Архитектура и микроархитектура. Однотактный RISC-V

Уровни абстракции

Уровень	Пример
Прикладные программы	Текстовый редактор
Системное ПО	ОС, драйверы
Архитектура набора команд	Инструкции, регистры
Микроархитектура	АЛУ, конвейер
Логика	Сумматоры, сдвиговые регистры
Цифровая электроника	Элементы AND, XOR, триггеры
Аналоговая электроника	Усилители, генераторы колебаний
Полупроводниковые устройства	Полевые и биполярные транзисторы

ISA и микроархитектура

Архитектура набора команд (instruction set architecture, ISA) — часть архитектуры, определяющая интерфейс микропроцессора, видимый программисту.

Микроархитектура — организация логических блоков, определяющих реализацию ISA в микропроцессоре.

Одна ISA может быть реализована в микроархитектурах с разным соотношением цены, производительности и потребляемой энергии.

Пример: x86 ISA

- Intel 80486
- Intel Core i7
- AMD Ryzen

ISA

- Инструкции
  - Могут использовать данные из регистров, памяти, а также констант из тела инструкции
  - Двоичные машинные коды (команды), а также их мнемонические представления, которые говорят процессору, что нужно делать
- Регистры
  - Самая быстрая память, но вмещает мало данных (десятки байт)
  - Адресуются по имени или порядковому номеру
  - Не путать с массивами триггеров
- Память
  - Много места для данных, но доступ к ней занимает больше времени
  - Побайтовая или пословная адресация

```
4d 8b 41 04             mov    r8, QWORD PTR [r9+0x4]
```

RISC и CISC архитектуры

RISC

Малый набор инструкций, которые выполняют простые операции.

Примеры:

- ARM
  - Микроконтроллеры STM32
  - Мобильные телефоны
- MIPS
  - PlayStation 2
- RISC-V
  - SiFive FE310

CISC

Большой набор инструкций различной сложности, которые могут кодировать по нескольку операций.

Примеры:

- PDP-11
- x86
  - Большинство ПК

RISC-V

RISC-V — открытая архитектура RISC-микропроцессоров с большим выбором расширений

- Спецификация:
  - The RISC-V Instruction Set Manual Volume I: User-Level ISA
  - The RISC-V Instruction Set Manual Volume II: Privileged Architecture

- RV32I — базовый вариант для работы с целыми числами
  - Расширение M — умножение и деление
  - Расширение A — атомарные операции
  - Расширение F — числа с плавающей точкой
  - Расширение D — числа двойной точности с плавающей точкой
  - etc.
- RV32E — модификация RV32I для встраиваемых (embedded) платформ
- RV64I
- RV128I

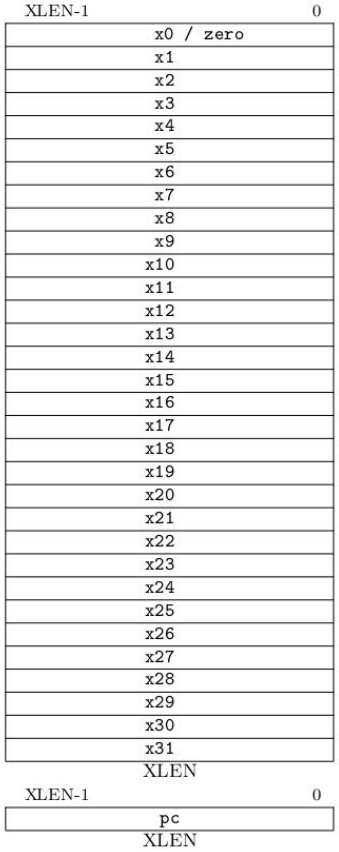
Обзор RV32I

Инструкции:

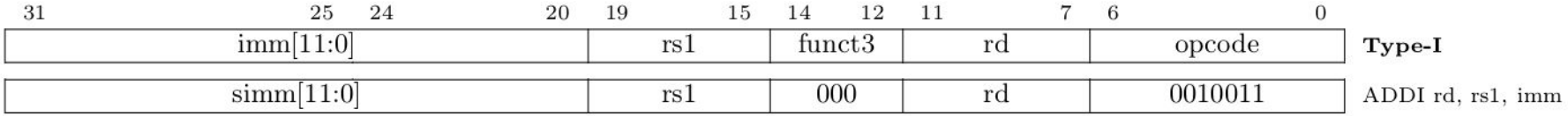
- 47 (38) инструкций длины 32 бита
  - Арифметические операции
  - Доступ к памяти
  - Условные и безусловные переходы
  - Системные инструкции
- 6 форматов: R, I, S, U, B, J

Регистры:

- 31 регистр общего назначения `x1`–`x31` для 32-битных целых чисел
- 32-битный константный нулевой регистр `x0`
- 32-битный счетчик команд (program counter - pc)



Инструкция ADDI



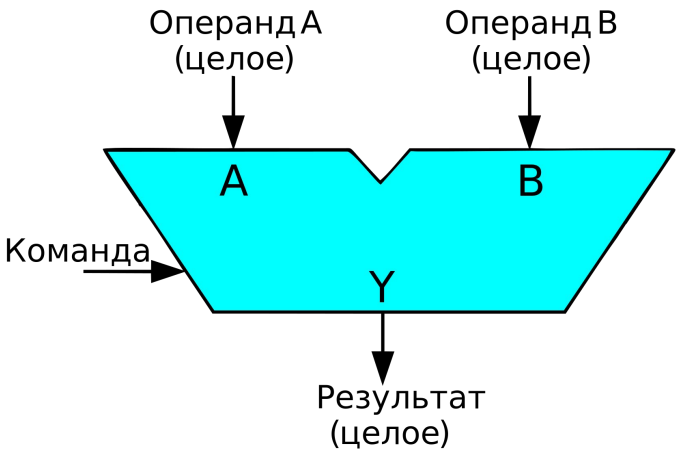
ADDI прибавляет знаковое 12-битное число (расширенное до 32 бит) к регистру rs1 и сохраняет в rd. Переполнение игнорируется. ADDI используется для реализации псевдо-инструкций MV, LI, NOP.

- Инструкция определяется полями `funct3` и `opcode`
- Поле `imm` содержит непосредственное значение операнда
- Поля `rs1` и `rd` содержат номера регистра-источника и регистра-назначения соответственно

Машинный код (hex)	Код на языке ассемблера RISC-V	Псевдокод
00b00293	addi x5, x0, 11	x5 <- 11
00928393	addi x7, x5, 9	x7 <- x5 + 9
00000013	addi x0, x0, 0	nop
00038593	addi x11, x7, 0	x11 <- x7
ffb38393	addi x7, x7, -5	x7 <- x7 - 5

АЛУ

Арифметико-логическое устройство — схема, производящая арифметические и логические над целыми числами




```
module alu(
    input [31:0]src_a,
    input [31:0]src_b,
    input op,
    output reg [31:0]res
);

always @(*) begin
    case (op)
        1'b0: res = src_a;
        1'b1: res = src_a + src_b;
    endcase
end

endmodule
```

Однотактное устройство управления

```
module control(
    input [31:0]instr,

    output reg [11:0]imm12,
    output reg rf_we,
    output reg alu_op
);

wire [6:0]opcode = instr[6:0];
wire [2:0]funct3 = instr[14:12];

always @(*) begin
    ...........
end

endmodule
```

```
always @(*) begin
    rf_we = 1'b0;
    alu_op = 1'b0;
    imm12 = 12'b0;

    casez ({funct3, opcode})
        10'b000_0010011: begin // ADDI
            rf_we = 1'b1;
            alu_op = 1'b1;
            imm12 = instr[31:20];
        end
        default: ;
    endcase
end
```

Однотактное ядро

```
module core(
    input clk,
    input [31:0]instr,
    input [31:0]last_pc,

    output [31:0]instr_addr
);

reg [31:0]pc = 0;
assign instr_addr = pc;

always @(posedge clk) begin
    pc <= (pc == last_pc) ? pc : pc + 1;
    $display("[%h] %h", pc, instr);
end

wire [4:0]rd = instr[11:7];
wire [4:0]rs1 = instr[19:15];

wire [31:0]alu_result;
wire [31:0]alu_b_src = imm32;
alu alu(
    .src_a(rf_rdata0), .src_b(alu_b_src), .op(alu_op),
    .res(alu_result)
);

wire [31:0]rf_rdata0;
wire [4:0]rf_raddr0 = rs1;
wire [31:0]rf_wdata = alu_result;
wire [4:0]rf_waddr = rd;
wire rf_we;
reg_file rf(
    .clk(clk),
    .raddr0(rf_raddr0), .rdata0(rf_rdata0),
    .raddr1(5'b0),
    .waddr(rf_waddr), .wdata(rf_wdata), .we(rf_we)
);

wire [31:0]imm32 = {{20{imm12[11]}}, imm12};

wire [11:0]imm12;
control control(
    .instr(instr),
    .imm12(imm12),
    .rf_we(rf_we),
    .alu_op(alu_op)
);

endmodule
```

Тестирование

```
module cpu_top(
    input clk
);

wire [31:0]instr_addr;
wire [31:0]instr_data;
rom rom(.addr(instr_addr), .q(instr_data));

core core(
    .clk(clk),
    .instr_data(instr_data), .last_pc(32'h4),
    .instr_addr(instr_addr)
);

endmodule
```

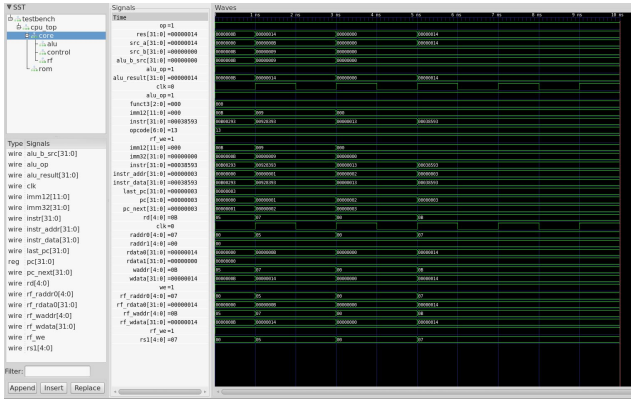
```
module testbench;

reg clk = 1'b0;

always
    #1 clk = ~clk;

cpu_top cpu_top(.clk(clk));

initial begin
    $dumpvars;
    #10 $finish;
end

endmodule
```

Результат

```
[00000000] 00b00293
x0: 00000000 x4: xxxxxxxx  x8: xxxxxxxx x12: xxxxxxxx
x1: xxxxxxxx x5: 0000000b  x9: xxxxxxxx x13: xxxxxxxx
x2: xxxxxxxx x6: xxxxxxxx x10: xxxxxxxx x14: xxxxxxxx
x3: xxxxxxxx x7: xxxxxxxx x11: xxxxxxxx x15: xxxxxxxx
[00000001] 00928393
x0: 00000000 x4: xxxxxxxx  x8: xxxxxxxx x12: xxxxxxxx
x1: xxxxxxxx x5: 0000000b  x9: xxxxxxxx x13: xxxxxxxx
x2: xxxxxxxx x6: xxxxxxxx x10: xxxxxxxx x14: xxxxxxxx
x3: xxxxxxxx x7: 00000014 x11: xxxxxxxx x15: xxxxxxxx
[00000002] 00000013
x0: 00000000 x4: xxxxxxxx  x8: xxxxxxxx x12: xxxxxxxx
x1: xxxxxxxx x5: 0000000b  x9: xxxxxxxx x13: xxxxxxxx
x2: xxxxxxxx x6: xxxxxxxx x10: xxxxxxxx x14: xxxxxxxx
x3: xxxxxxxx x7: 00000014 x11: xxxxxxxx x15: xxxxxxxx
[00000003] 00038593
x0: 00000000 x4: xxxxxxxx  x8: xxxxxxxx x12: xxxxxxxx
x1: xxxxxxxx x5: 0000000b  x9: xxxxxxxx x13: xxxxxxxx
x2: xxxxxxxx x6: xxxxxxxx x10: xxxxxxxx x14: xxxxxxxx
x3: xxxxxxxx x7: 00000014 x11: 00000014 x15: xxxxxxxx
[00000004] ffb38393
x0: 00000000 x4: xxxxxxxx  x8: xxxxxxxx x12: xxxxxxxx
x1: xxxxxxxx x5: 0000000b  x9: xxxxxxxx x13: xxxxxxxx
x2: xxxxxxxx x6: xxxxxxxx x10: xxxxxxxx x14: xxxxxxxx
x3: xxxxxxxx x7: 0000000f x11: 00000014 x15: xxxxxxxx
```

program.txt	program.s

```
00b00293    addi x5,  x0,  11
00928393    addi x7,  x5,   9
00000013    addi x0,  x0,   0
00038593    addi x11, x7,   0
ffb38393    addi x7,  x7,  -5
```



GitHub

github.com/viktor-prutyanov/drec-fpga-intro