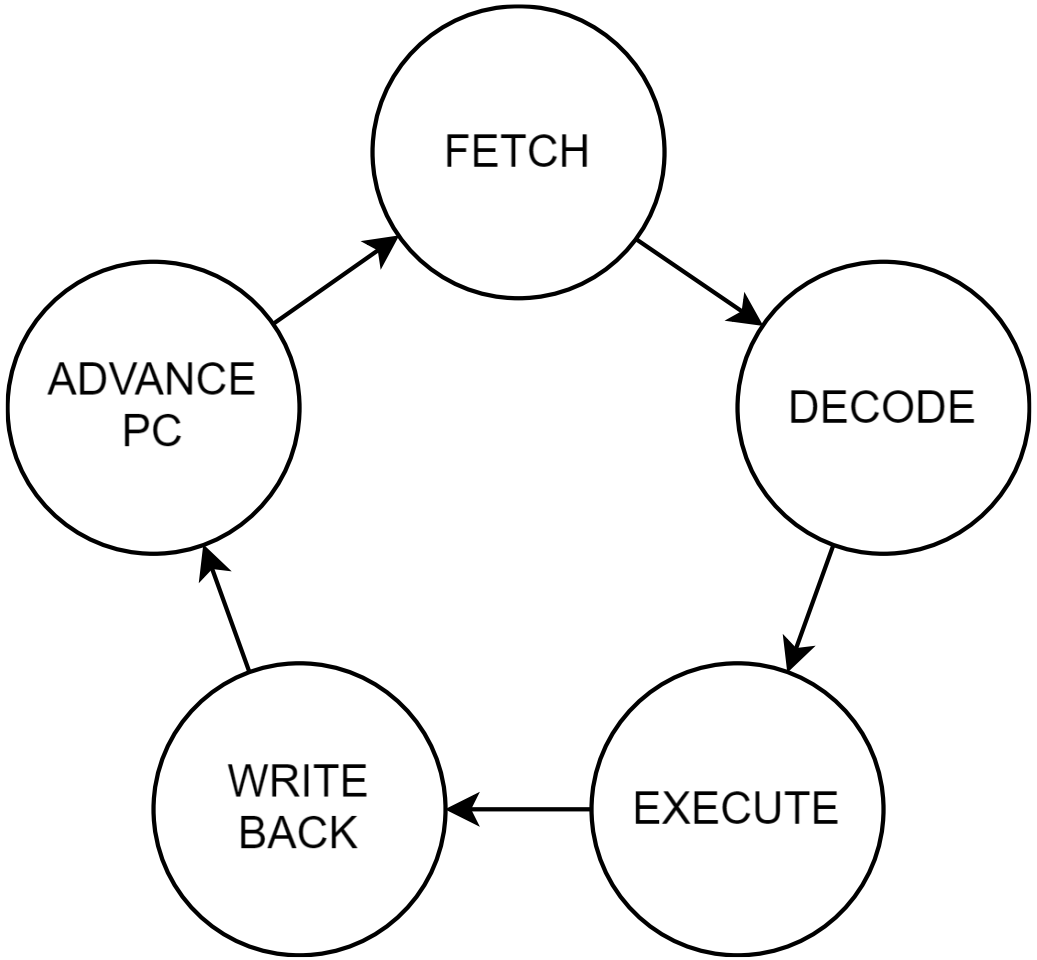
Ускорение симуляции

МФТИ
Осень 2024

Дисклеймер

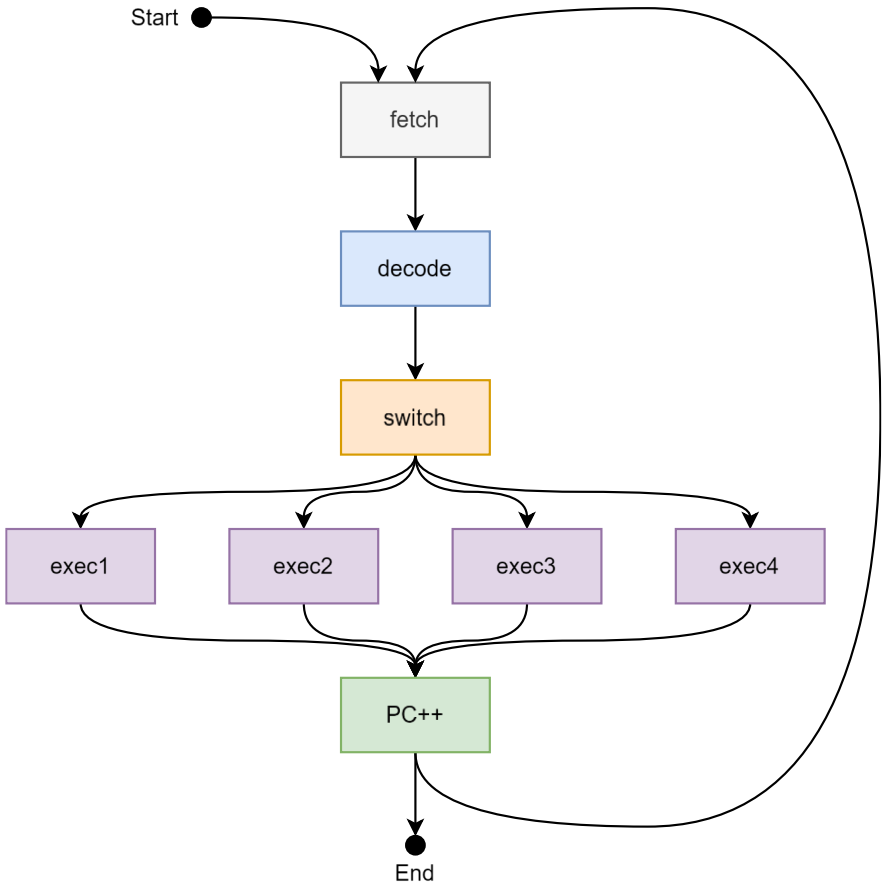
На данном занятии будут рассмотрены общеизвестные подходы и способы по ускорению симуляции. Реальное ускорение может зависеть от хостовой машины, симулируемой архитектуры, компилятора и множества других мелочей. Не исключено, что некоторые подходы могут даже ухудшить реальную производительность. Единственным критерием истинности в исследовании производительности является экспериментальный замер.

Функциональный симулятор: интерпретация



Функциональный симулятор: интерпретация Наивная реализация

- Инструкции интерпретируются в цикле
- Каждая инструкция декодируется перед исполнением
- С помощью switch’а происходит переход к интерпретации текущей инструкции
- Попробуйте покритиковать такой подход



Функциональный симулятор: интерпретация Наивный подход к декодированию

```
void *mymemcpy(void *dst,
               void *src,
               size_t N) {
    for (size_t i = 0; i < N; ++i)
      ((unsigned char *)dst)[i] =
        ((unsigned char *)src)[i];
    return dst;
}
```

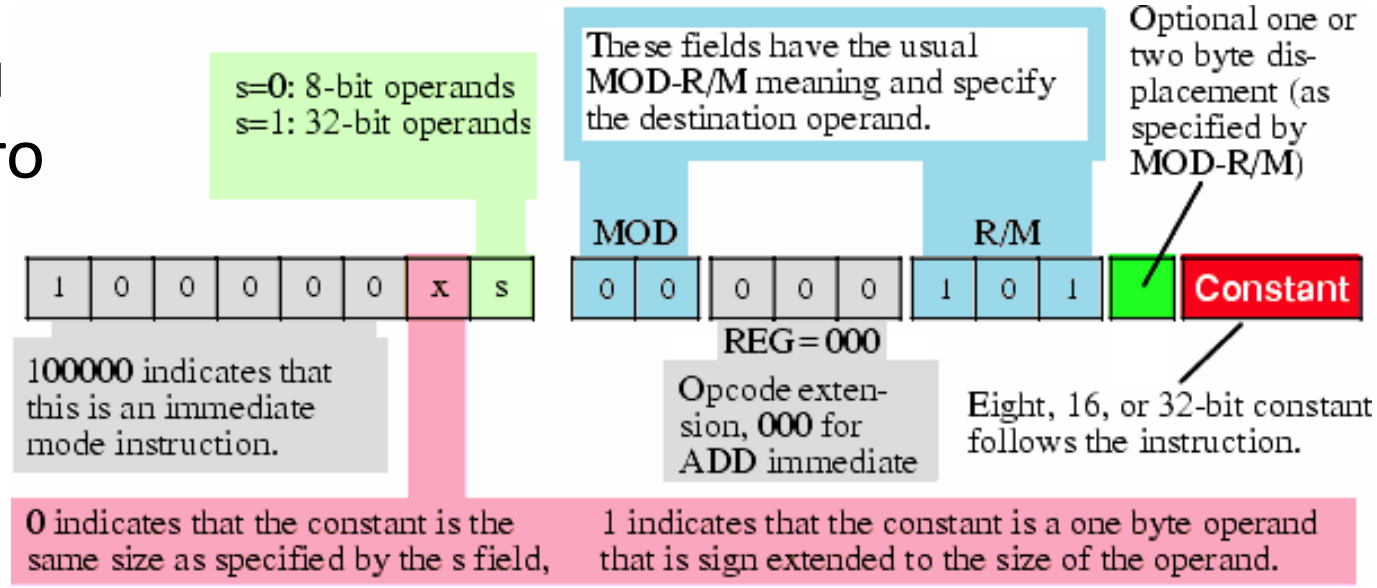
В наивной реализации инструкции, соответствующие метке `.L3` будут декодированы множество раз

```
mymemcpy:
        beq     a2,zero,.L2
        mv      a5,a0
        add     a2,a1,a2
.L3:
        lbu     a4,0(a1)
        addi    a1,a1,1
        addi    a5,a5,1
        sb      a4,-1(a5)
        bne     a1,a2,.L3
.L2:
        ret
```

godbolt.org

Пример декодирования сложной инструкции

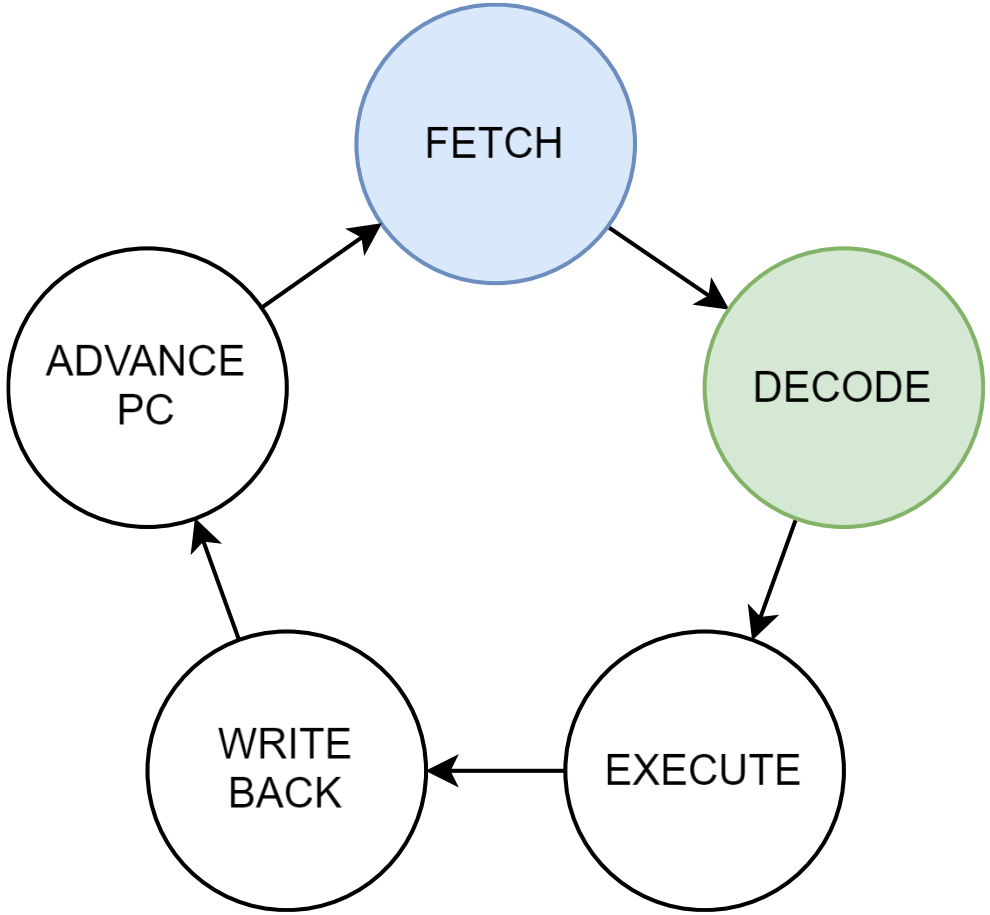
- В x86 инструкции могут иметь сильно разный размер
- Зачастую размер инструкции сложно определить до самого конца
- Так, размер константы в данном примере зависит от значения поля MOD-R/M



https://www-user.tu-chemnitz.de

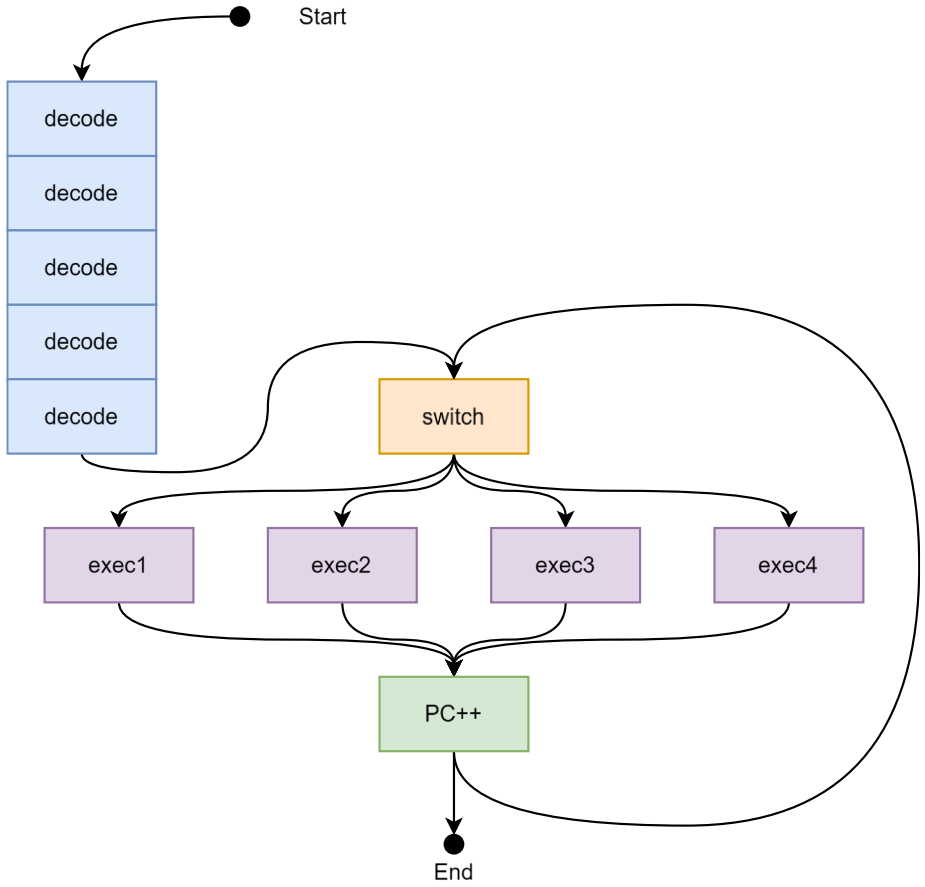
Функциональный симулятор: интерпретация Наивный подход к декодированию

- Декодировать одну инструкцию может быть затратно
- Многократное декодирование даже простой кодировки влияет на общую скорость интерпретации
- Что можно предложить для оптимизации декодирования?



Функциональный симулятор: интерпретация Оптимизация декодирования

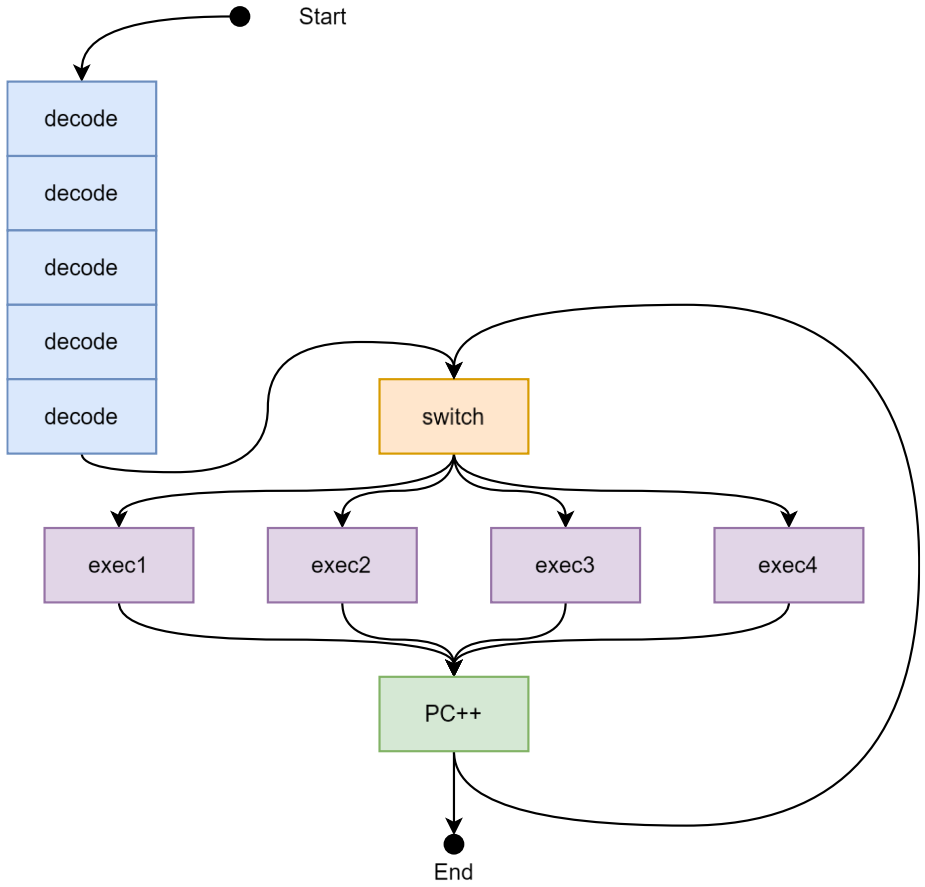
- Декодируем все инструкции перед интерпретацией
- Перемалываем сразу декодированные инструкции
- Можете ли вы покритиковать данный подход?



Функциональный симулятор: интерпретация Оптимизация декодирования

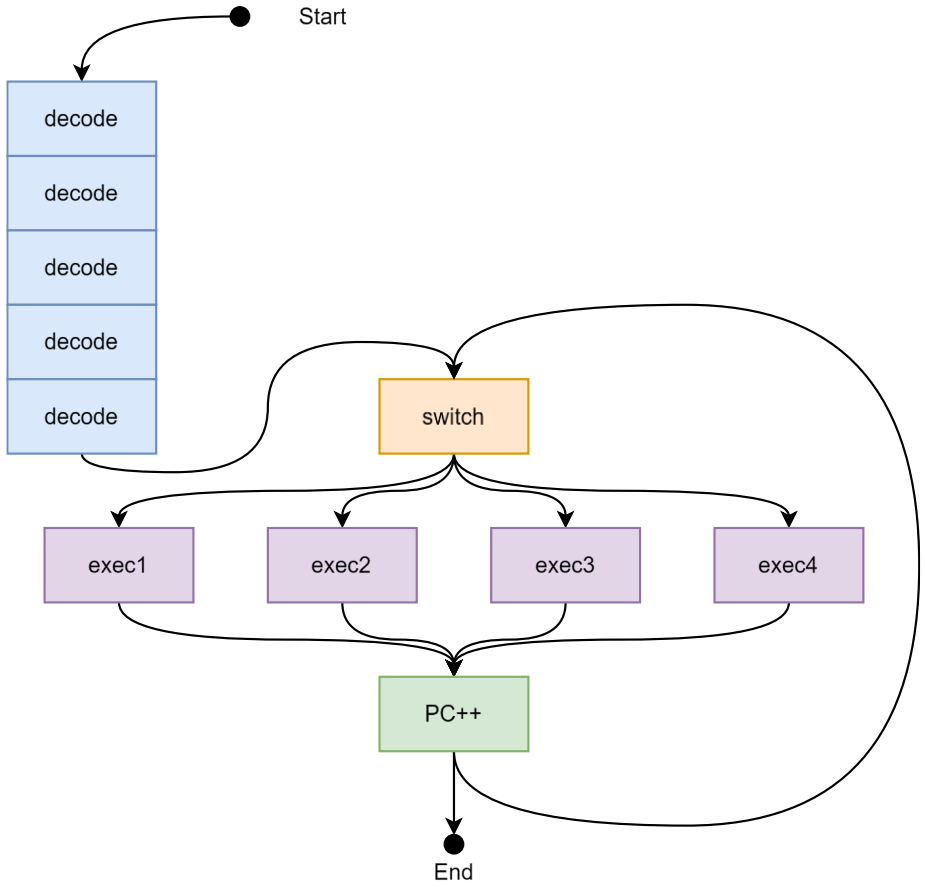
- Декодируем все инструкции перед интерпретацией
- Перемалываем сразу декодированные инструкции

- Можете ли вы покритиковать данный подход?
  - Необходимо не забывать декодировать переписанные в процессе выполнения инструкции



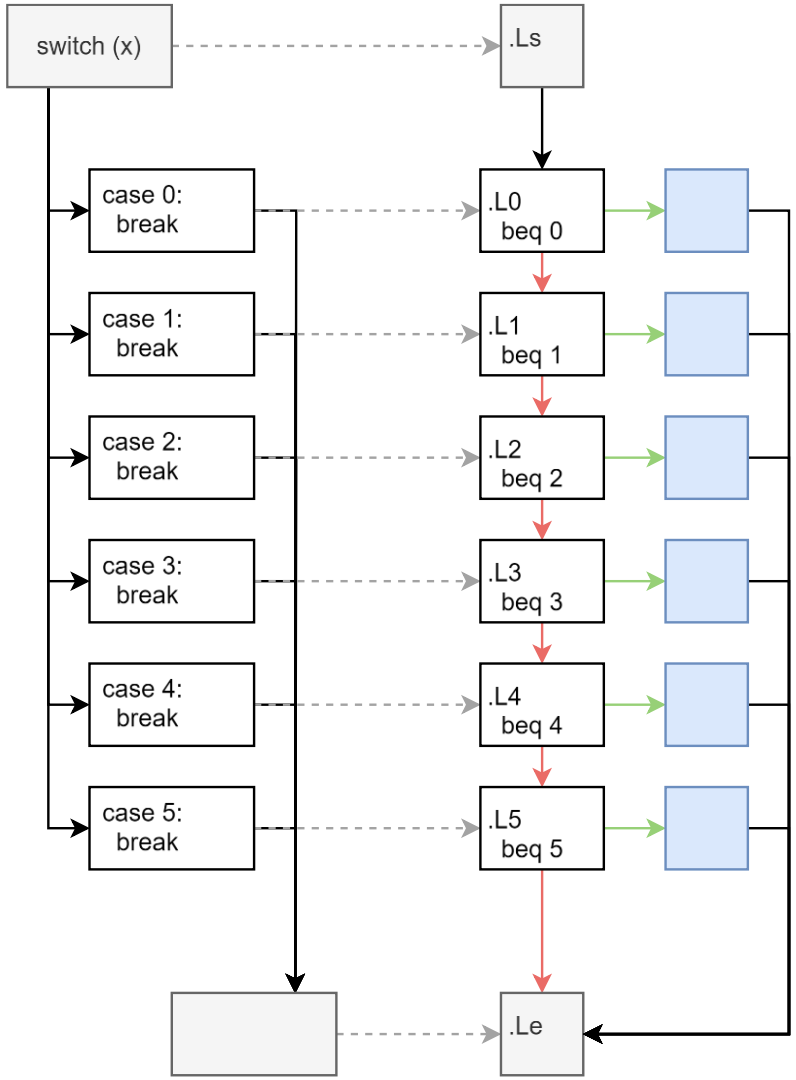
Функциональный симулятор: интерпретация

- Предположите что еще может замедлять наш симулятор



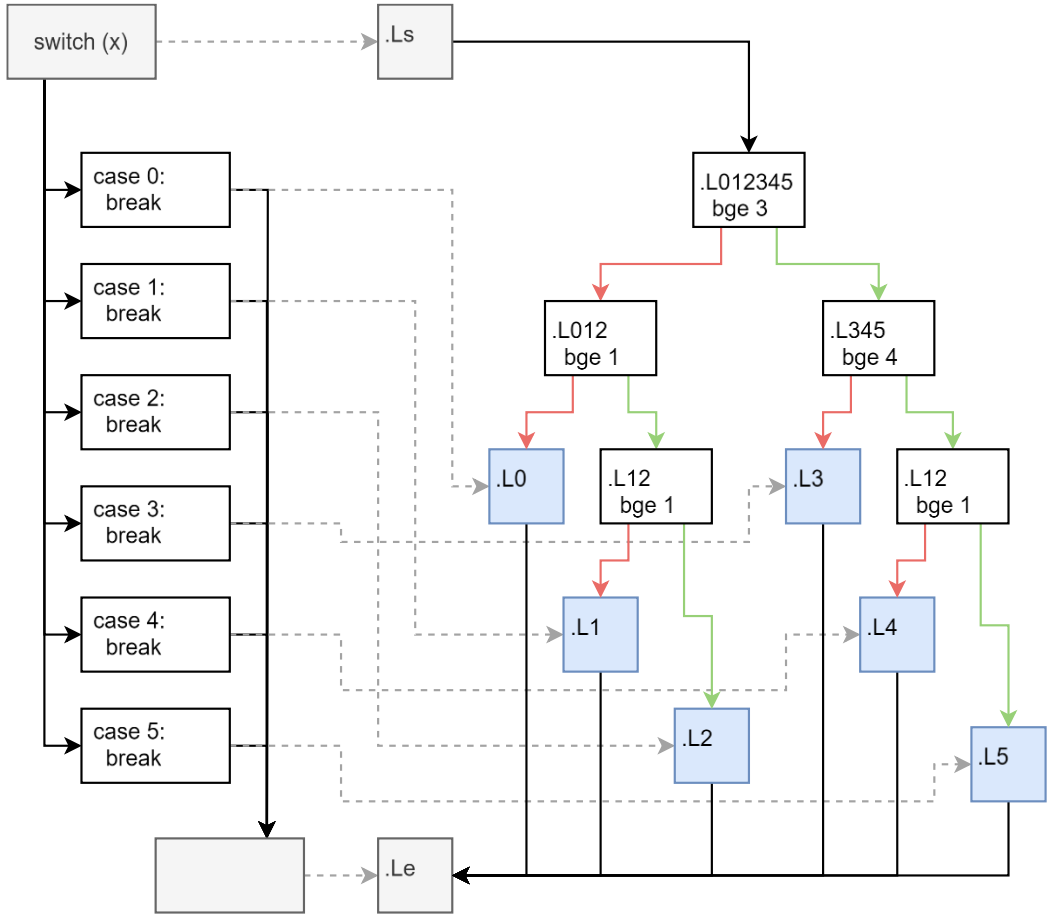
Что такое switch для компилятора?

- Семантически switch – это последовательность if-else выражений
- На `-O0` компилятор превратит switch в просто последовательность условных переходов
- Какая асимптотическая сложность у такого подхода?
- Может ли компилятор это как-нибудь оптимизировать?



Оптимизация switch: бинарное дерево

- `switch-case` по сути выполняет задачу поиска
- Сделаем поиск за `O(log(N))`, используя бинарное дерево
- Не кажется ли, что можно довести до `O(1)`?



Оптимизация switch: таблица переходов

- Мы статически знаем адреса всех блоков, которые соответствуют `case` блокам, можем ли мы просто посчитать адрес нужного `case` блока и перейти сразу к нему?
- Да, можем, такой метод называется таблицей переходов (`jump-table`)
- Если размер инструкций во всех `case` одинаковый, то компилятор может вычислить адрес перехода и сделать безусловный переход:
  - `start` – адрес метки первого `case` блока
  - `index` – номер `case` блока
  - `stride` – размер каждого `case` блока
  - Адрес перехода: `dst = start + index * stride`
- Иногда это еще называют *вычисляемый goto* (`computed goto`)

Оптимизация switch: таблица переходов

Если размер инструкций в case блоках различается, то все решается дополнительным уровнем косвенности

```
switch(EnumValue) {
   case Val1: // EnumValue == 0
     return Obj.method1();
   case Val2: // EnumValue == 1
     return Obj.method2();
   case Val3: // EnumValue == 2
     int x = Obj.method3();
     return Obj.method4(x);
}
```

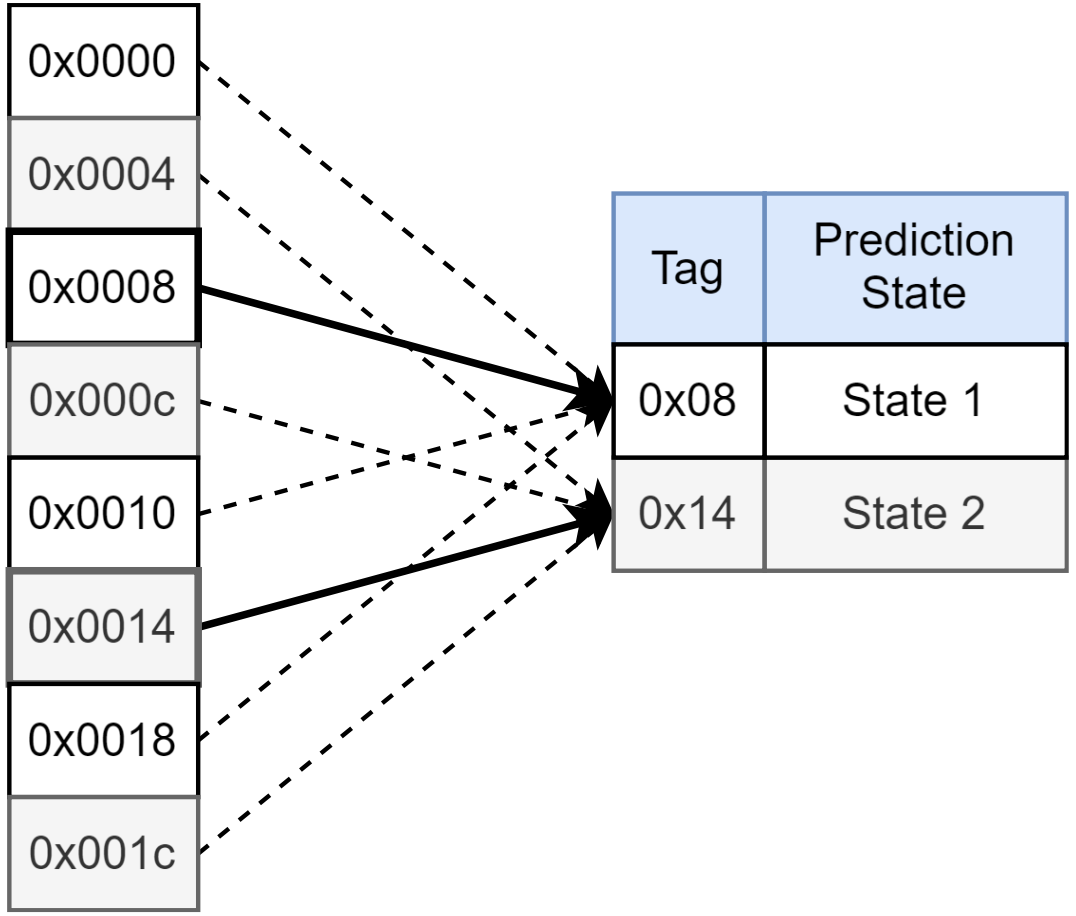
```
.L0:
   la   t0, .table # load address of .table
   slli t1, a0, 2  # offset = case num * instr size
   add  t0, t0, t1 # addr = start + offset
   jr   t0         # the jump
.table:
   j Val1
   j Val2
   j Val3
.Val1:
.Val2:
.Val3:
```

Критика таблицы переходов

- В rv32i примерно 30+ инструкций
- Между инструкциями в программе как правило нет никаких зависимостей по очередности
  - Например, после add может идти любой опкод, и такая ситуация распространяется на каждый case
- Почему BPU плох в такой задаче?
- Что можно с этим сделать?

BPU подробно

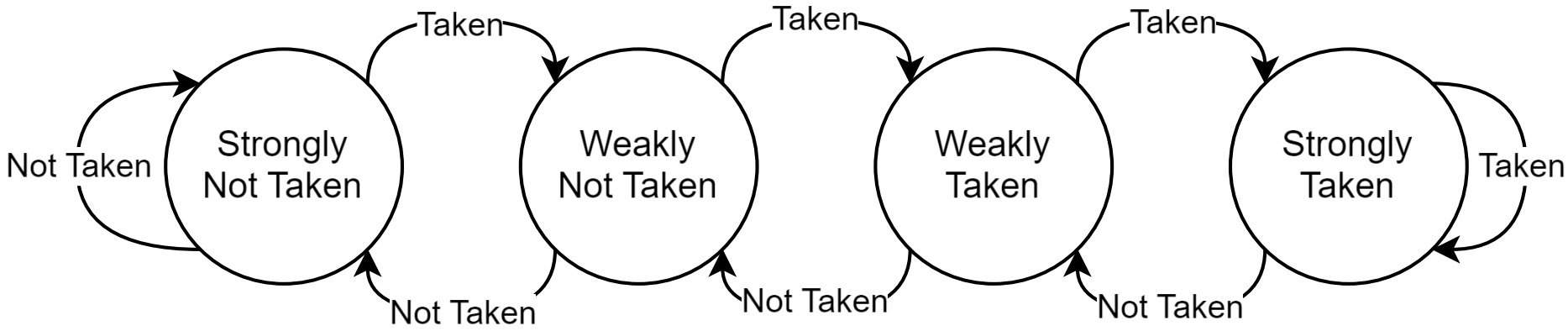
- По устройству branch predictor представляет таблицу
  - Tag это первые N бит адреса branch инструкции, которую отслеживает BPU
  - State это состояние предсказателя, соответствующее отслеживаемому branch’у
- Алгоритмы предсказания в BPU бывают разные



Предсказатель на основе конечного автомата

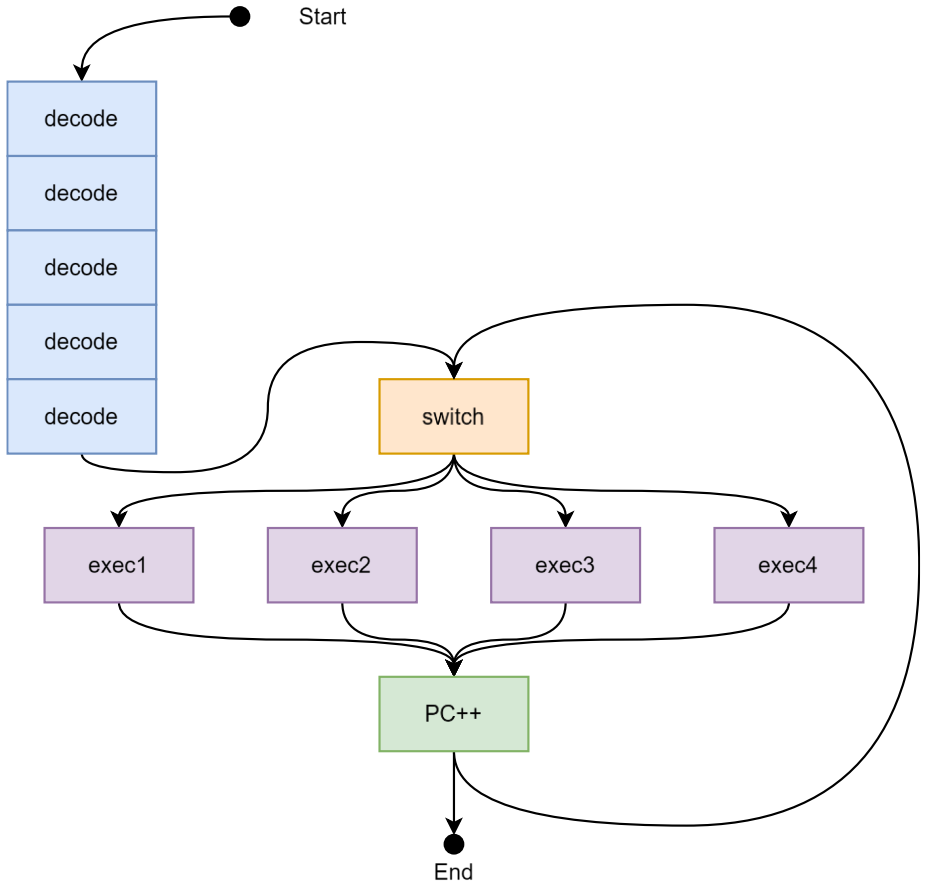
Конечный автомат – абстракция, описывающая устройство, которое имеет один вход, один выход и в каждый момент времени находится в одном из конченого числа состояний

Пример предсказателя на основе конечного автомата – 2-битный насыщающийся счетчик (*saturation counter*):



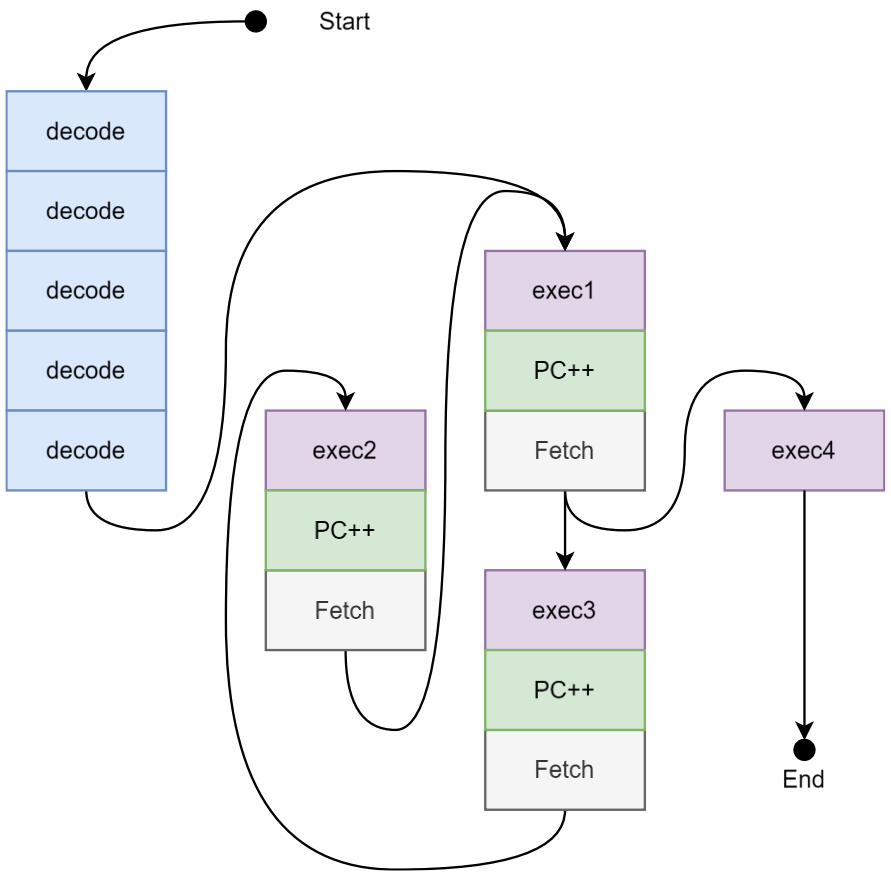
Функциональный симулятор: разбираемся с тормозящим switch’ем

- Таким образом, BPU делает предсказания, основываясь на адресе и ограниченной истории переходов
- В нашем случае история переходов абсолютно хаотическая
- Можем ли мы как-то помочь BPU?



Функциональный симулятор: разбираемся с тормозящим switch’ем

- Можем ли мы как-то помочь BPU?
- Не зная конкретный алгоритм предсказания попробуем размазать единственный переход по нескольким местам:
  - После интерпретирования инструкции не прыгаем в общую точку, а прыгаем в код, интерпретирующий следующую инструкцию
- Данная техника называется шитый код (*threaded code*)



Шитый код

- Идея клевая, но хотели бы вы ее реализовать?
- Во-первых, придется использовать goto
- Во-вторых, GNU extension, которое позволяет брать адрес метки (оператор &&)
- В-третьих, компилятор будет всячески оптимизировать и схлопывать размазанные goto
- В gcc-5.1 появилась оптимизация автоматически превращающая `switch` в шитый код (`-fthread-jumps`)

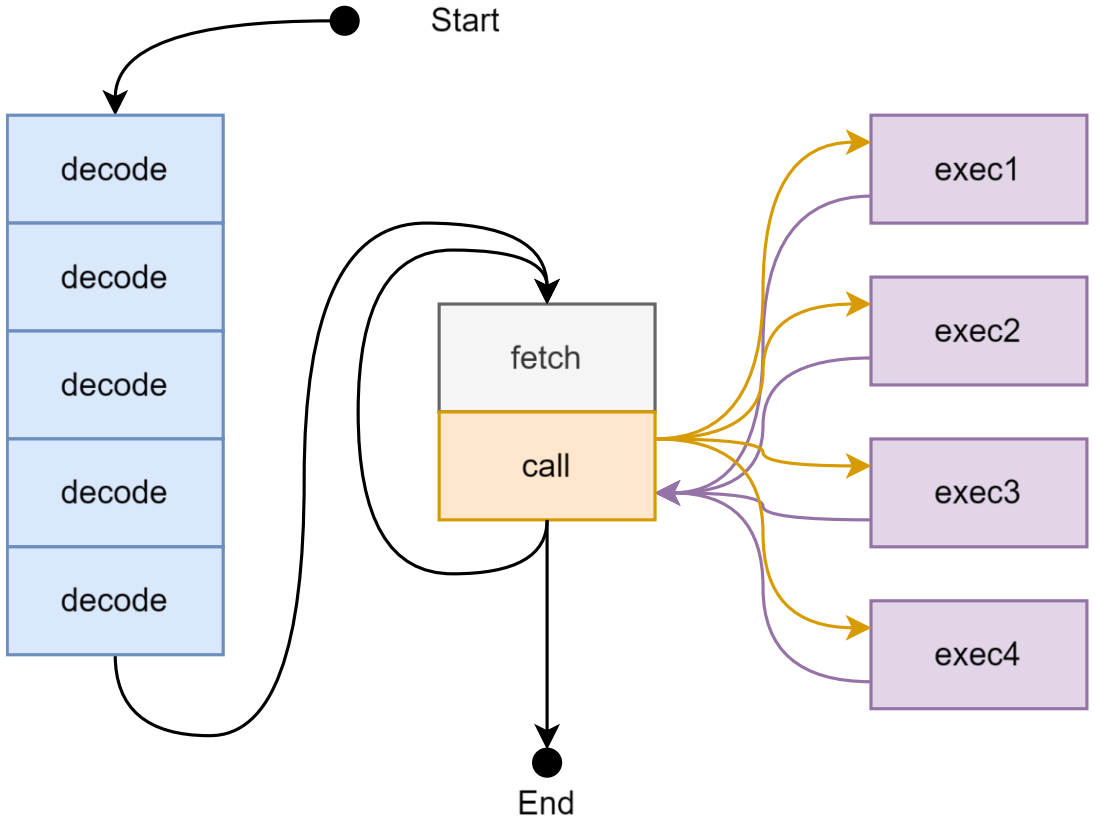
```
const void *labels[] = {&&case_add,
                                ...};
Instruction *inst = fetch(...);


case_add:
    do_add(inst);
    inst = fetch(...);
    goto *labels[inst->opc];
...
```

Как делают приличные люди

- Если приведенные аргументы вас не пугают, то вы бесстрашны
- Обычные смертные предпочитают находить более элегантные подходы
- Стандартный подход «ручного unswitching’а» заключается в вынесении всего кода из case блока в функцию, после чего все такие функции кладутся в одну таблицу функций
- Хм, очень напоминает таблицу переходов…

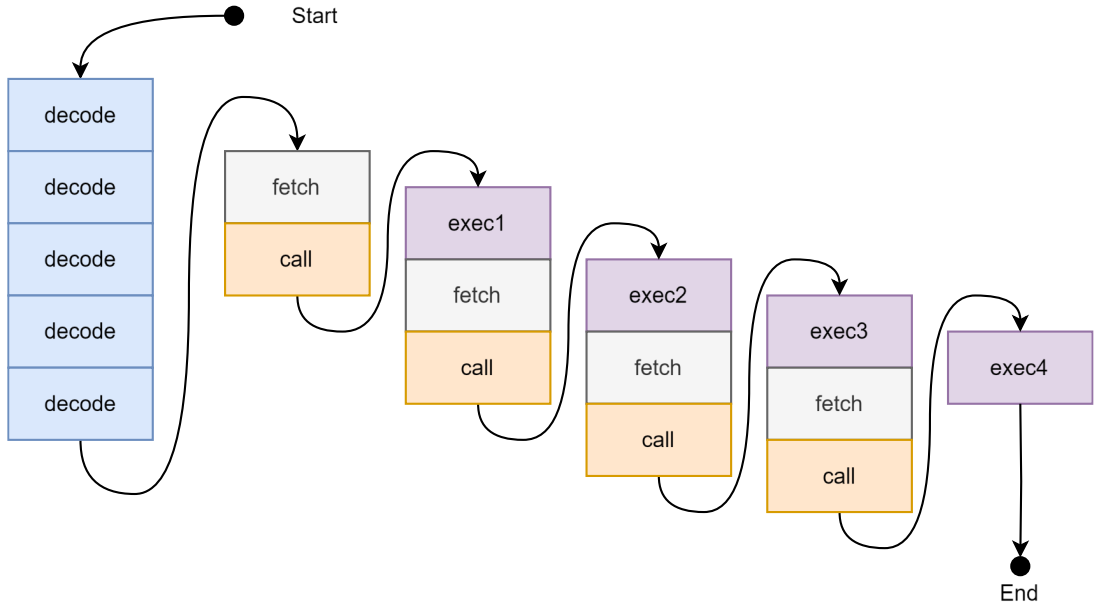
Разбираемся со switch’ем по-джентльменски

- Так это и есть таблица переходов, но в цивилизованном виде
- Только вместо computed goto у нас computed call
- Который тоже довольно плохо предсказывается…
- Давайте тут тоже попытаемся что-нибудь пришить



Разбираемся со switch’ем: сшиваем функции

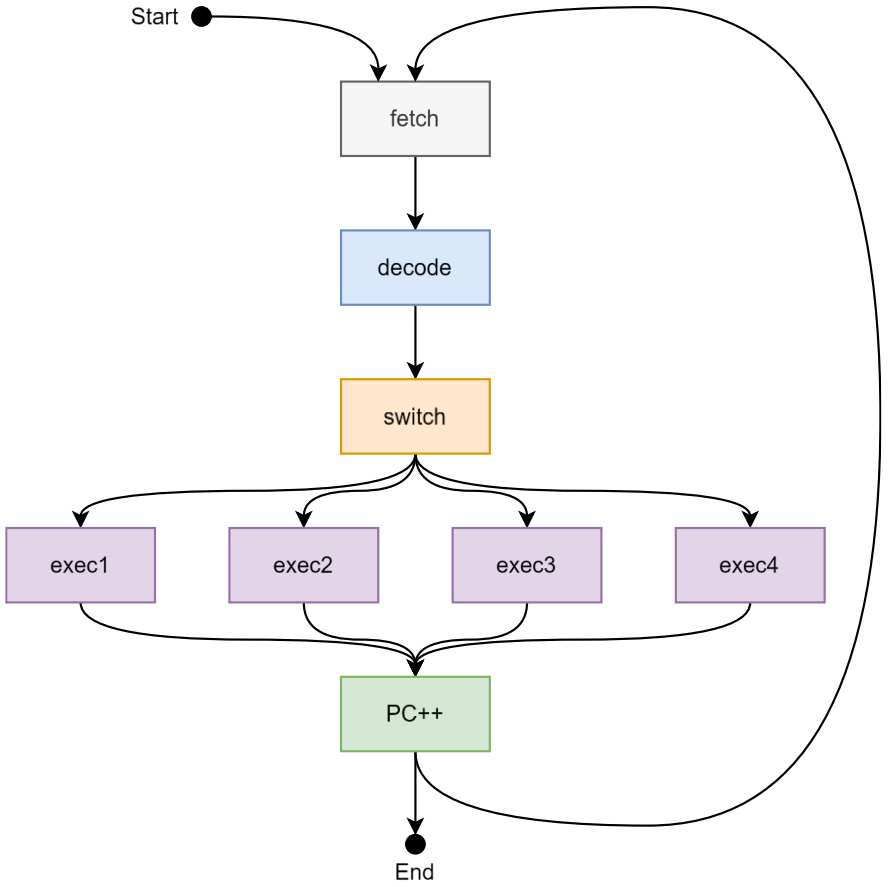
- В шитом коде мы прыгали из конца одного case, в начало другого
- Здесь сделаем так же – будем вызывать следующую функцию в конце предыдущей
- Не возникнут ли у нас проблемы?



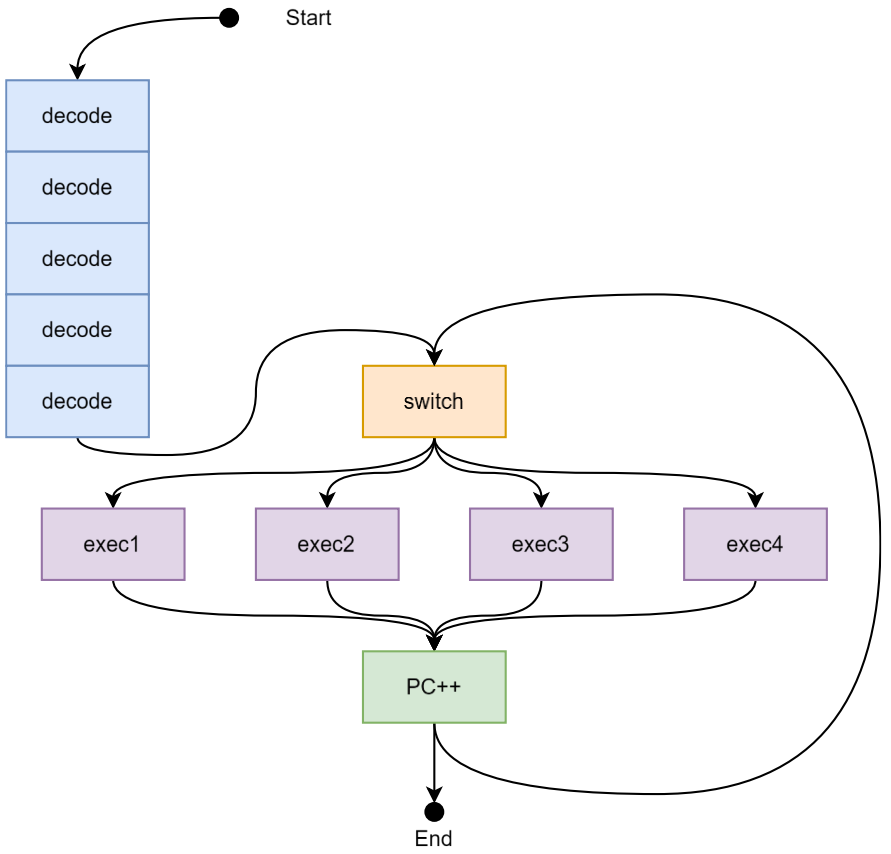
Tail call optimization

- Проблем с переполнением стека не будет, потому что нас спасет компилятор
- Когда вызов одной функции находится в самом конце другой, такой вызов называют хвостовым (*tail call*)
- При большой глубине вызовов переполняется стек, потому что с каждым вызовом аллоцируется дополнительная память на стеке
- Оптимизация хвостового вызова (*tail call optimization*) позволяет не выделять место на стеке под фрейм вызываемой функции

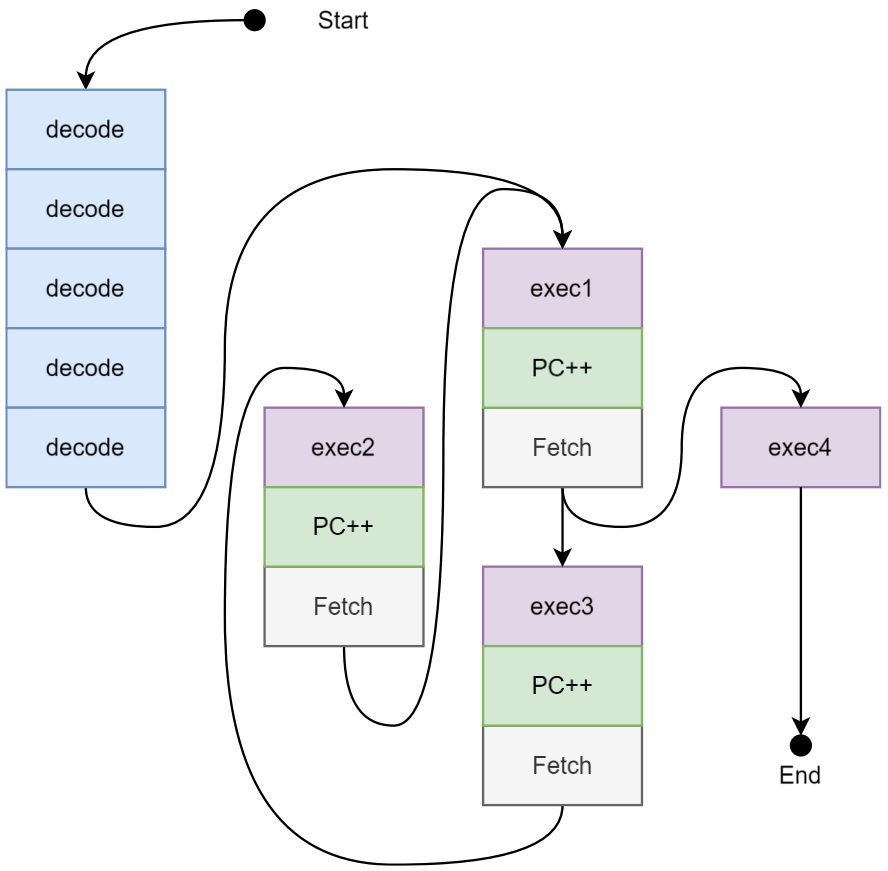
Повторим шаги: наивная реализация



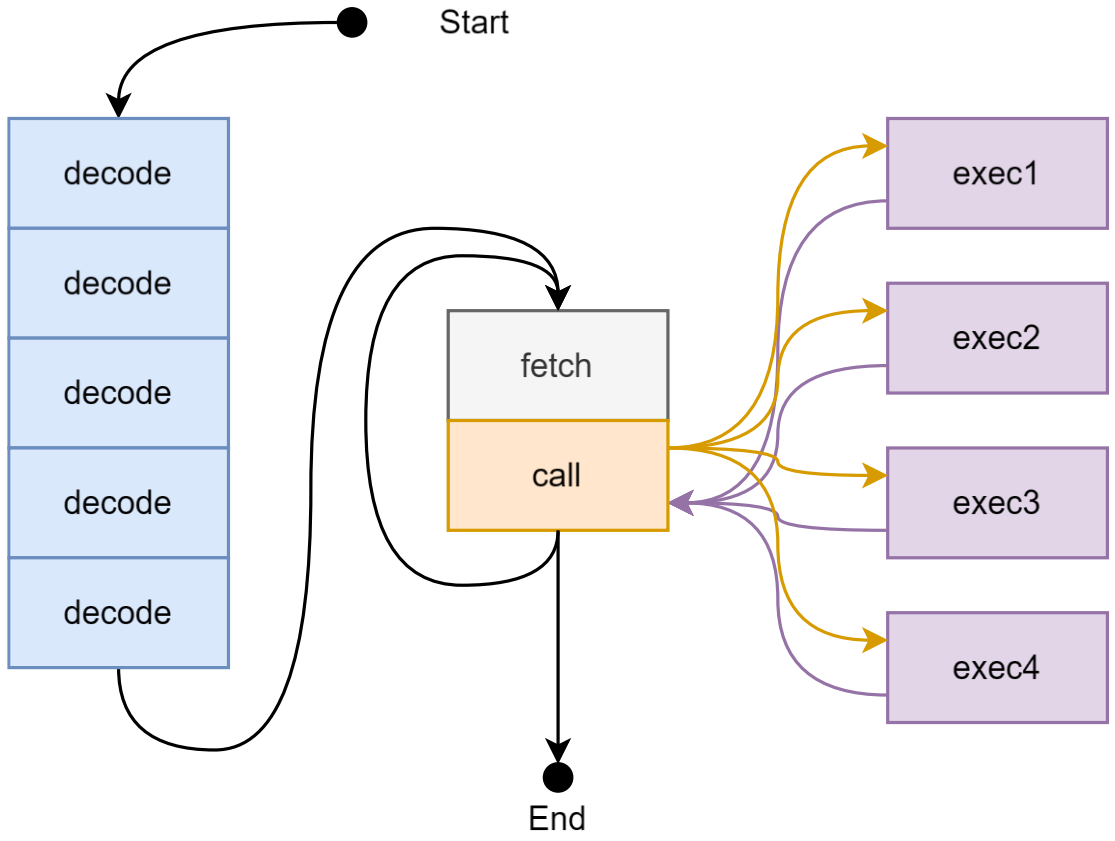
Повторим шаги: предварительное декодирование



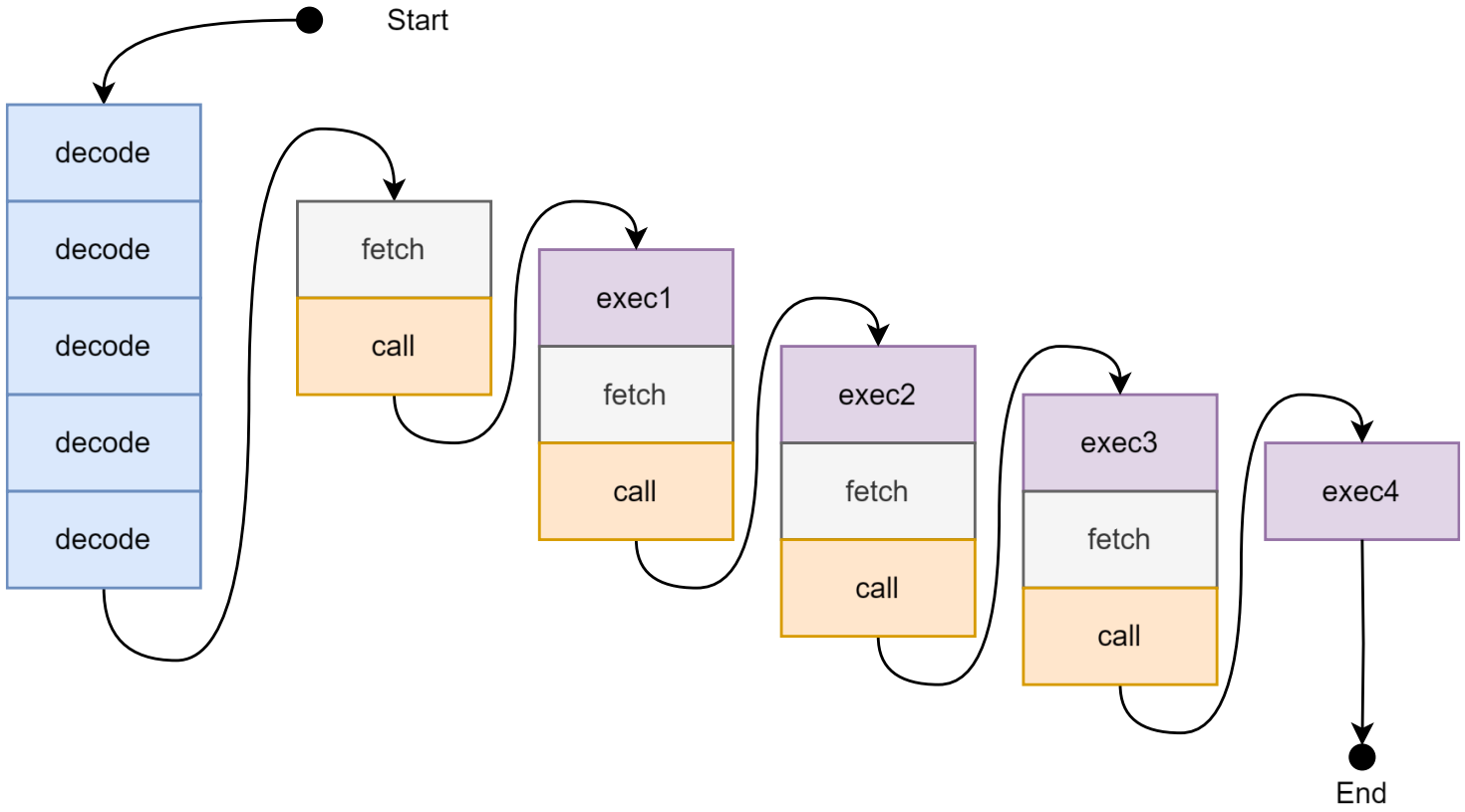
Повторим шаги: шитый код



Повторим шаги: таблица вызовов



Повторим шаги: хвостовые вызовы



Функциональный симулятор: интерпретация

Все это время мы думали, как ускорить выполнение, когда мы исполняем по одной инструкции

Можем ли мы ускорить симуляцию, если интерпретировать не одну инструкцию, а сразу несколько?

Функциональный симулятор: трансляция

Все это время мы думали, как ускорить выполнение, когда мы исполняем по одной инструкции

Можем ли мы ускорить симуляцию, если интерпретировать не одну инструкцию, а сразу несколько?

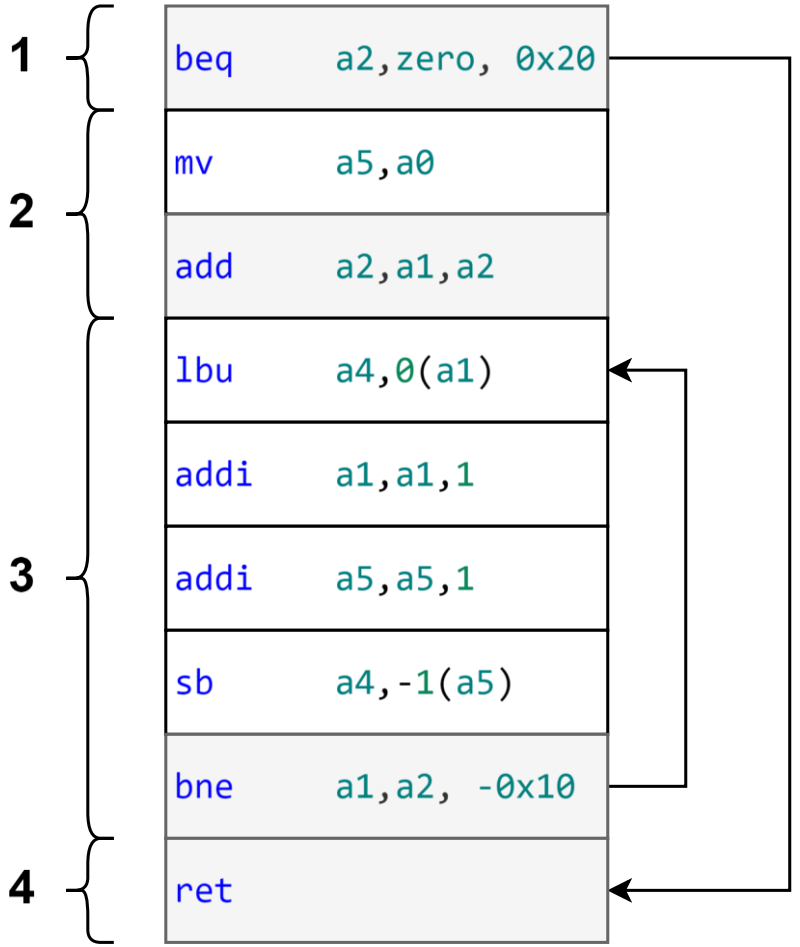
Да, можем с помощью бинарной трансляции

Трансляция

1. Разбиваем последовательность инструкций на *базовые блоки*

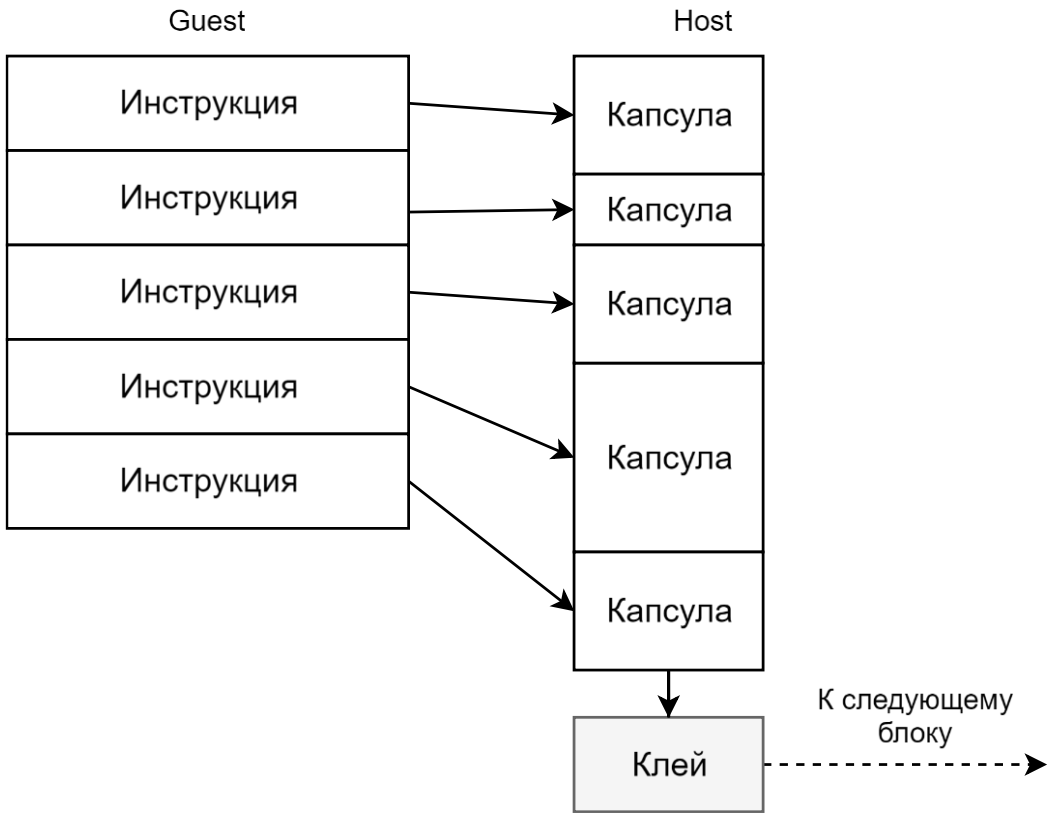
Базовый блок – последовательность инструкций или кода, имеющую одну точку входа и одну точку выхода

Базовые блоки заканчиваются инструкциями `branch`, `jump`, `ret`



Трансляция

2. Транслируем базовый блок гостевой архитектуры (guest) в базовый блок хозяйской архитектуры (host)



Трансляция

3. Прыгаем в транслированный код на хозяйской машине и исполняем базовый блок

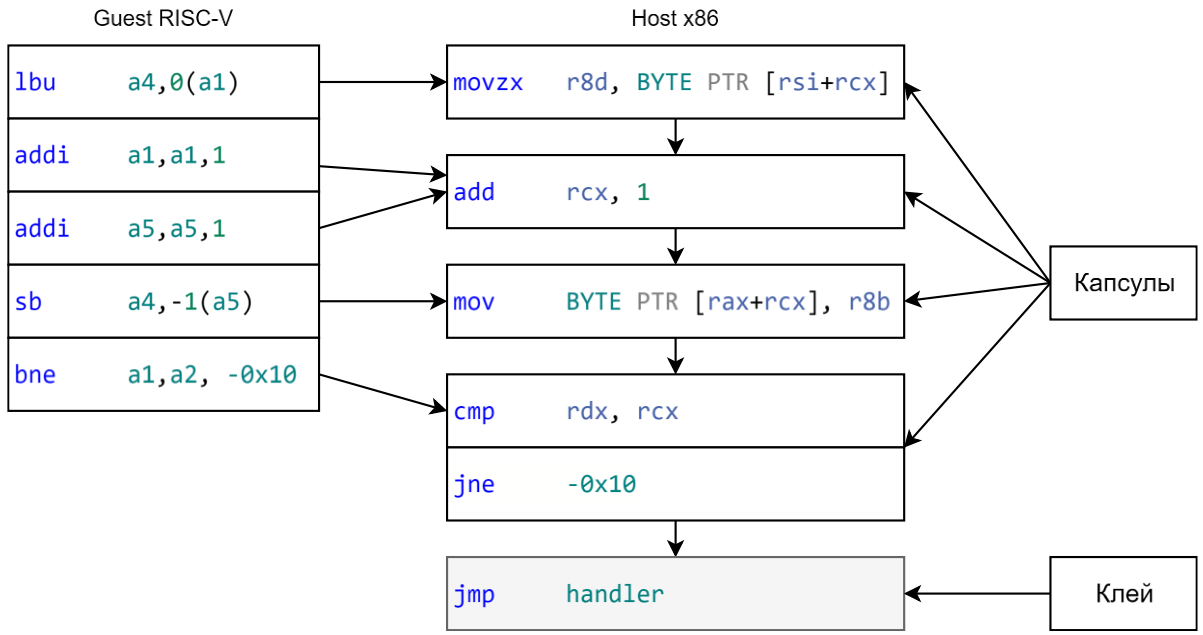
```
void *execute_bb(const void *bb,
                 size_t size) {
  void *code_mem = malloc(size);
  memcpy(code_mem, bb, size);
  allow_exec(code_mem, size);
  goto code_mem;
  return code_mem;
}
```

Оптимизация трансляции

Что еще на ваш взгляд можно добавить для повышения эффективности бинарной трансляции?

Оптимизация трансляции

- Оптимизация базовых блоков:
  - Пропуск nop инструкций
  - Объединение инструкций



Обсуждение: проблемы трансляции

Какие проблемы вы видите у бинарной трансляции?

Проблемы трансляции

Какие проблемы вы видите у бинарной трансляции?

- Самомодифицирующийся код
- Изменение адресов для jump и branch инструкций
- Ограниченность оптимизаций
- Системные вызовы невозможно транслировать

Интересный случай трансляции

- Представьте, что вы разрабатываете процессор KabyLake и у вас уже есть процессор SkyLake (7 и 6 поколения Intel Core соответственно)
- Очевидно, что большая часть ISA совпадает
- Можно ли это как-то использовать для ускорения трансляции?

Интересный случай трансляции

- Представьте, что вы разрабатываете процессор KabyLake и у вас уже есть процессор SkyLake (7 и 6 поколения Intel Core соответственно)
- Очевидно, что большая часть ISA совпадает
- Можно ли это как-то использовать для ускорения трансляции?
- Да, можно напрямую выполнять код, оптимизируя только горячие участки
- Такой подход называется прямое исполнение (*direct execution*)

Обсуждение: прямое исполнение

- Что нужно предусмотреть при прямом исполнении?

Обсуждение: прямое исполнение

Что нужно предусмотреть при прямом исполнении?

1. Неподдержанные инструкции
   - Такие инструкции придется интерпретировать/транслировать
2. Системные вызовы
   - Системные вызовы возможно только интерпретировать
3. Различное расположение внешних ресурсов (память, периферия)
   - Необходимо учитывать трансляцию адресов
4. Необходимо поддерживать изоляцию процесса
   - Программа не должна иметь возможность понять, что она запущена на другой архитектуре

To be continued …

На следующем занятии

- Узнаем что такое среда исполнения
- Как делать вызовы к среде исполнения
- Как процессор останавливается на breakpoint’е
- Как это нам поможет с `printf`
- Как так получилось, что для того, чтобы распечатать в консоль Hello красным и жирным шрифтом, нужно делать что-то странное: `printf("\033[31;1mHello\033[0m\n");`

Список литературы

- The RISC-V Instruction Set Manual Volume I Unprivileged Architecture Version 20240411
- Презентации курса "Основы программного моделирования", Е. Юлюгин: https://github.com/yulyugin/sim-lectures