Моделирование центрального процессора с помощью интерпретации

Курс «Программное моделирование вычислительных систем»

Григорий Речистов
grigory.rechistov@phystech.edu

24 февраля 2016 г.

На прошлой лекции

Требования к симуляторам:

- точность,
- скорость,
- совместимость/расширяемость

Вопросы

- В чём измеряется скорость симуляции?

Вопросы

- В чём измеряется скорость симуляции?
- Как соотносятся скорости функционального и потактового симуляторов?

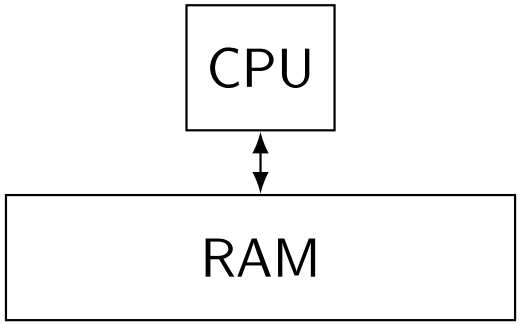
Вопросы

- В чём измеряется скорость симуляции?
- Как соотносятся скорости функционального и потактового симуляторов?
- Для чего необходимо предоставлять API симулятора?

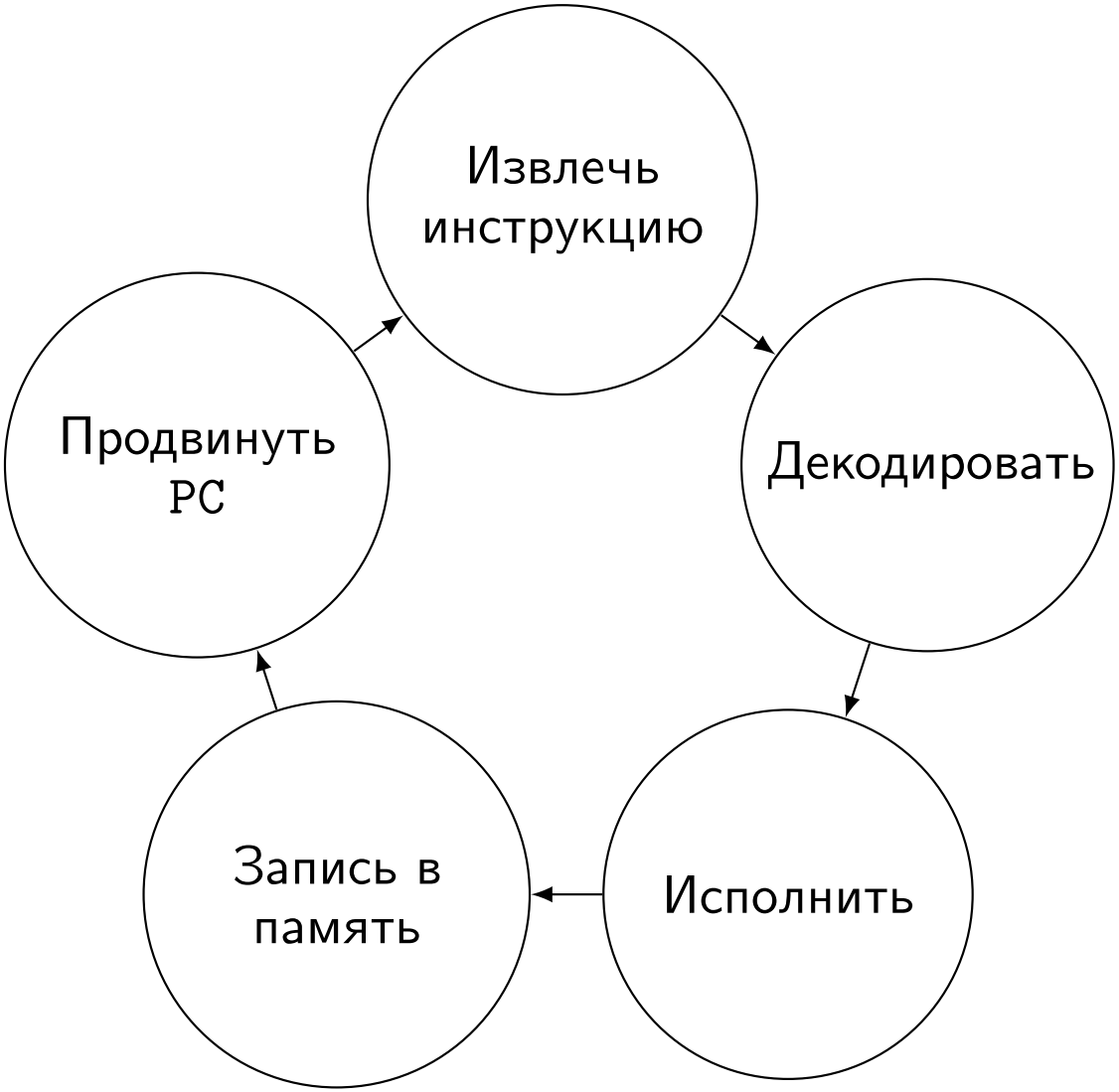
Вопросы

- В чём измеряется скорость симуляции?
- Как соотносятся скорости функционального и потактового симуляторов?
- Для чего необходимо предоставлять API симулятора? Чтобы пользователи могли с ним поиграть.

Моделируемая система



Конвейер процессора



Переключаемый интерпретатор (switched)

```
while (run) {
    raw_code = fetch(PC);
    (opcode, operands) = decode(raw_code);
    switch (opcode) {
    case opcode1:
        func1(operands); PC++; break;

    case opcode2:
        func2(operands); PC++; break;

    /*...*/
    }
}
```

Чтение инструкции из памяти

```
data = mem[pc];
```

Чтение инструкции из памяти

```
data = mem[pc];
```

Скорее,

```
data = read_mem(pc);
```

Чтение инструкции из памяти

```
data = mem[pc];
```

Скорее,

```
data = read_mem(pc);
```

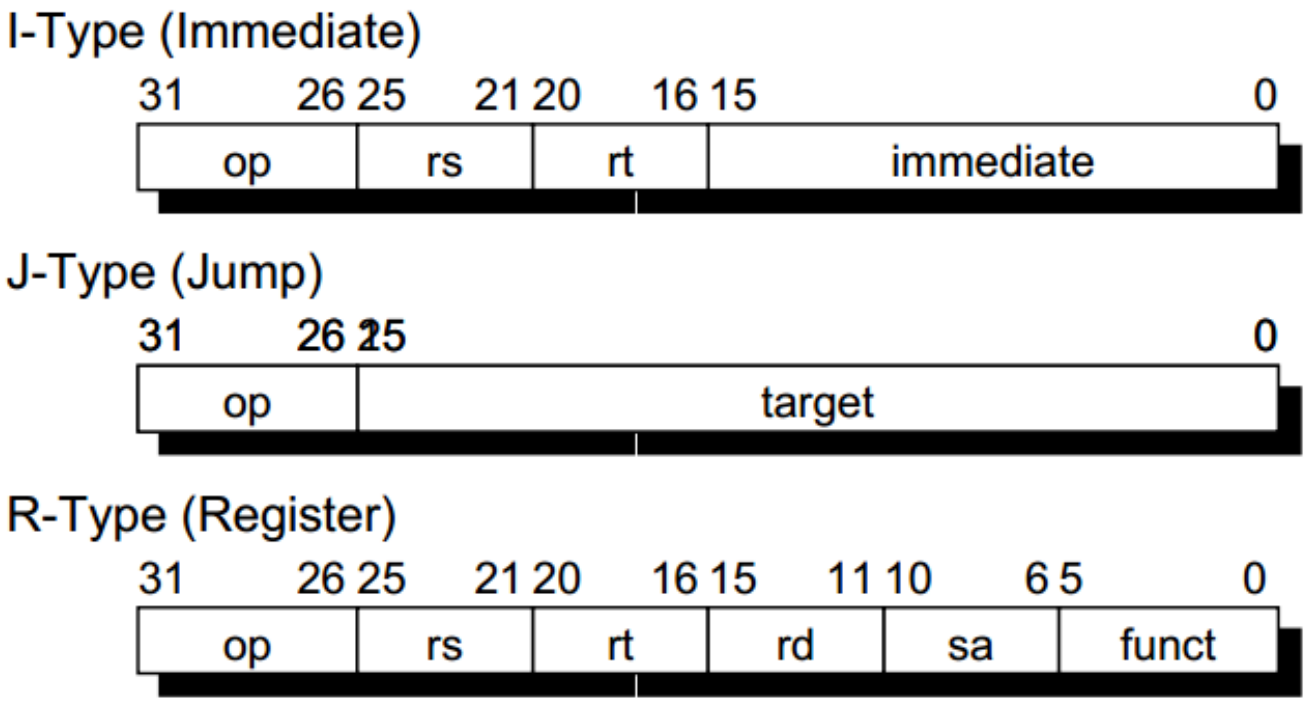
И не забыть про преобразование адресов:

```
paddr = v2p(pc); // pc - vaddr
data = mem[paddr];
```

Декодирование

Перевод данных об инструкции из машинного представления (последовательность байт) во внутреннее (высокоуровневое), удобное для последующего использования

Пример: MIPS



MIPS Technologies Inc. MIPS32 4K™ Processor Core Family Software User's Manual - 2002

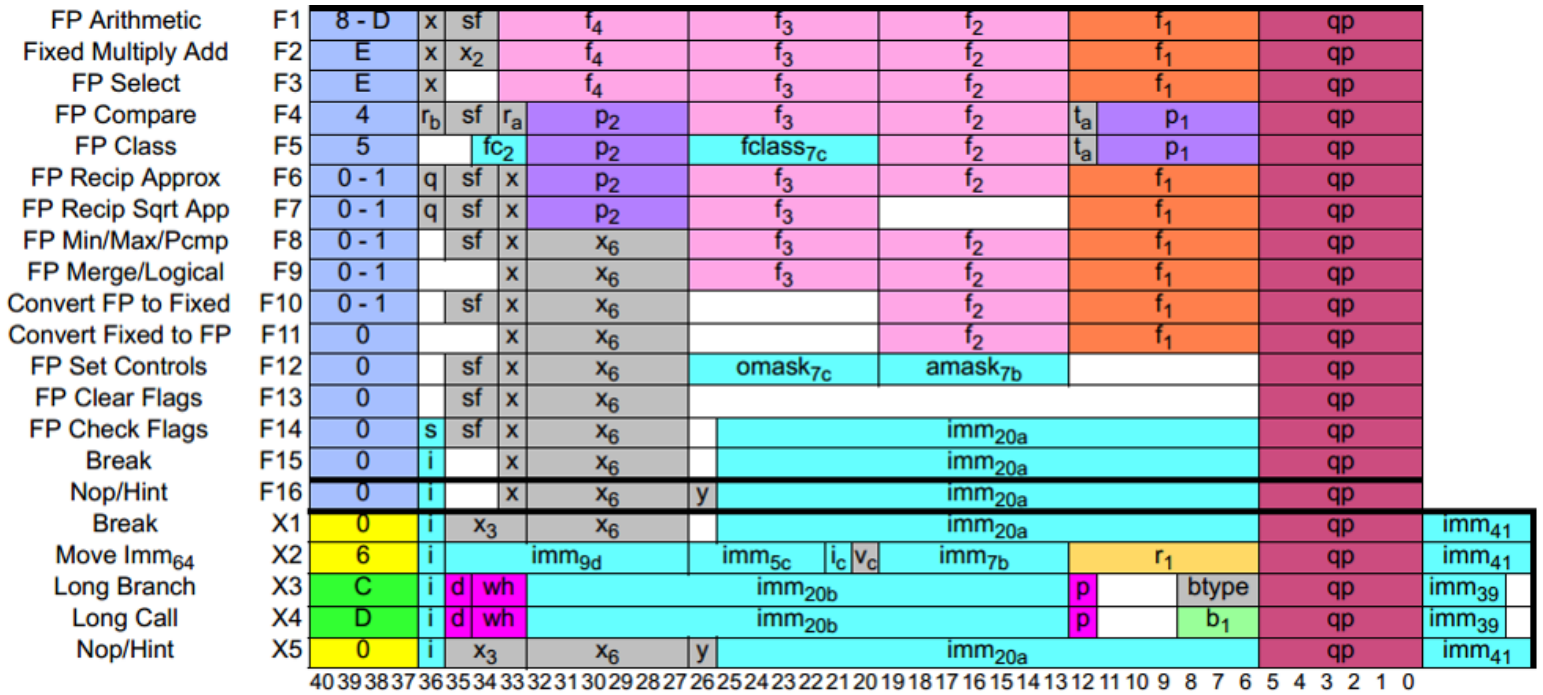
Пример: код 1

```
#define BITS(v, s, e) (v >> s) & (( 1 << (e-s+1)) - 1)

typedef struct decode {
    uint32_t op;
    uint32_t rs;
    uint32_t rt;
    int32_t  imm;
    int32_t  tgt;
    /* ... */
} decode_t;

static inline
int32_t sign_extend(uint32_t v, int width)
    {/* ... */};
```

Пример: код 2

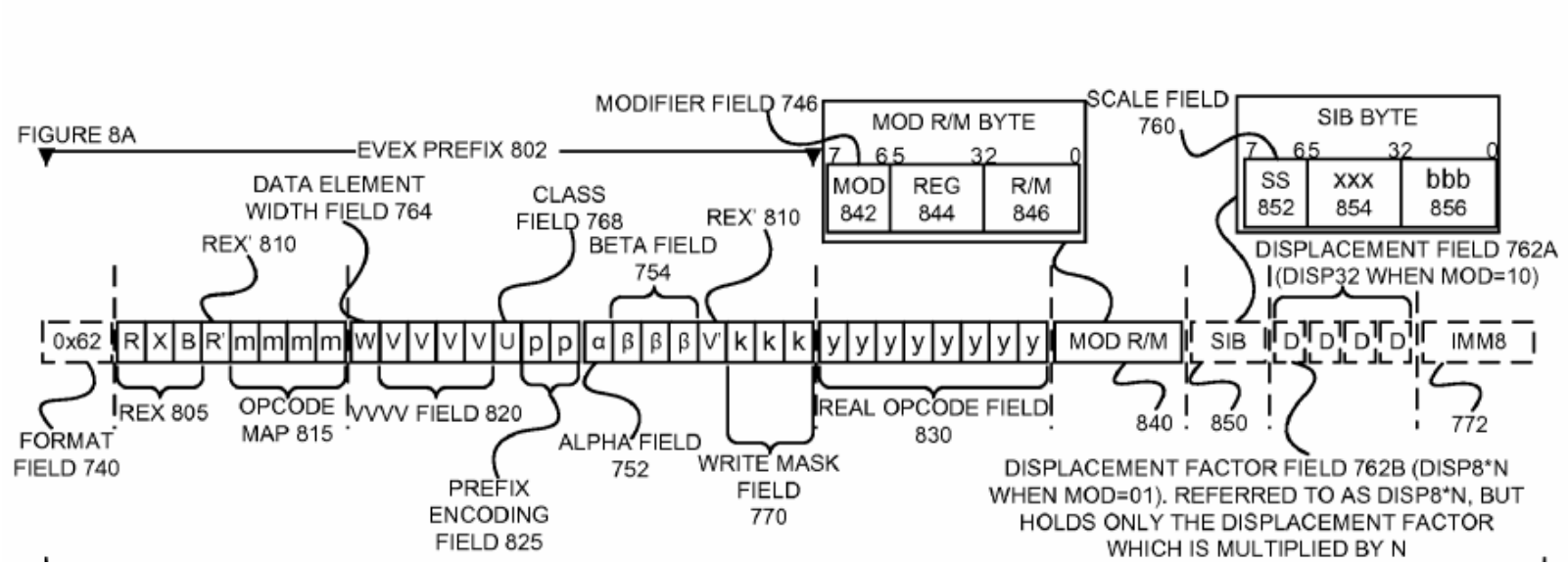
```
decode_t decode(uint32_t raw) {
    uint32_t op = BITS(raw, 26, 31);
    uint32_t rs = BITS(raw, 21, 25);
    uint32_t rt = BITS(raw, 16, 20);
    int32_t  imm = sign_extend(BITS(raw, 0, 15));
    int32_t  tgt = sign_extend(BITS(raw, 0, 25));
    /* ... */
    uint32_t funct = BITS(raw, 0, 5);

    return (decode_t){op, rs, rt, /* ... */, funct};
}
```

Более сложный пример: Intel IA-64 2.3



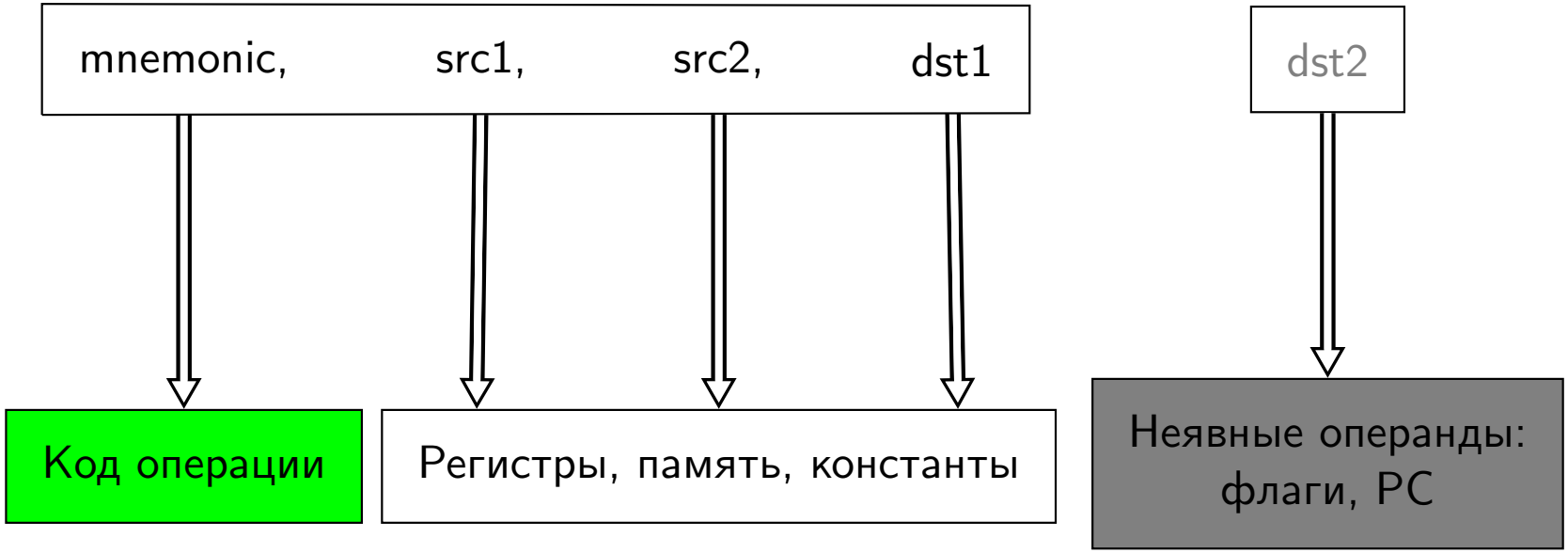
Intel Corporation. Intel® Itanium® Architecture Software Developer's Manual, p 3:296

Пример посложнее: Intel IA-32



J.C.S. Adrian et al. Systems, Apparatuses, and Methods for Blending Two Source Operands into a Single Destination Using a Writemask. US Patent Application Publication. № 2012/0254588 A1

Что извлекать из машинного кода инструкции



На выходе декодера:

- Успех, неуспех, недостаточно данных
- Для успеха: длина инструкции
- Для успеха: эмулирующая процедура

Декодирование

- Код декодера редко пишется вручную, чаще он генерируется по описанию
- A5 YX 0Z 00 $\Rightarrow$ MOD RX, RY, RZ
- В общем случае: классическая задача построения синтаксического анализатора
- На практике: специализированные инструменты и языки

Декодирование: суровая реальность

- Переменная длина инструкций. IA-32: от 8 до 120 бит. Сколько байт пытаться декодировать за один раз?
- Зависимость смысла от префикса, режима работы процессора. Пример: 0x40–0x4f в IA-32/Intel 64/AMD64
- Полное несоответствие какому-либо здравому смыслу

Дизассемблирование

- Дизассемблирование — перевод инструкций из машинного представление в понятный человеку вид (мнемонику)
- (За)кодирование (encoding) — перевод инструкций из мнемонической записи в машинный код

Дизассемблирование

- Дизассемблирование — перевод инструкций из машинного представление в понятный человеку вид (мнемонику)
- (За)кодирование (encoding) — перевод инструкций из мнемонической записи в машинный код

Вопрос: однозначны ли операции: декодирования, дизассемблирования, кодирования?

Исполнение

- Базовая единица — функция-эмулятор одной инструкции (service routine)
- Функция-эмуляторы пишутся на языке высокого уровня → переносимость кода между хозяйскими платформами, компиляторами

Симулируемое состояние

```
typedef struct {
    uint32_t regs[16];
    bool z_flag;
    bool n_flag;
    bool o_flag;
    bool c_flag;
    uint32_t pc;
} cpu_t;
```

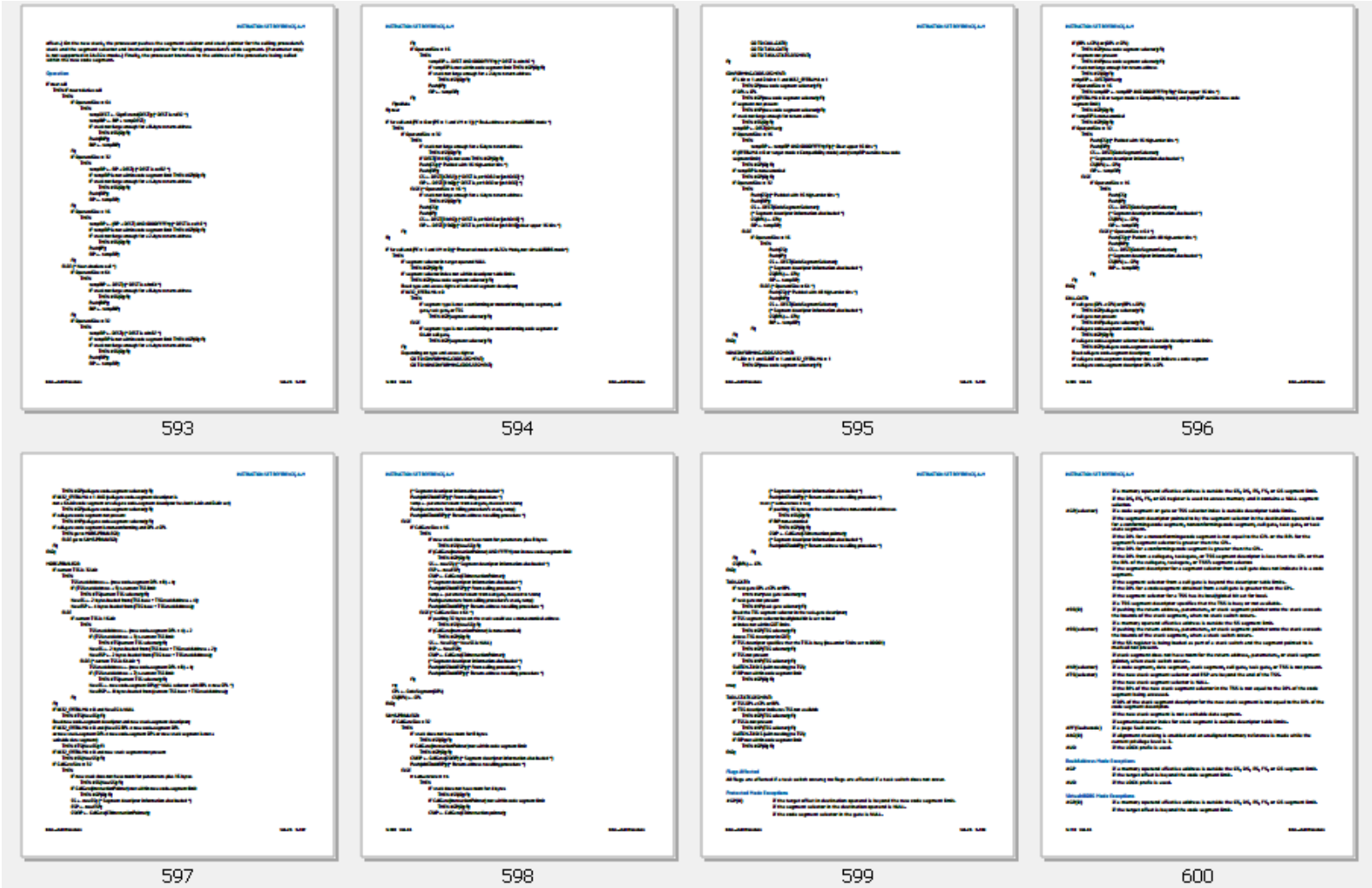
Пример: ADD reg reg reg

```
void add32_rrr(cpu_t *cpu, int src1, int src2, int dst) {
    cpu->regs[dst] = cpu->regs[src1] + cpu->regs[src2];
```

Пример: ADD reg reg reg

```
void add32_rrr(cpu_t *cpu, int src1, int src2, int dst) {
    cpu->regs[dst] = cpu->regs[src1] + cpu->regs[src2];

    cpu->z_flag = cpu->regs[dst] == 0;
    cpu->n_flag = cpu->regs[dst] & (1 << 31);
    cpu->o_flag = cpu->regs[dst] <
            MAX(cpu->regs[src1], cpu->regs[src2]);
    cpu->c_flag = calc_c_flag(cpu->regs[src1],
                              cpu->regs[src2]);
}
```

Пример посложнее: IA-32 CALL



Запись результата в память

```
write_mem(cpu, dst_addr, data, size);
```

Запись результата в память

```
write_mem(cpu, dst_addr, data, size);
```

- Невыровненный адрес
- Граница страниц
- Запись в память «только для чтения»
- Исключения для всей или части записи/чтения

Уточнённый цикл работы процессора

Классификация

- Interruptions (термин из документации IA-64) — вмешательство, перерыв, приостановка
- Exception — синхронное исключение, без повторения текущей инструкции
- Fault — синхронное, с повторением текущей инструкции
- Trap — синхронное, без повторения, намеренно вызванное
- Interrupt — внешнее асинхронное прерывание
- Abort — внешние асинхронное с отсутствием информации о точке возврата

Продвижение PC

- Для большинства команд: увеличение счетчика на длину обработанной инструкции

```
cpu->pc += instr_length;
```

Продвижение PC

- Для большинства команд: увеличение счетчика на длину обработанной инструкции
  `cpu->pc += instr_length;`
- Бывают исключения: `REP MOVS` в IA-32

Продвижение PC

- Для большинства команд: увеличение счетчика на длину обработанной инструкции
  `cpu->pc += instr_length;`
- Бывают исключения: `REP MOVS` в IA-32
- Явное изменение PC — команды управления исполнением:
  - (Без)условный (не)прямой прыжок/переход
  - Вызов/возврат из процедуры
  - Системный вызов

Преимущества и недостатки интерпретации

- Пишется на языках высокого уровня: код переносим
- Простая структура: надёжность, расширяемость, переиспользование

Преимущества и недостатки интерпретации

- Пишется на языках высокого уровня: код переносим
- Простая структура: надёжность, расширяемость, переиспользование
- (Очень) низкая скорость работы

Куда тратится время?

```
start: interruption = false;
while (!interruption) {
    raw_code = fetch(PC);
    (opcode, operands) = decode(raw_code); // <-- здесь
    switch (opcode) { // <-- здесь
    case opcode1:
        func1(operands); PC++; break;
    case opcode2:
        func2(operands); PC++; break;
    /*...*/
    }
}
handle_interruption();
goto start;
```

Шитая интерпретация (threaded interpretation)

Вместо возвращения к началу цикла «прыгаем» прямо на исполнение следующей инструкции

```
func0: /* simulate instr0 */; PC++;
  next_opcode = decode(fetch(PC));
  goto func_ptr[next_opcode];
func1: /* simulate instr1 */; PC++;
  next_opcode = decode(fetch(PC));
  goto func_ptr[next_opcode];
func2: /* simulate instr2 */; PC++;
  next_opcode = decode(fetch(PC));
  goto func_ptr[next_opcode];
```

http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array

Кэширующая интерпретация

- В большинстве случаев в код гостевого приложения неизменен
- Велика вероятность того, что инструкции с некоторыми PC будут исполнены много раз (*задача*)
- Зачем каждый раз их декодировать?
- Заводим таблицу соответствия «адрес инструкции $\rightarrow$ декодированный результат»

Кэширующая интерпретация

```
while (!interruption) {
  if (operation = cache[PC]); // короткий путь
  else { // не в кэше, длинный путь
   operation = decode(fetch(PC));
   cache[PC] = operation; // на будущее
  }
  switch (operation) {
     /* ... */
  }
}
```

Кэширующая интерпретация

- Ёмкость любого кэша ограничена, старые данные надо выбрасывать
- Необходимо следить за неизменностью исходного кода, иначе сохранённое соответствие будет неверным

Итоги

- Фазы исполнения: F, D, E, W, A
- Decoder, disassembler, encoder
- Переключаемый (switched) И.
- Шитый (threaded) И.
- Кэширующий И.
- Ситуации: interrupt, trap, exception, fault, abort

Литература I

- Семь видов интерпретаторов виртуальной машины. В поисках самого быстрого http://habrahabr.ru/company/intel/blog/261665/
- D. Mihoka , S. Shwartsman. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure http://bochs.sourceforge.net/
- Fredrik Larsson, Peter Magnusson, Bengt Werner. SimGen: Development of Efficient Instruction Set Simulators ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-R--97-03--SE.ps.Z

Литература II

- Yair Lifshitz, Robert Cohn, Inbal Livni, Omer Tabach, Mark Charney, Kim Hazelwood. Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration
  http://www.cs.virginia.edu/kim/docs/wish11zsim.pdf
- Префиксы в системе команд IA-32
  http://habrahabr.ru/company/intel/blog/200598/
- Программная симуляция микропроцессора. Коробка передач
  http://habrahabr.ru/company/intel/blog/202926/

На следующей лекции

Моделирование архитектурного состояния

Спасибо за внимание!

Слайды и материалы курса доступны по адресу
http://is.gd/ivuboc

Замечание: все торговые марки и логотипы, использованные в данном материале, являются собственностью их владельцев. Представленная здесь точка зрения отражает личное мнение автора, не выступающего от лица какой-либо организации.