

Мастер-класс

**“Разработка однотактного процессора архитектуры
RISC-V”**

Структура занятия

- **Теоретическая часть №1 (30 минут)**

Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog

- **Практическая часть №1 (45 минут)**

Описание комбинационной и последовательностной логики на языке Verilog

- **Теоретическая часть №2 (30 минут)**

Введение в архитектуру RISC-V.

- **Практическая часть №2 (45 минут)**

Использование assembler RISC-V

- **Теоретическая часть №3 (30 минут)**

Введение в микроархитектуру RISC-V.

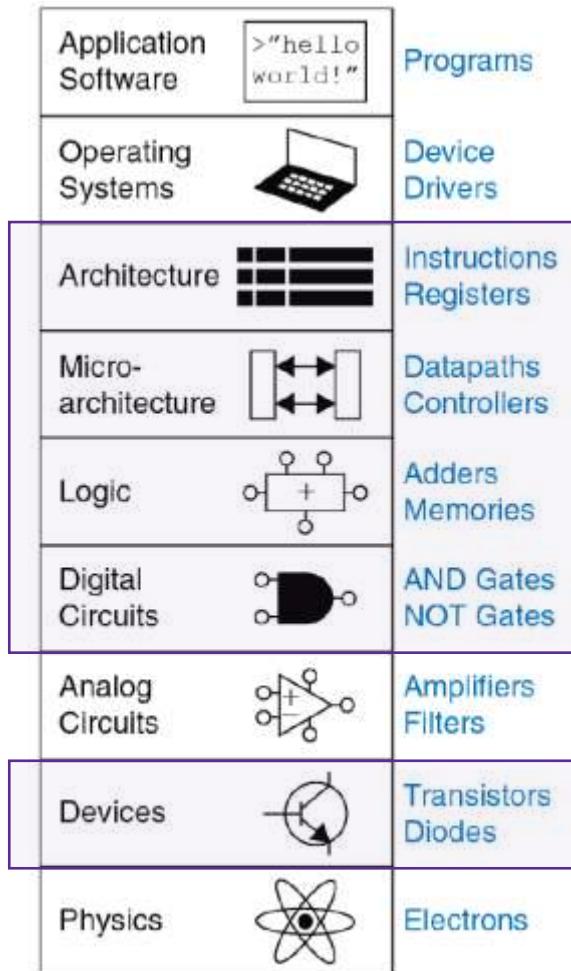
- **Практическая часть №3 (45 минут)**

Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog

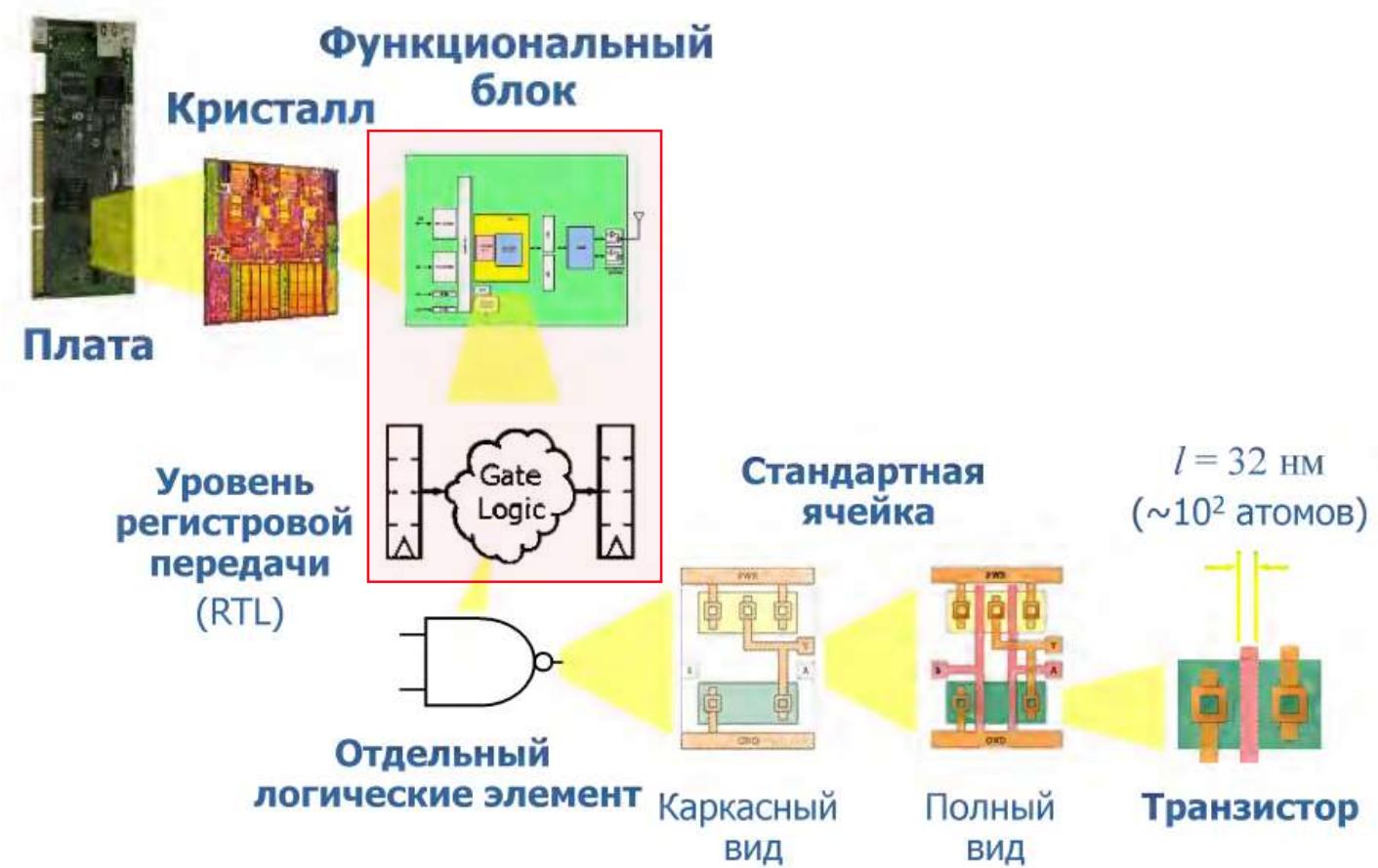
- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- Описание комбинационной и последовательностной логики на языке Verilog
- Введение в архитектуру RISC-V.
- Использование assembler RISC-V
- Введение в микроархитектуру RISC-V.
- Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Уровни абстракции в микроэлектронике

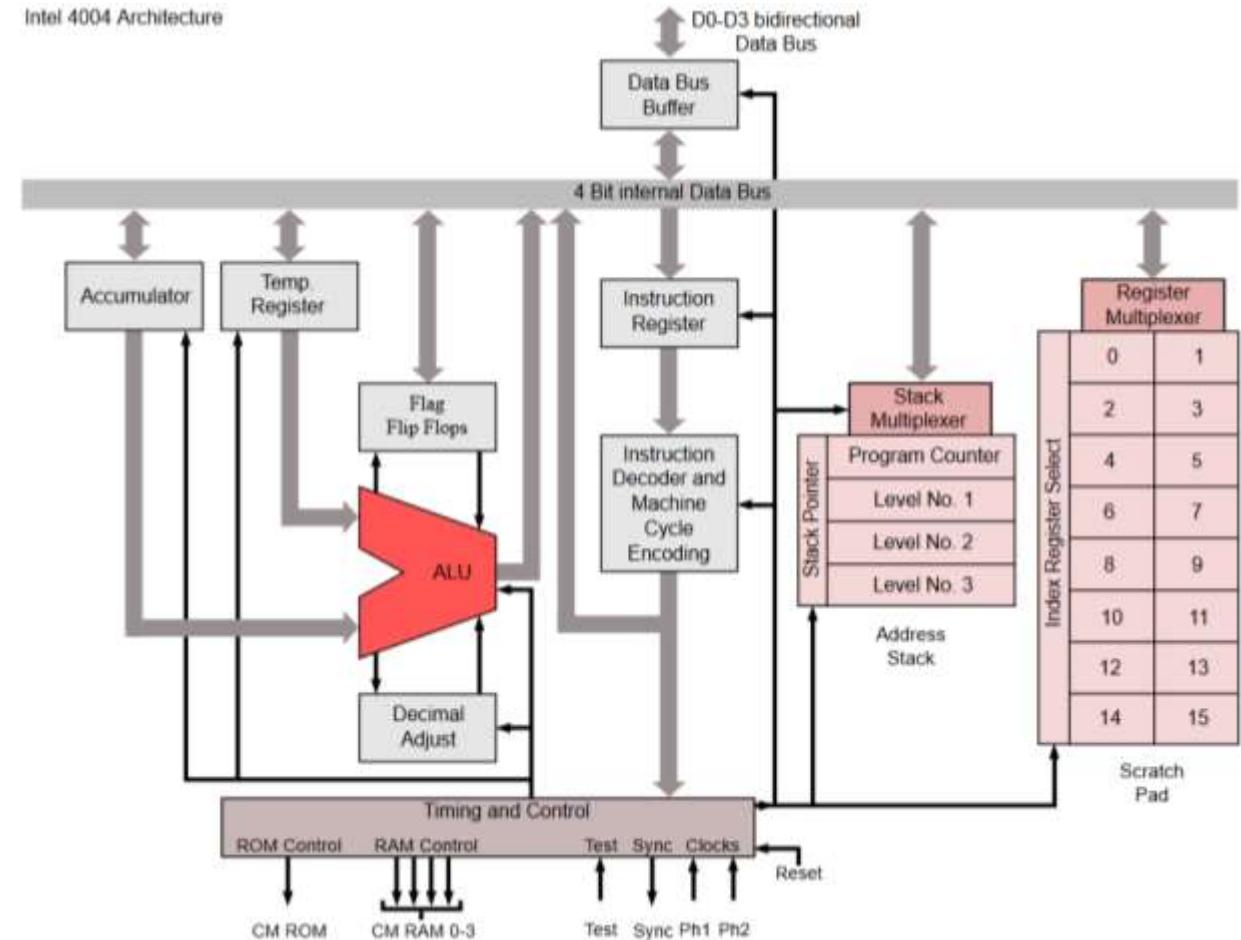
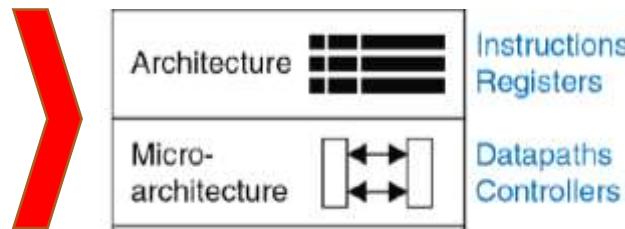


Уровни абстракции в микроэлектронике

Application Software	>"hello world!"
Operating Systems	Device Drivers
Architecture	Instructions Registers
Micro-architecture	Datapaths Controllers
Logic	Adders Memories
Digital Circuits	AND Gates NOT Gates
Analog Circuits	Amplifiers Filters
Devices	Transistors Diodes
Physics	Electrons

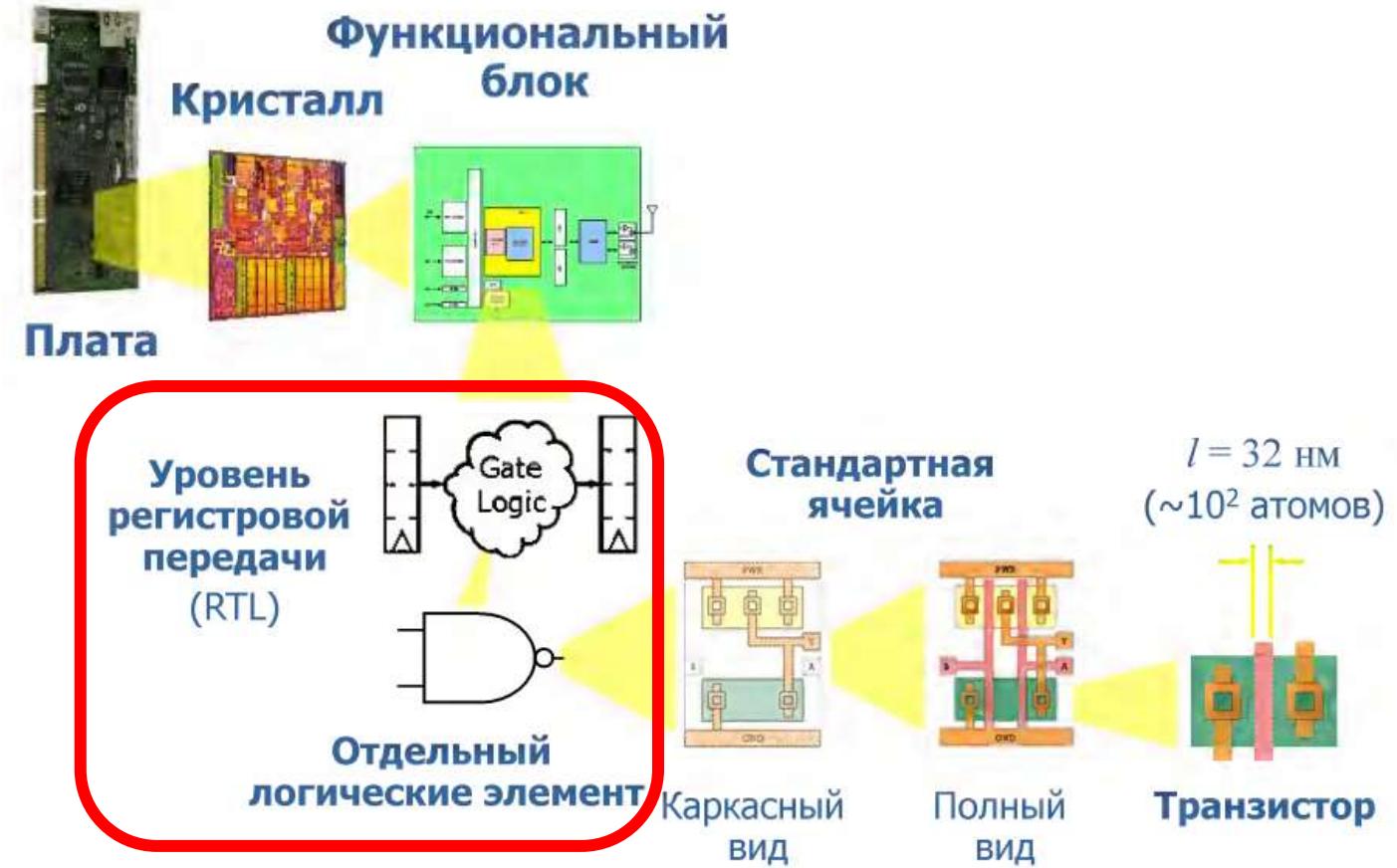
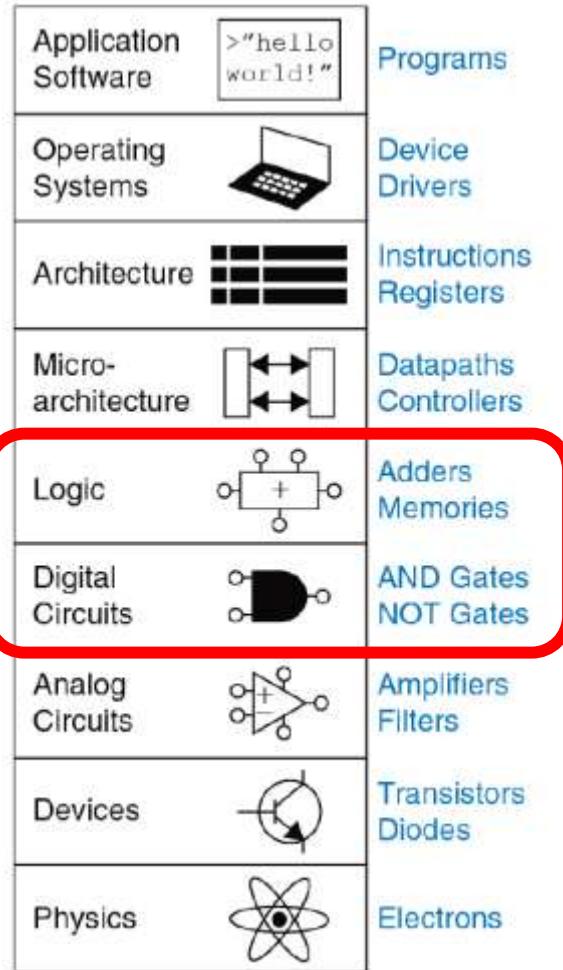


Уровни абстракции в микроэлектронике



Архитектура процессорного ядра intel 4004 (1971)

Уровни абстракции в микроэлектронике

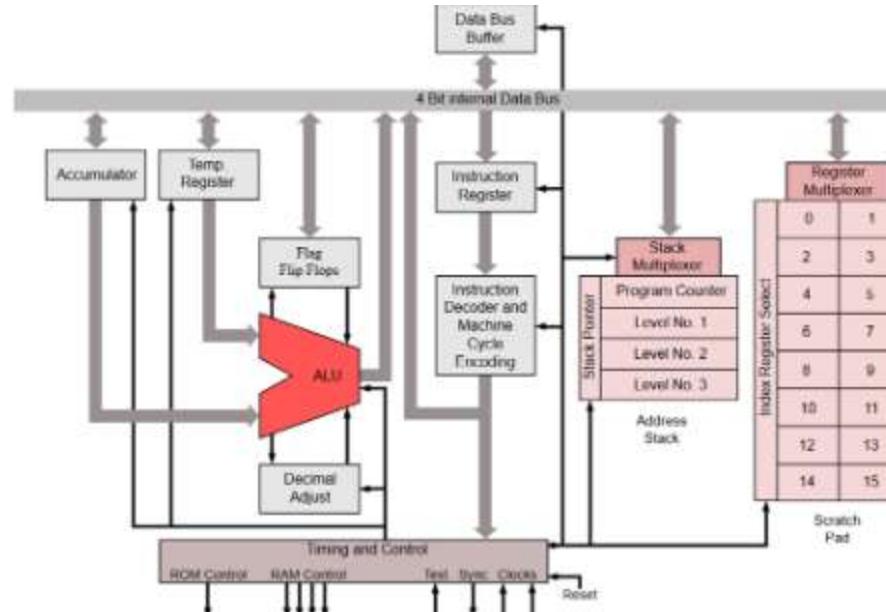


Уровень регистровых передач (RTL)

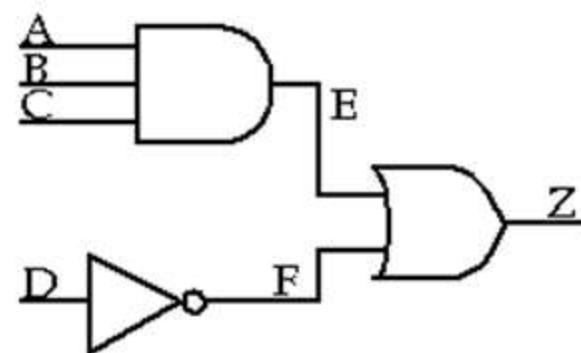
Уровень описания
регистровых передач
(RTL Design)



Логическая схема
(Logic Design)

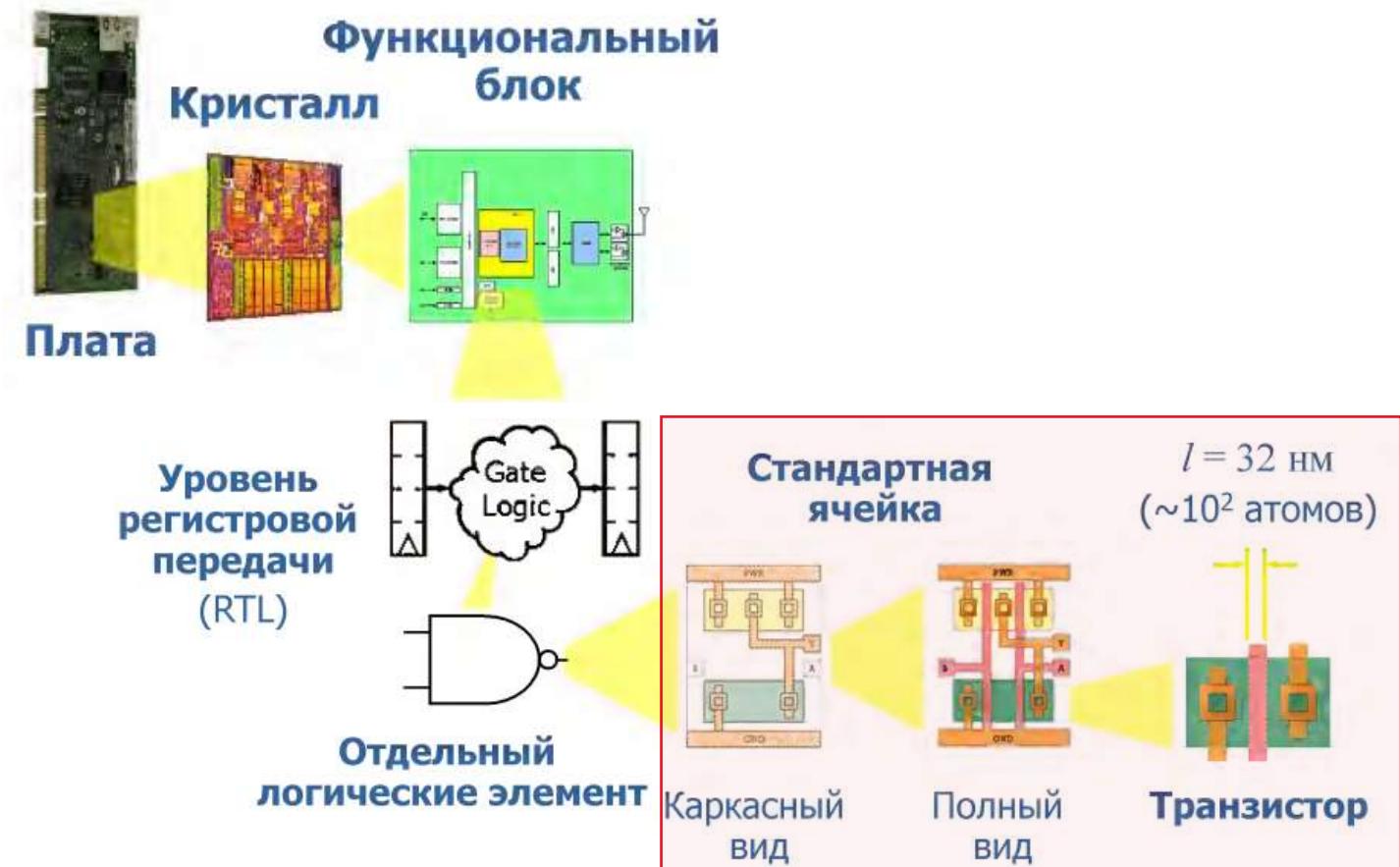


Архитектура процессорного ядра intel 4004 (1971)

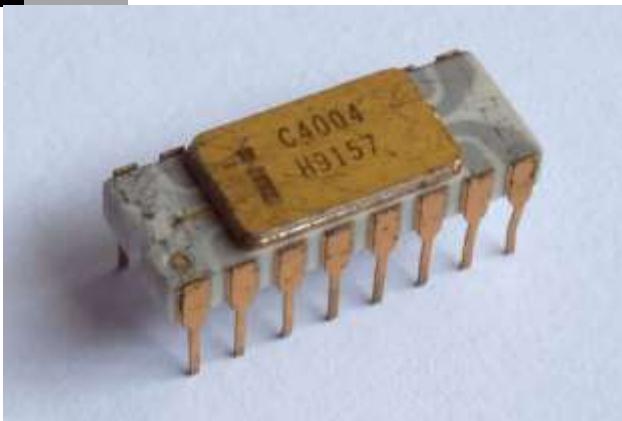


Уровни абстракции в микроэлектронике

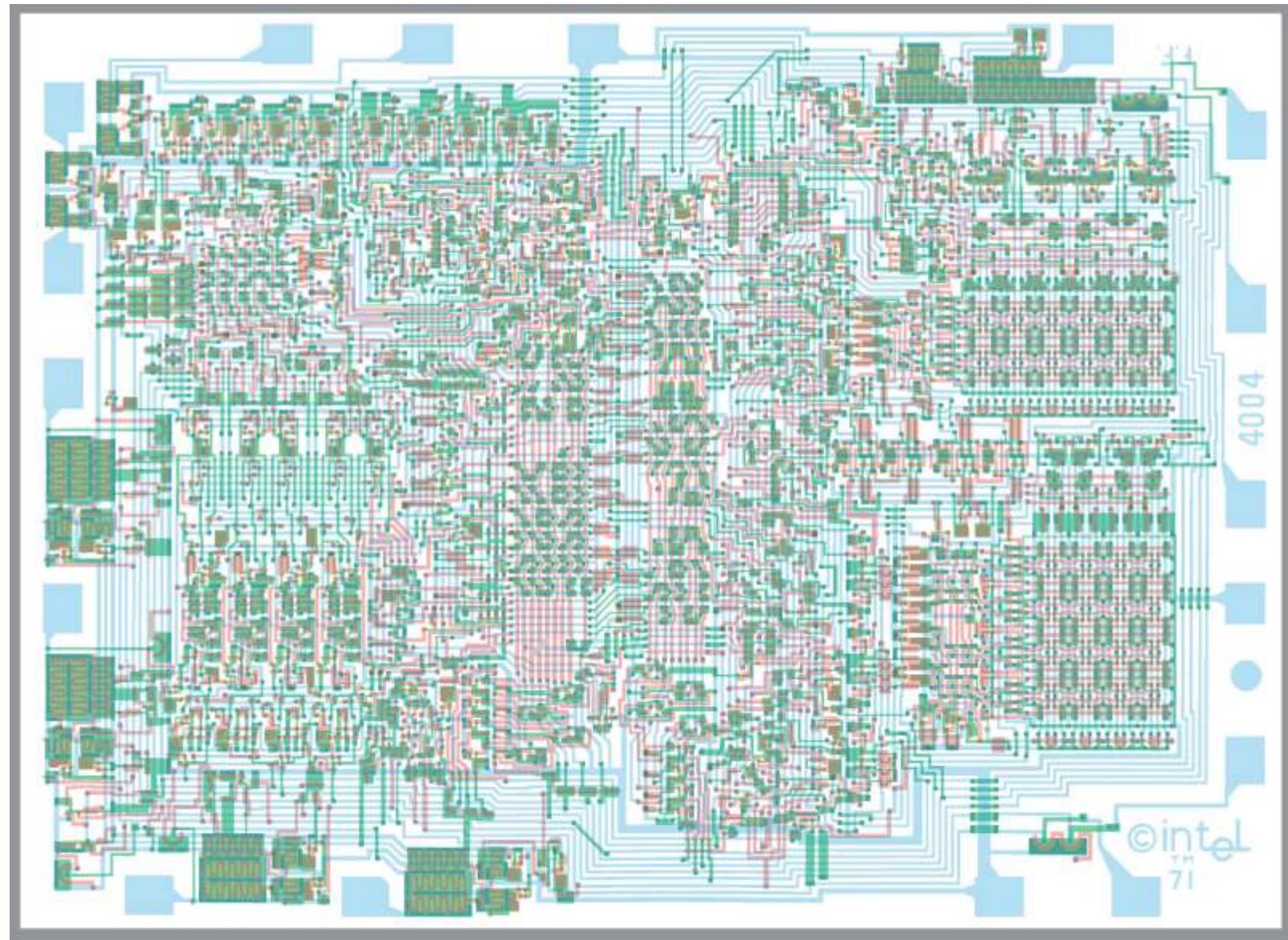
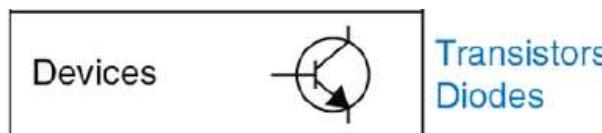
Application Software	>"hello world!"
Operating Systems	Device Drivers
Architecture	Instructions Registers
Micro-architecture	Datapaths Controllers
Logic	Adders Memories
Digital Circuits	AND Gates NOT Gates
Analog Circuits	Amplifiers Filters
Devices	Transistors Diodes
Physics	Electrons



Intel 4004



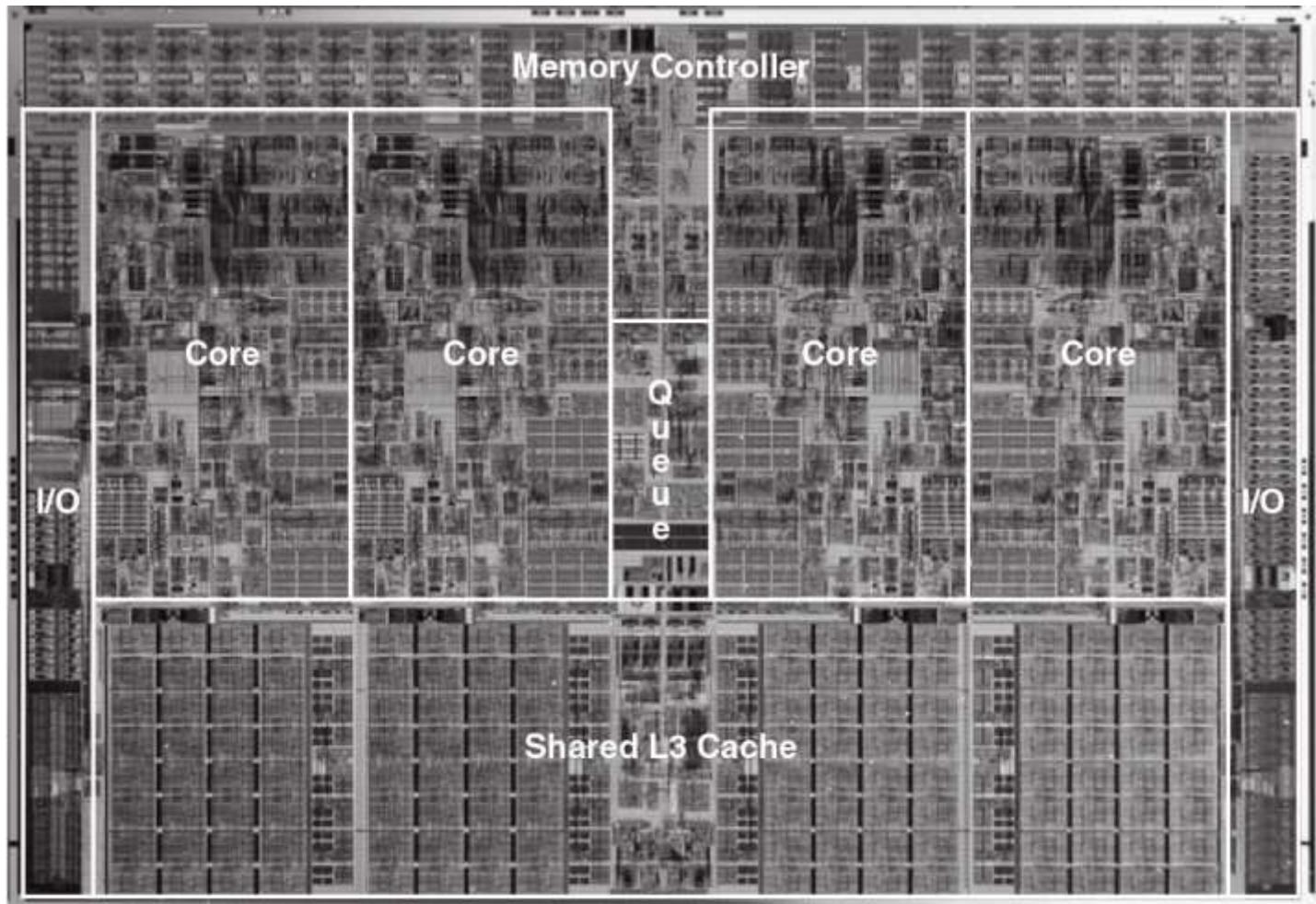
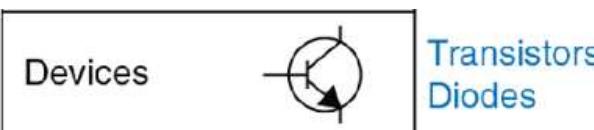
- Дата анонса: 15 ноября 1971 года
- Количество транзисторов: **2300**
- Площадь кристалла: 12 мм²
- Техпроцесс: 10 мкм
- Тактовая частота: 740 кГц опорная (или 92,6 кГц тактовая, время выполнения одной инструкции 8 тактов)
- Разрядность регистров: 4 бита



Core i7

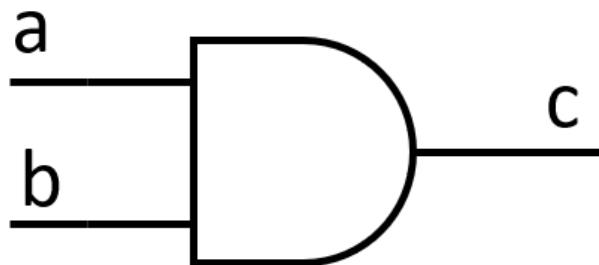


- Дата анонса первого процессора серии: 2008 год
- Количество транзисторов: **~1,16*10^9**
- Техпроцесс: 45—14 нм



Hardware description language

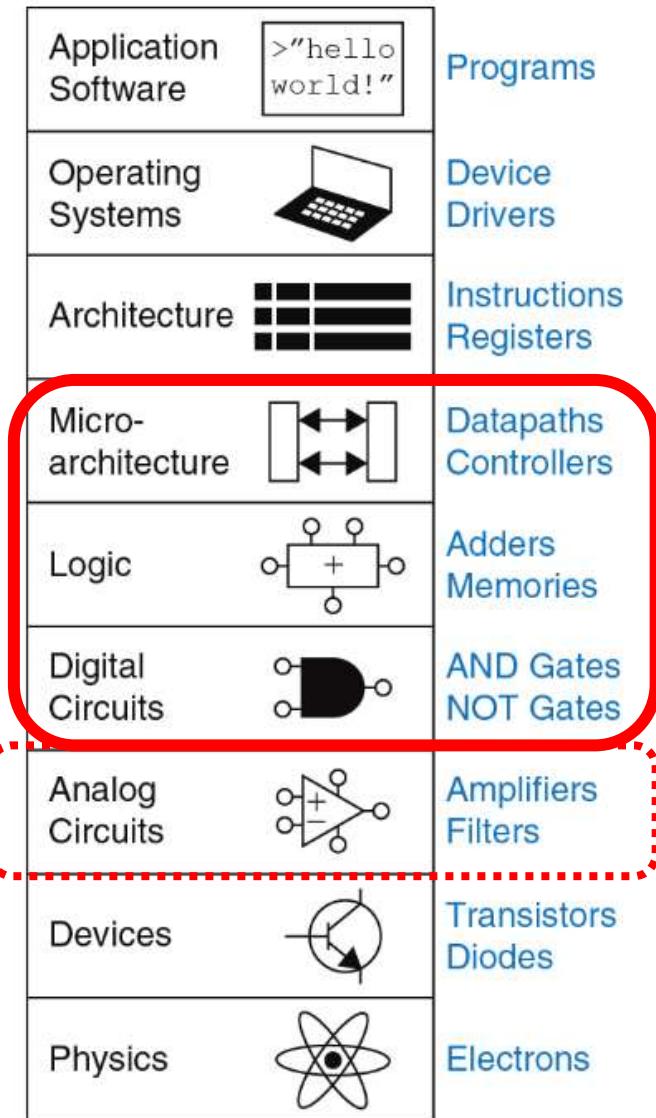
HDL – Hardware Description Language
(Язык описания аппаратуры)



assign $c = a \& b;$

- *Verilog*
- *VHDL*
- *SystemVerilog*

Verilog-A

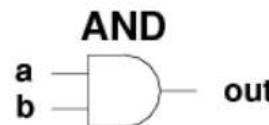


Verilog HDL. История

- Verilog был разработан компанией Gateway Design Automation в 1984 году как фирменный язык для симуляции логических схем.
- В 1989 году Gateway приобрела компания Cadence, и Verilog стал открытый стандартом в 1990 году под управлением сообщества Open Verilog International.
- Язык стал стандартом IEEE в 1995 году.
- В 2005 году язык был расширен для упорядочивания и лучшей поддержки моделирования и верификации систем.
- Эти расширения были объединены в единый стандарт, который сейчас называется SystemVerilog (стандарт IEEE 1800-2009).



Комбинационная логика



a	b	out
0	0	0
0	1	0
1	0	0
1	1	1



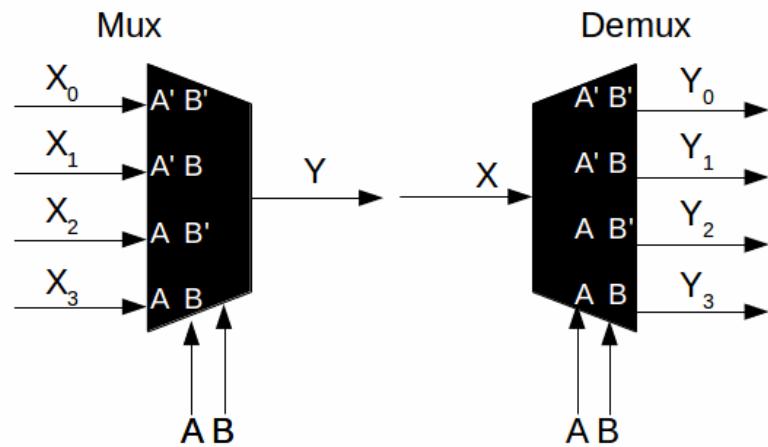
a	b	out
0	0	0
0	1	1
1	0	1
1	1	1



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

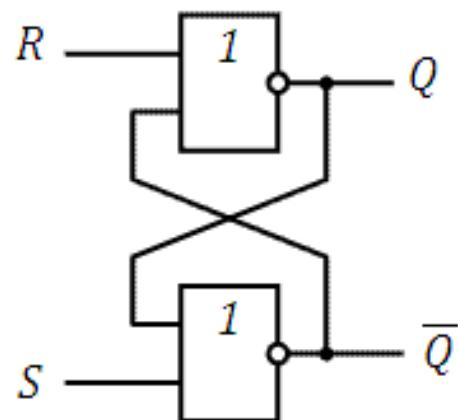


in	out
0	1
1	0

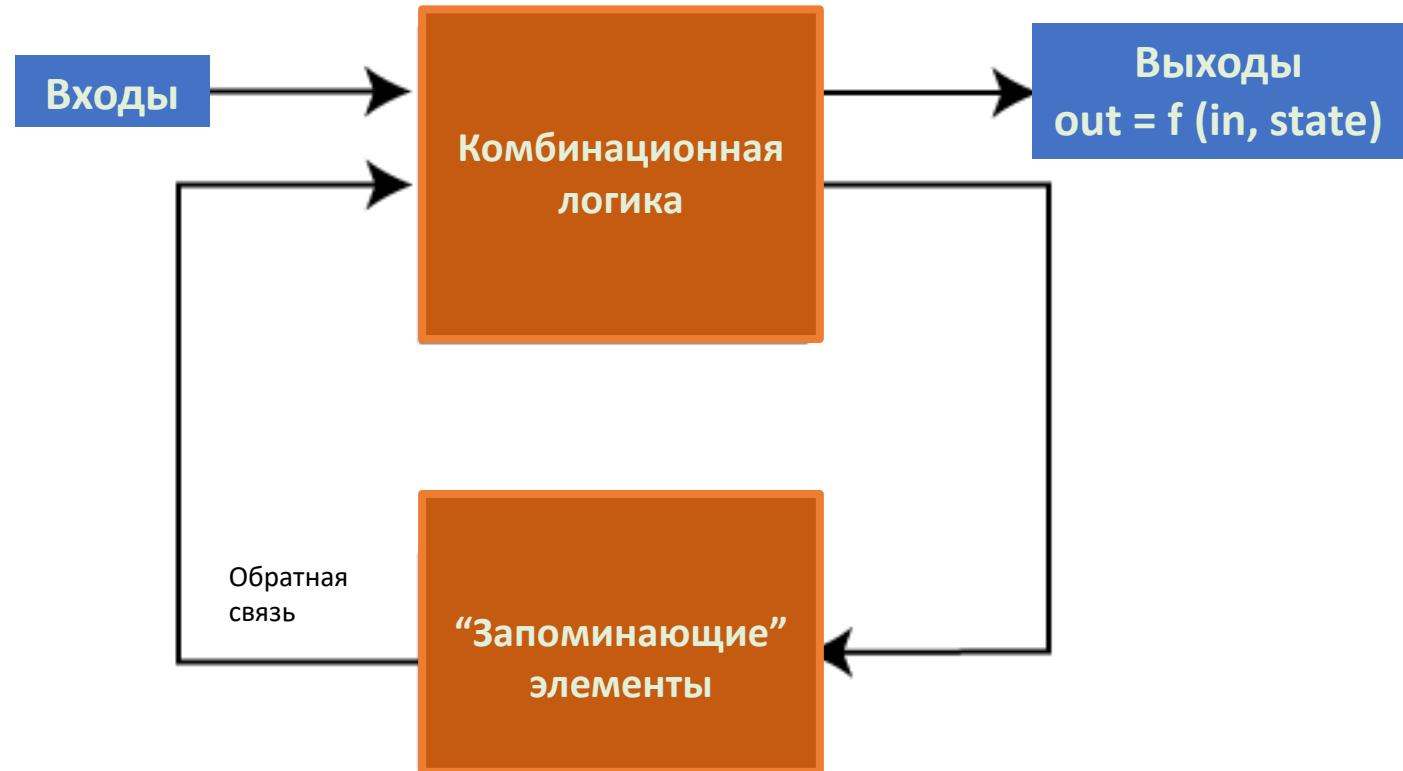


Комбинационная логика включает в себя базовые логические операции, а так же производные от них схемы, например мультиплексор, дешифратор, сумматор и т.д.

Последовательностная логика



RS-триггер на двух
элементах ИЛИ-НЕ



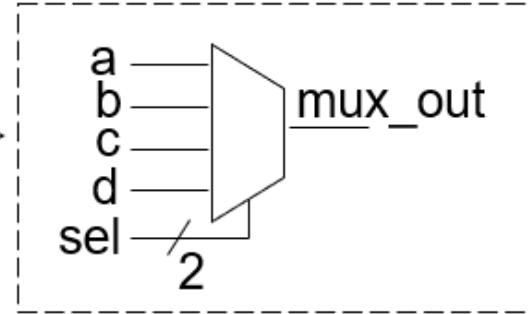
Последовательностная логика включает в себя базовые элементы (D-триггеры), работающие по тактовому сигналу, способные “запоминать текущее состояние” и производные от них, такие как регистр, счётчик, конечный автомат и т.д.

Синтез цифровой схемы

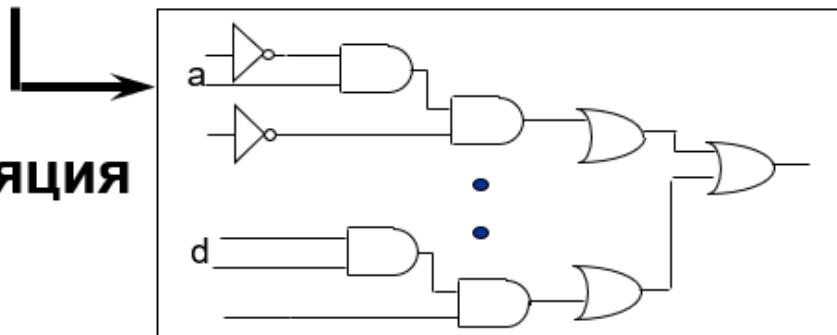
Verilog - описание

```
always @(a, b, c, d, sel)
  case (sel)
    2'b00: mux_out = a;
    2b'01: mux_out = b;
    2b'10: mux_out = c;
    2b'11: mux_out = d;
  endcase
```

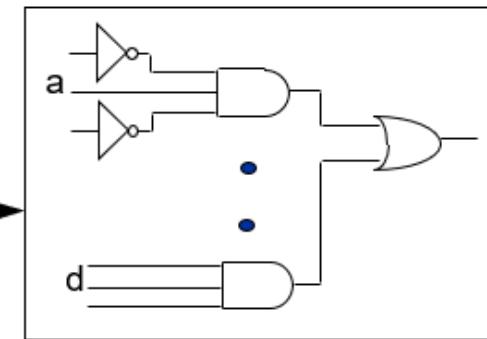
Подразумевается
поведение
мультплексора



Трансляция



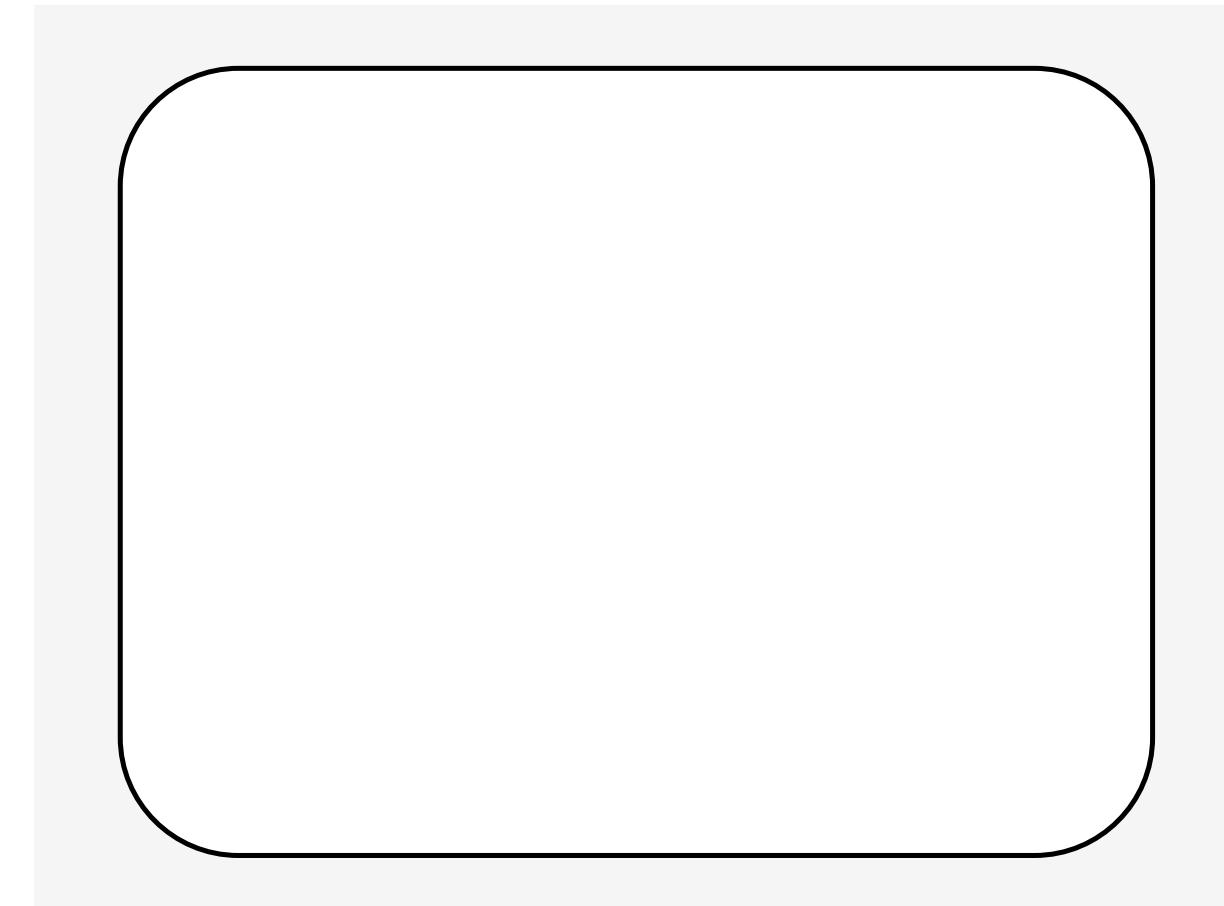
Оптимизация



Описание модуля верхнего уровня

module

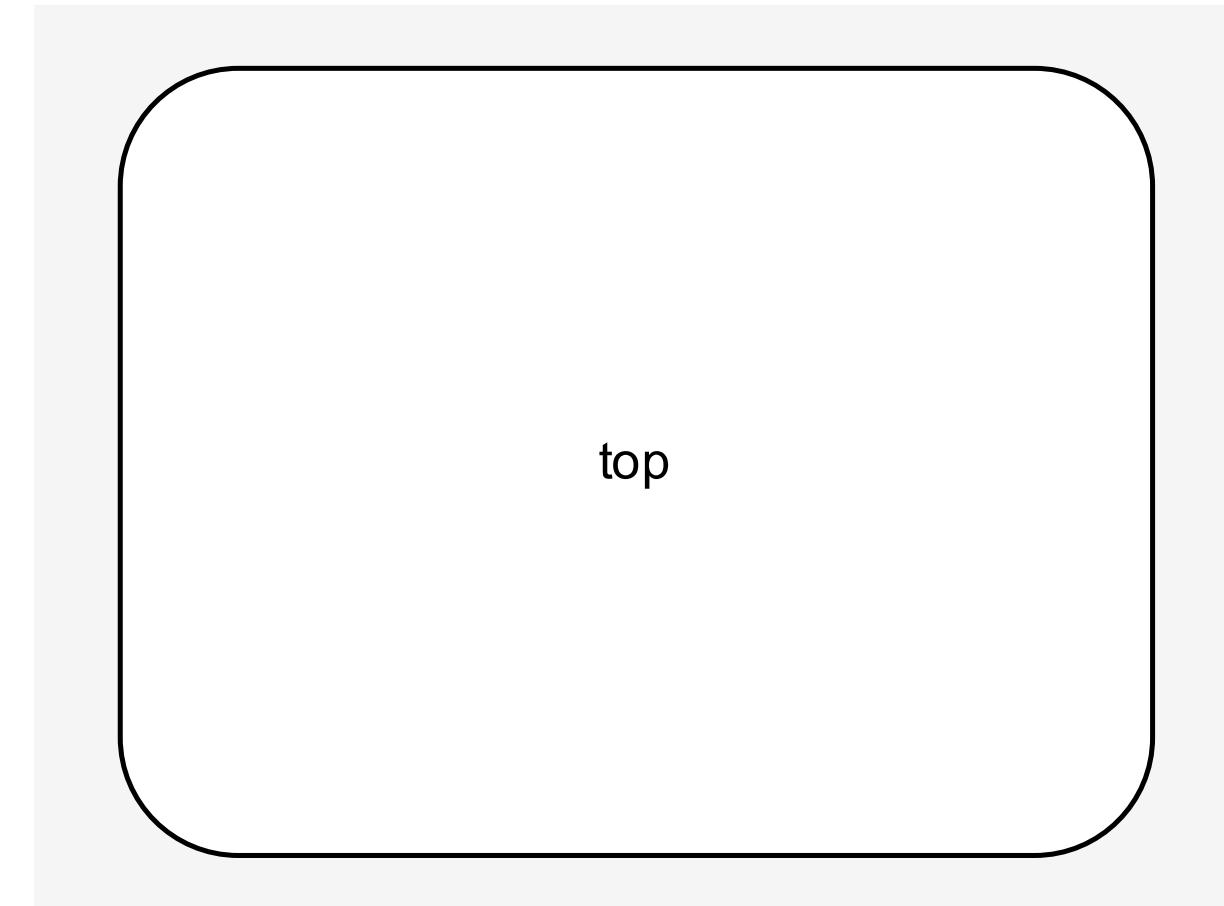
endmodule



Описание модуля верхнего уровня

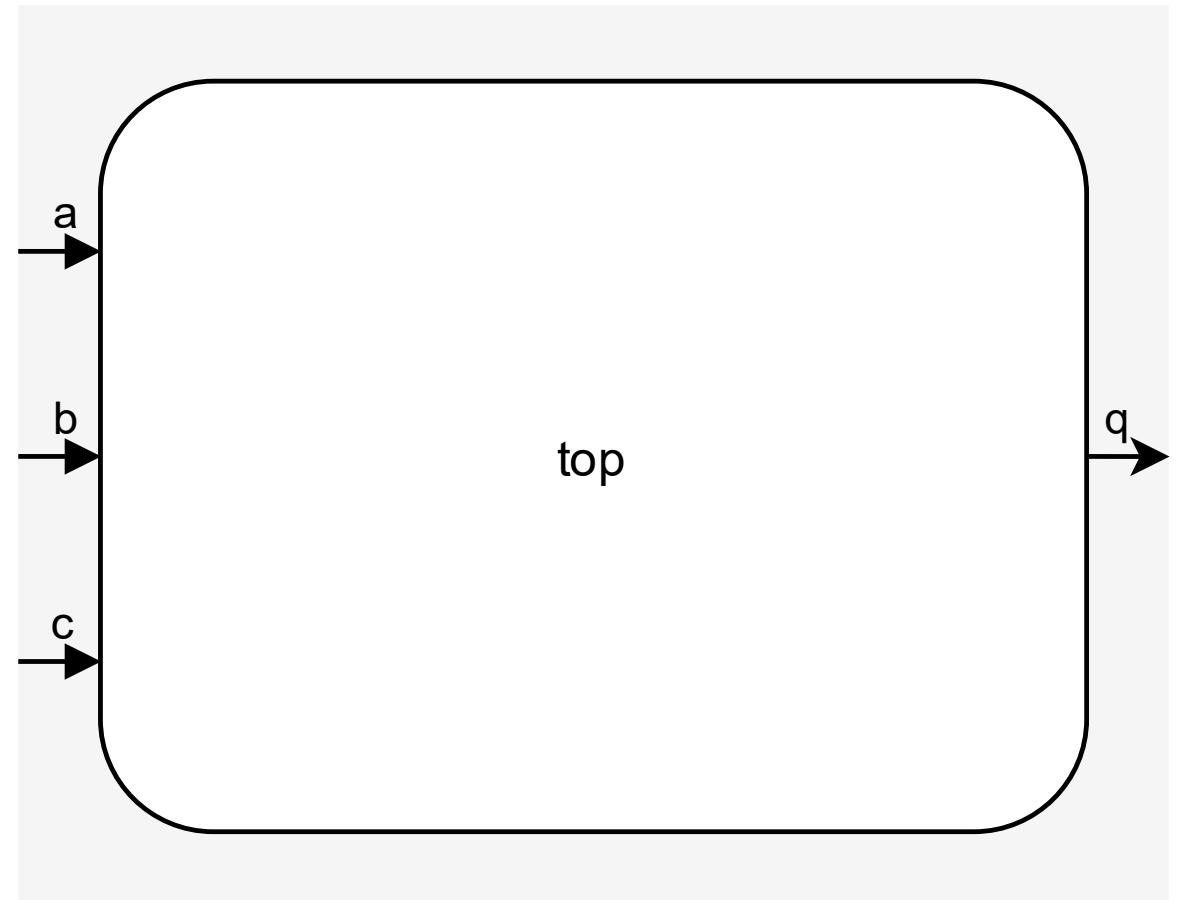
```
module top
```

```
endmodule
```



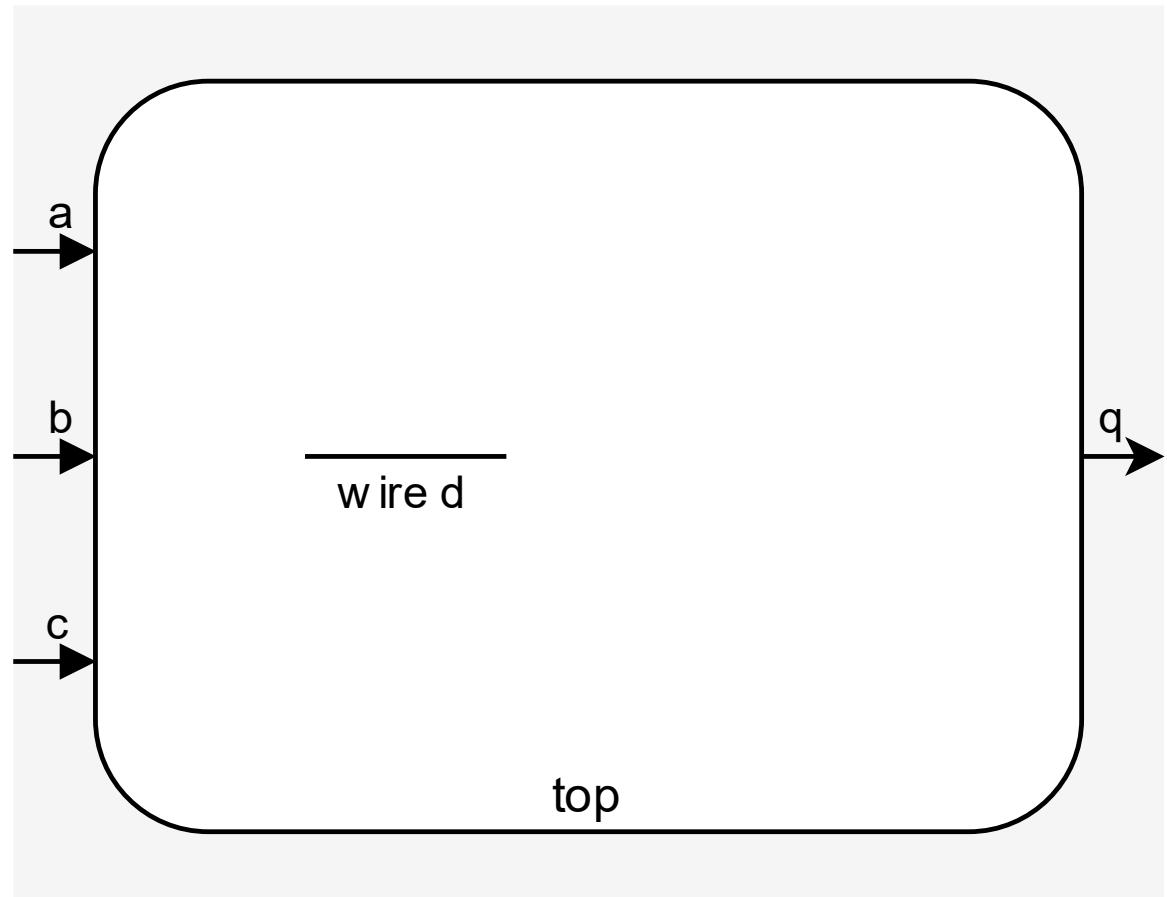
Описание модуля верхнего уровня

```
module top (
    input a,
    input b,
    input c,
    output q
);
endmodule
```



Создание провода wire

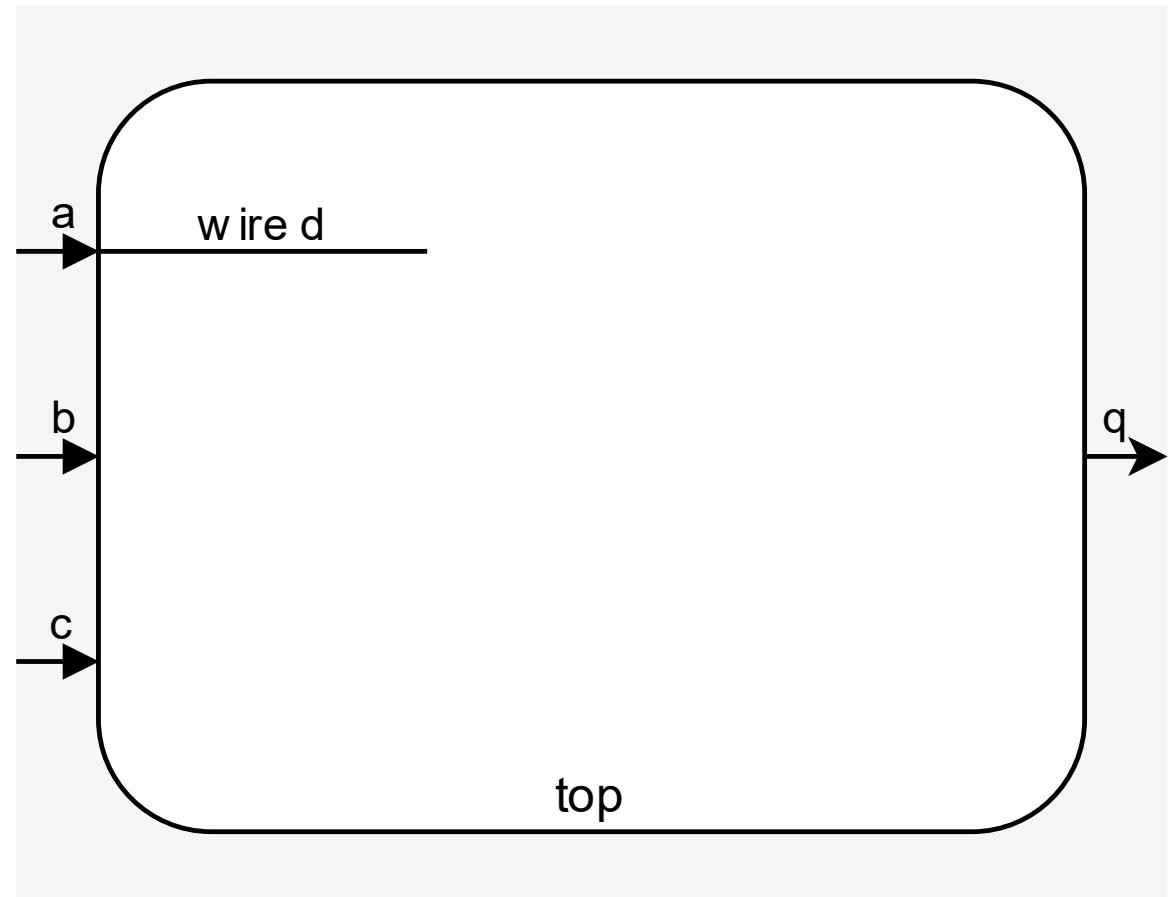
```
module top (
    input c,
    input a,
    input b,
    output q
);
wire d;
endmodule
```



Назначение сигнала входного порта проводу через assign

```
module top (
    input c,
    input a,
    input b,
    output q
);
wire d;
assign d = a;

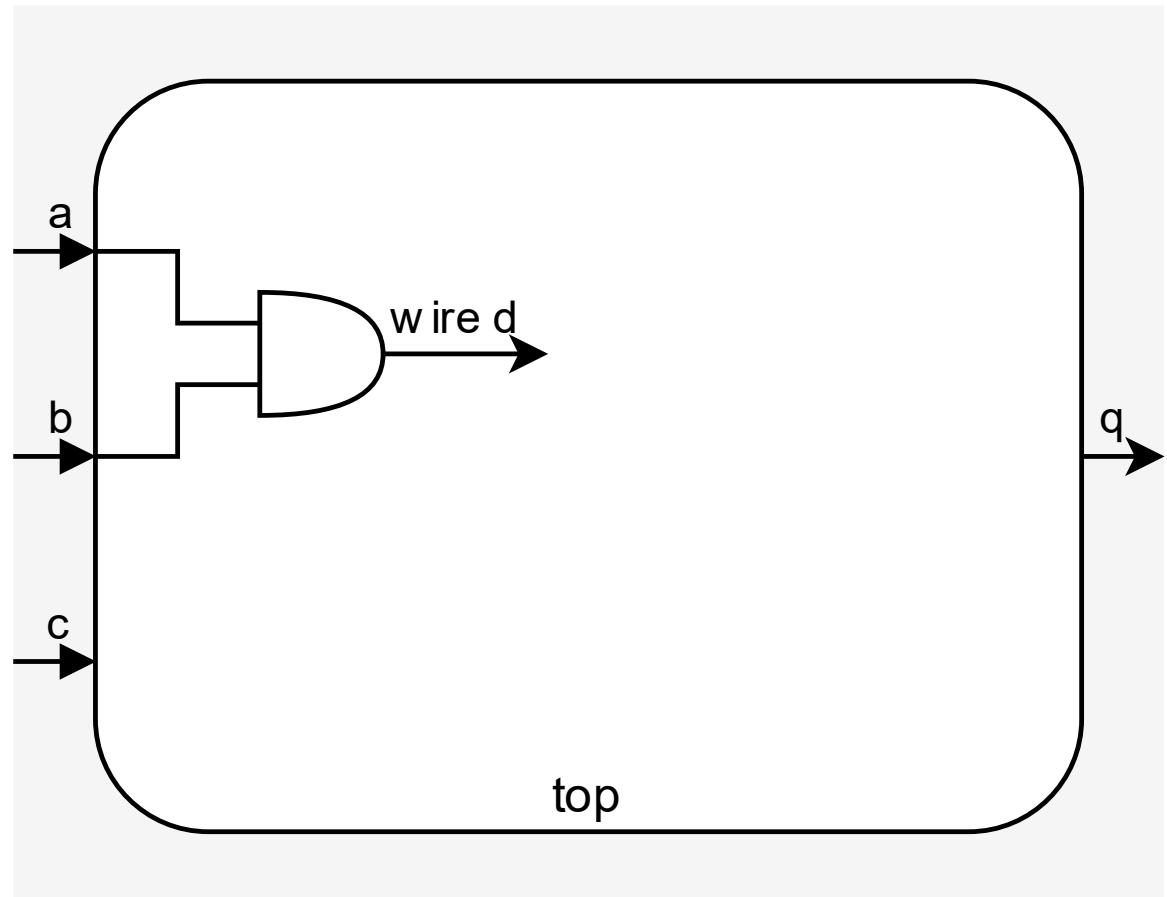
endmodule
```



Использование логического оператора

```
module top (
    input c,
    input a,
    input b,
    output q
);
wire d;
assign d = a & b;

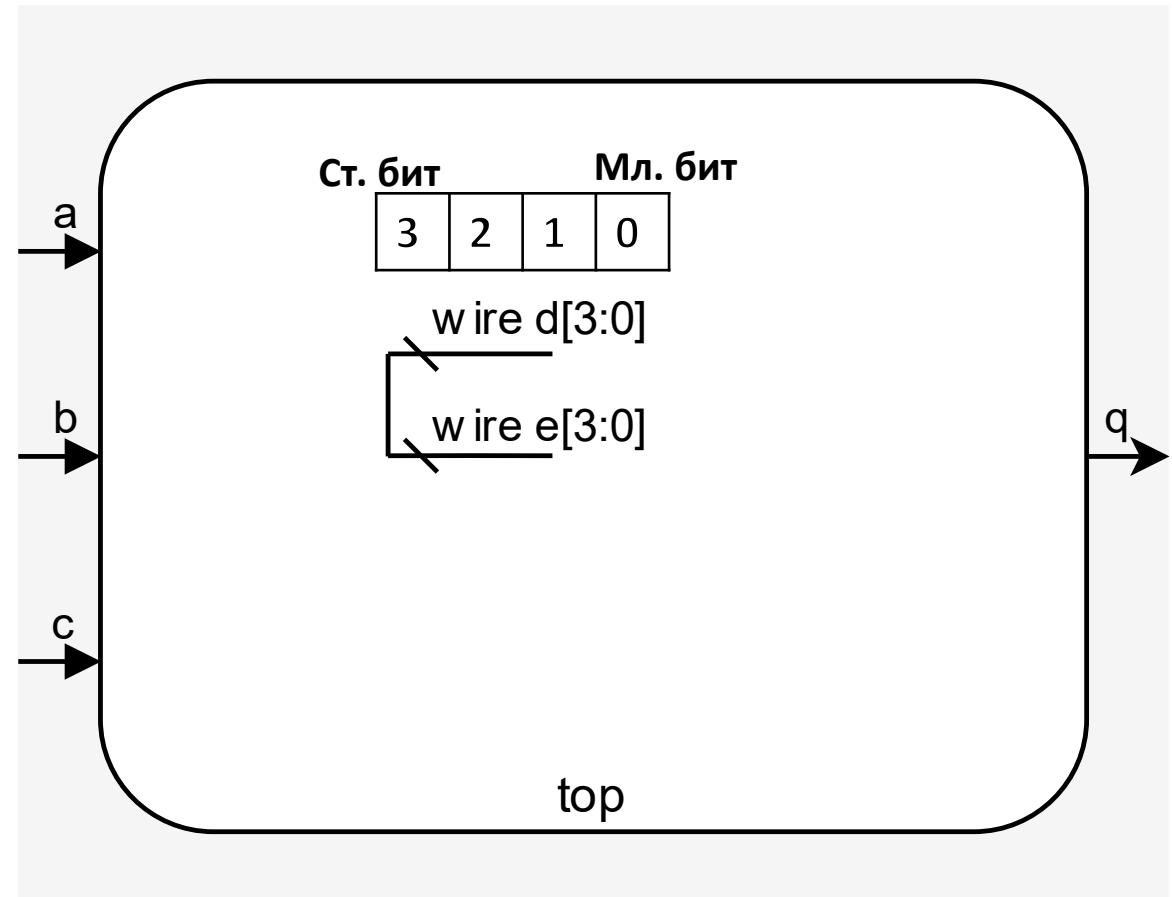
endmodule
```



Присвоение значения одной шины к другой

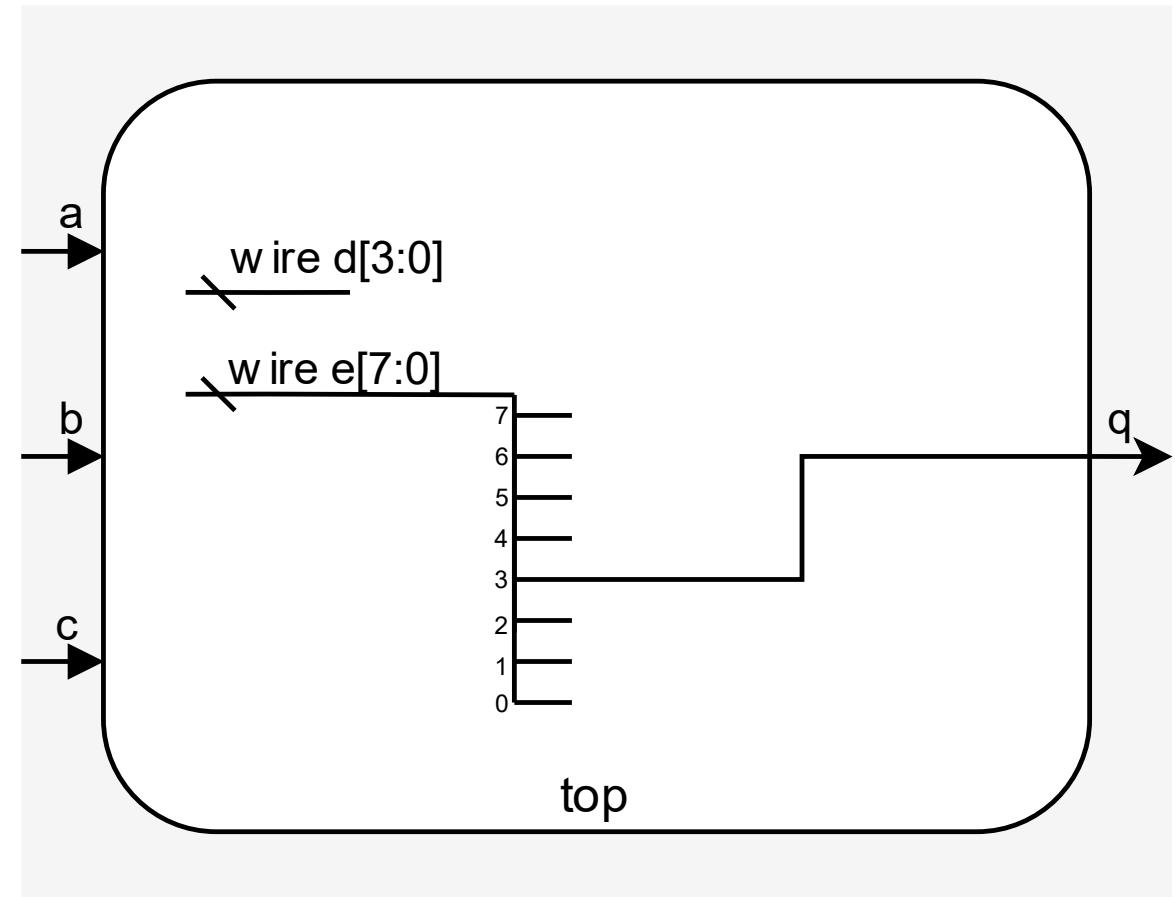
```
module top (
    input c,
    input a,
    input b,
    output q
);
wire [3:0] d; //две 4-битные шины
wire [3:0] e;
//иначе можно объявить
//через wire [3:0] d, e;
assign d = e;

endmodule
```



Выборка бита из шины через указание индекса бита

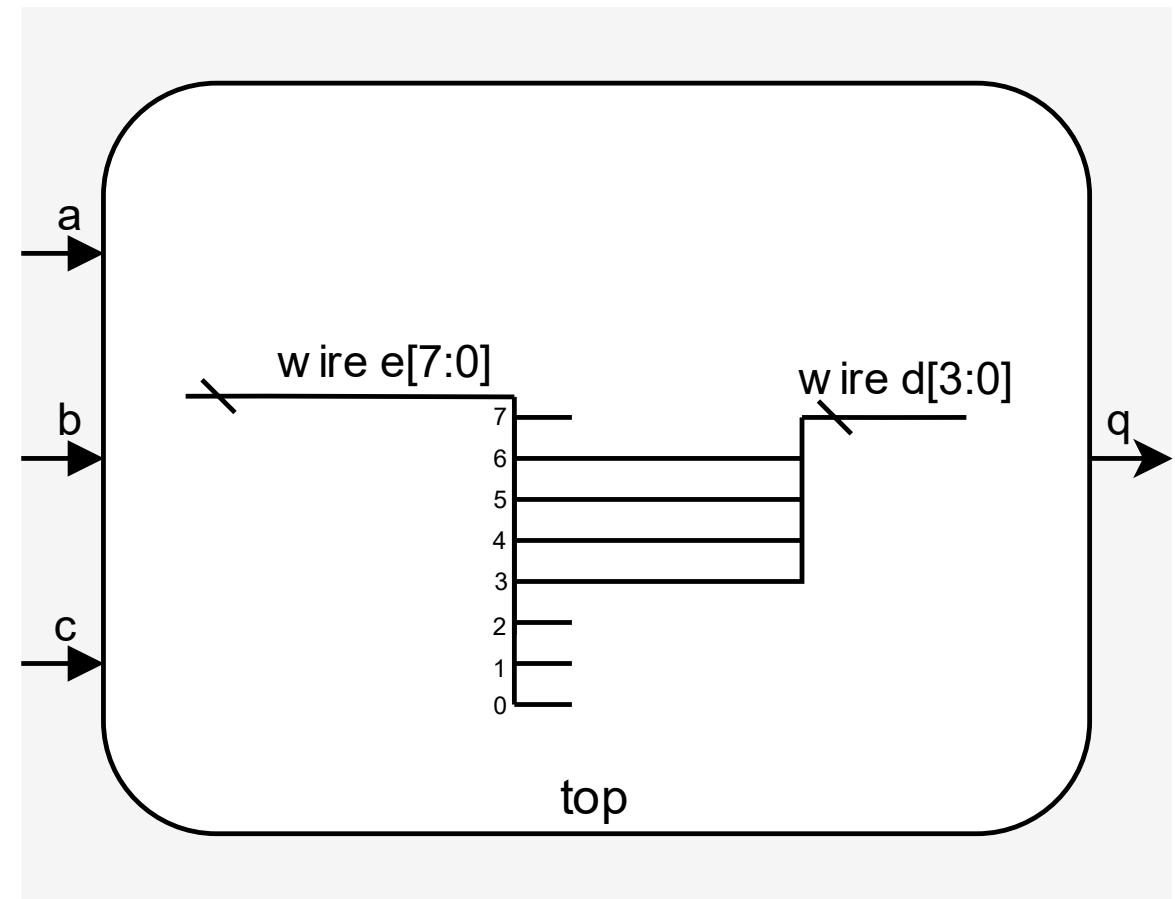
```
module top (
    input c,
    input a,
    input b,
    output q
);
wire [3:0] d;
wire [7:0] e;
assign q = e[3];
endmodule
```



Выборка нескольких бит из шины через диапазон

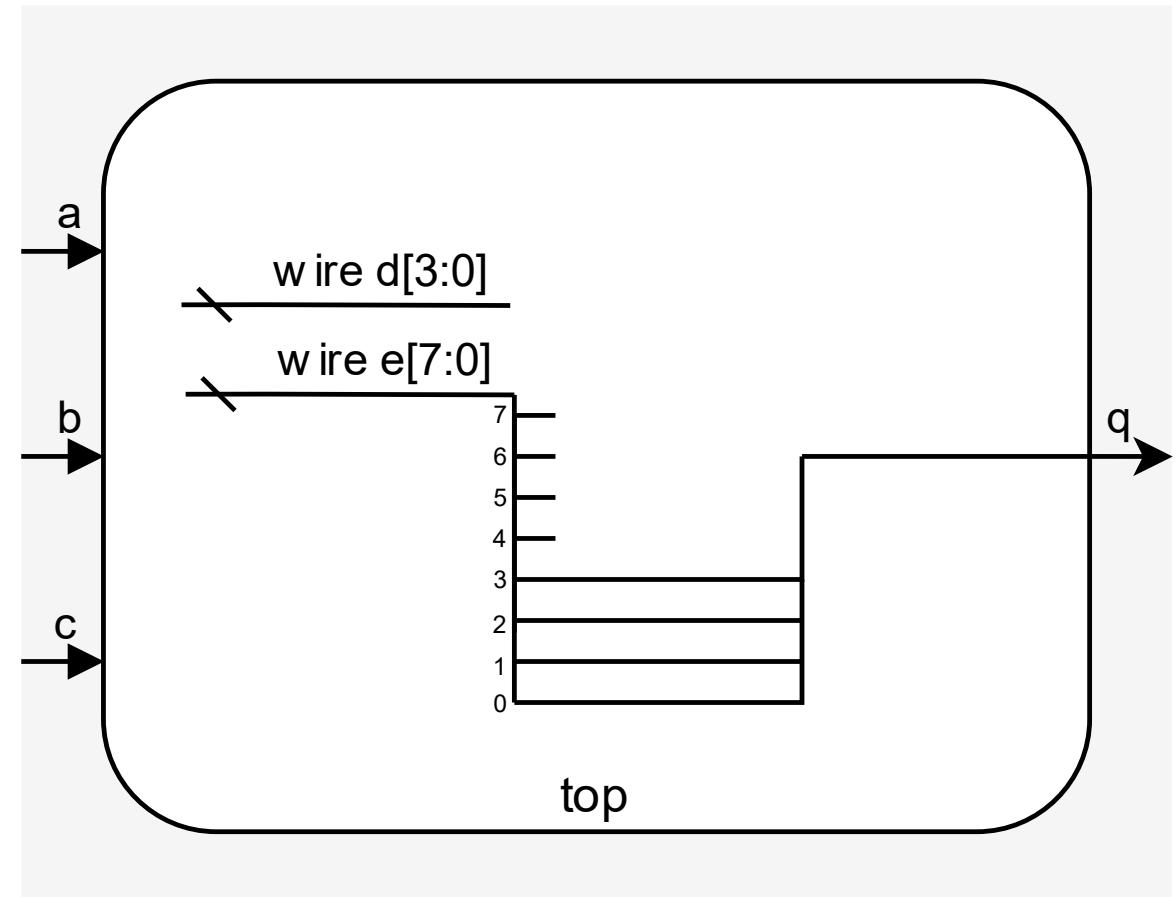
```
module top (
    input c,
    input a,
    input b,
    output q
);
wire [3:0] d;
wire [7:0] e;
assign d = e[6:3];

endmodule
```



Выборка нескольких бит из шины через другую шину

```
module top (
    input c,
    input a,
    input b,
    output q
);
wire [3:0] d;
wire [7:0] e;
assign q = e[d];
endmodule
```



Использование массива шин

```

module top (
    input c,
    input a,
    input b,
    output q
);

wire [3:0] d;
wire [7:0] e [0:24]; // массив из 25
                      // 8-битных шин
assign d = e[14][7:4]; // подключение
                      // 4 бит из 14 шины массива
endmodule

```



Формат описания чисел Verilog HDL

a = 4'b7

число бит
система счисления
значение числа

```
wire [10:0] a = 7; // 32-битное десятичное число, которое "обрезается" до 11 бит
wire [10:0] b = 'd7; // 11-битное десятичное число
wire [10:0] b = 11'd7; // 11-битное десятичное число
wire [3:0] c = 4'b0101; // 4-битное двоичное число
wire [3:0] d = 8'h7B; // 8-ми битное шестнадцатеричное число
wire [47:0] e = 48'hEFCA7Ed98F; // 48-ми битное шестнадцатеричное число
wire signed [10:0] b = -11'd7 ; // 11-битное отрицательное десятичное число
```

Основные операции Verilog HDL

Символ	Назначение
{}	Конкатенация (concatenation)
+ - * /	Арифметические (arithmetic)
%	Модуль (modulus)
> >= < <=	Отношения (relational)
!	Логическое отрицание (logical NOT)
&&	Логическое И (logical AND)
	Логическое ИЛИ (logical OR)

Основные операции Verilog HDL. Продолжение

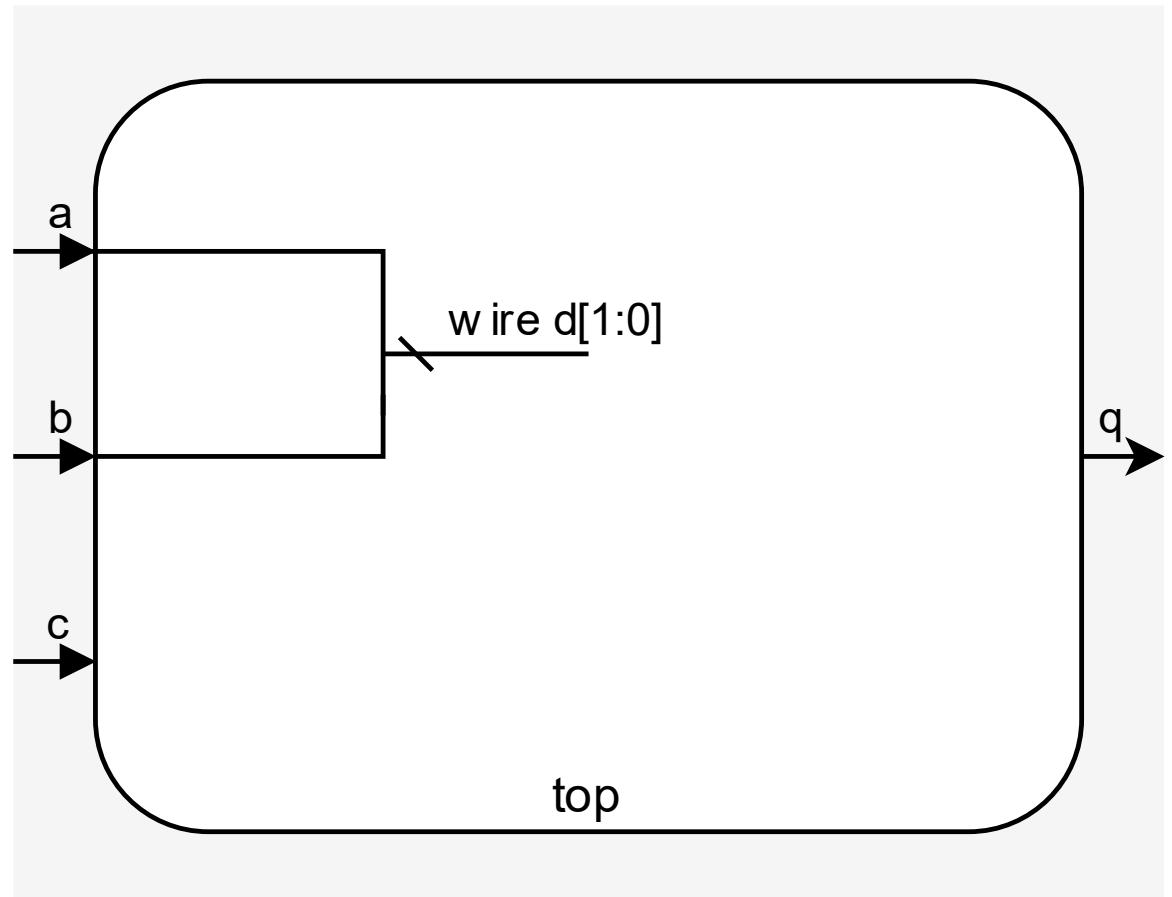
Символ	Назначение
<code>==</code>	Логическое равенство (logical equality)
<code>!=</code>	Логическое неравенство (logical inequality)
<code>====</code>	Идентичность (case equality)
<code>!====</code>	Не идентичность (case inequality)
<code>~</code>	Побитовая инверсия (bit-wise NOT)
<code>&</code>	Побитовое И (bit-wise AND)
<code> </code>	Побитовое ИЛИ (bit-wise OR)

Основные операции Verilog HDL. Продолжение

Символ	Назначение
<<	Сдвиг влево (left shift)
>>	Сдвиг вправо (right shift)
<<<	Циклический сдвиг влево (arithm. left shift)
>>>	Циклический сдвиг вправо (arithm. right shift)
?:	Тернарный оператор (ternary)

Манипуляции с битами Verilog HDL

```
module top (
    input c,
    input a,
    input b,
    output q
);
    wire [1:0] d;
    assign d = {a,b};
endmodule
```

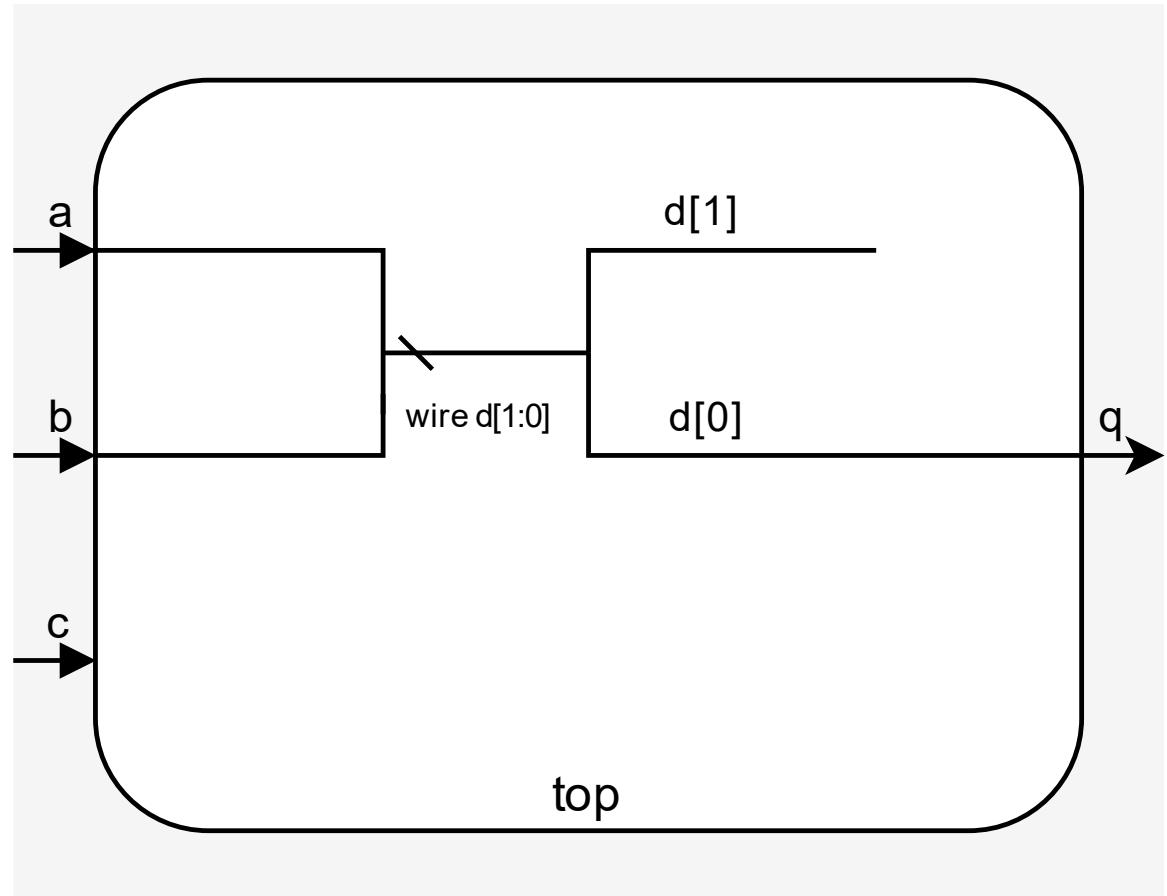


Манипуляции с битами Verilog HDL

```
module top (
    input c,
    input a,
    input b,
    output q
);

wire [1:0] d;
assign d = {a,b};
assign q = d[0];

endmodule
```



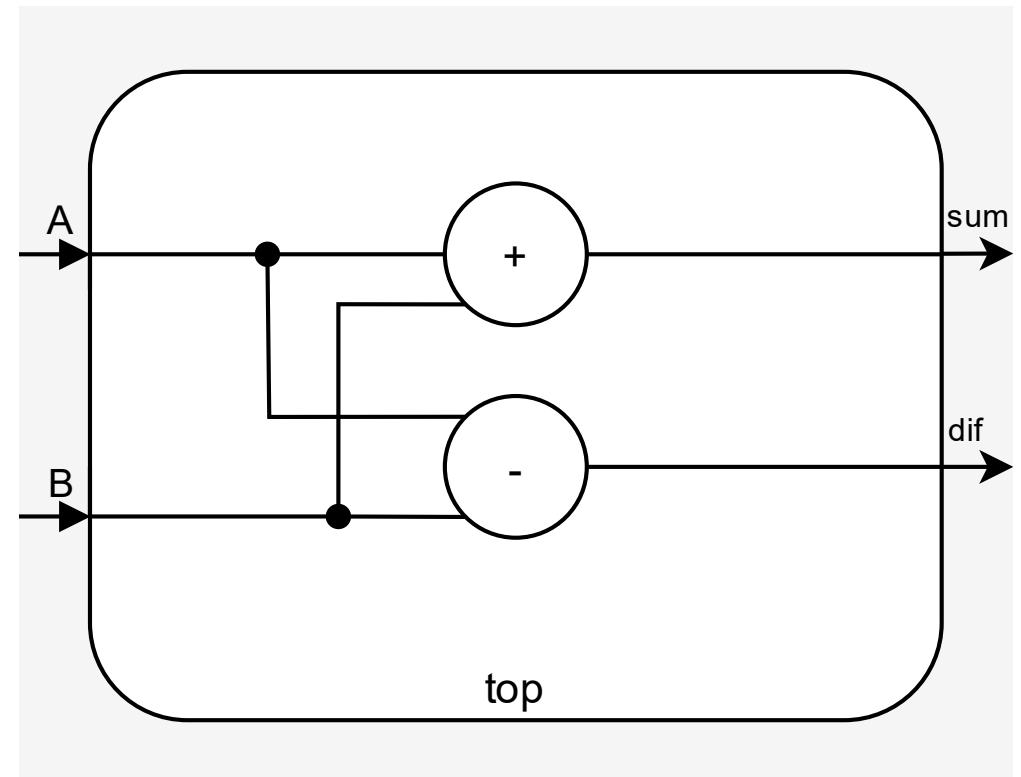
Сумма, Вычитание

```

module simple_adder_subtractor(
    input [7:0] A, B, // два 8-битных входных операнда
    output [8:0] sum, dif // результат суммы, вычитания
);
    assign sum = A + B;
    assign dif = A - B;
endmodule

module simple_adder_subtractor_signed(
    input signed [7:0] A, B, // два 8-битных входных операнда, старший
    // бит отводится под знак
    output signed [8:0] sum, dif // результат суммы, вычитания
);
    assign sum = A + B;
    assign dif = A - B;
endmodule

```

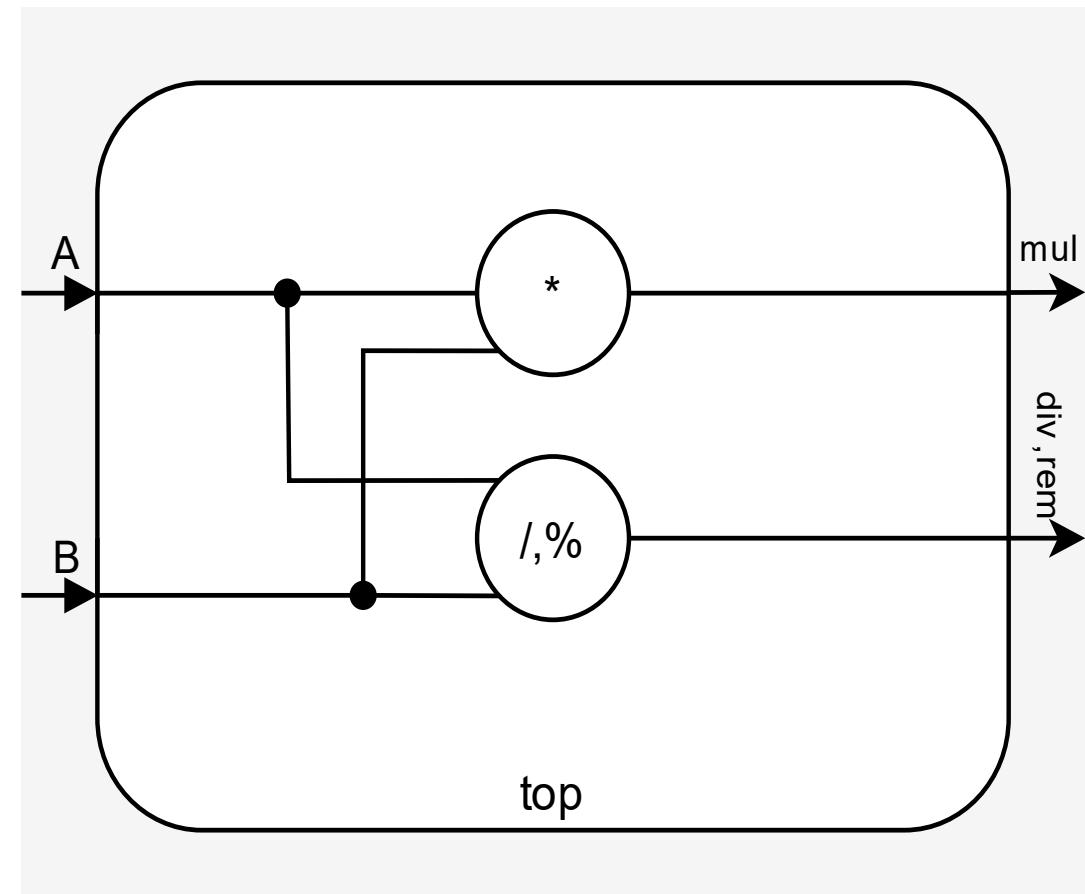


Умножение, Деление

```

module simple_mul_div (
    input [7:0] A,B, // два входных 8-ми битных операнда
    output [15:0] mul,div,rem // для избежания переполнения
    // разрядность результата операции
    // = разрядность operandов *2
);
assign mul = A * B;
assign div = A / B;
assign out_rem = A % B;
// Такие операции не используются в синтезируемом коде из-за
// сложности реализации
// по площади и плохими временными характеристиками. Такой
// код может быть использован для симуляции работы блока
// Например в процессоре деление реализуется итерационными
// алгоритмами
endmodule

```

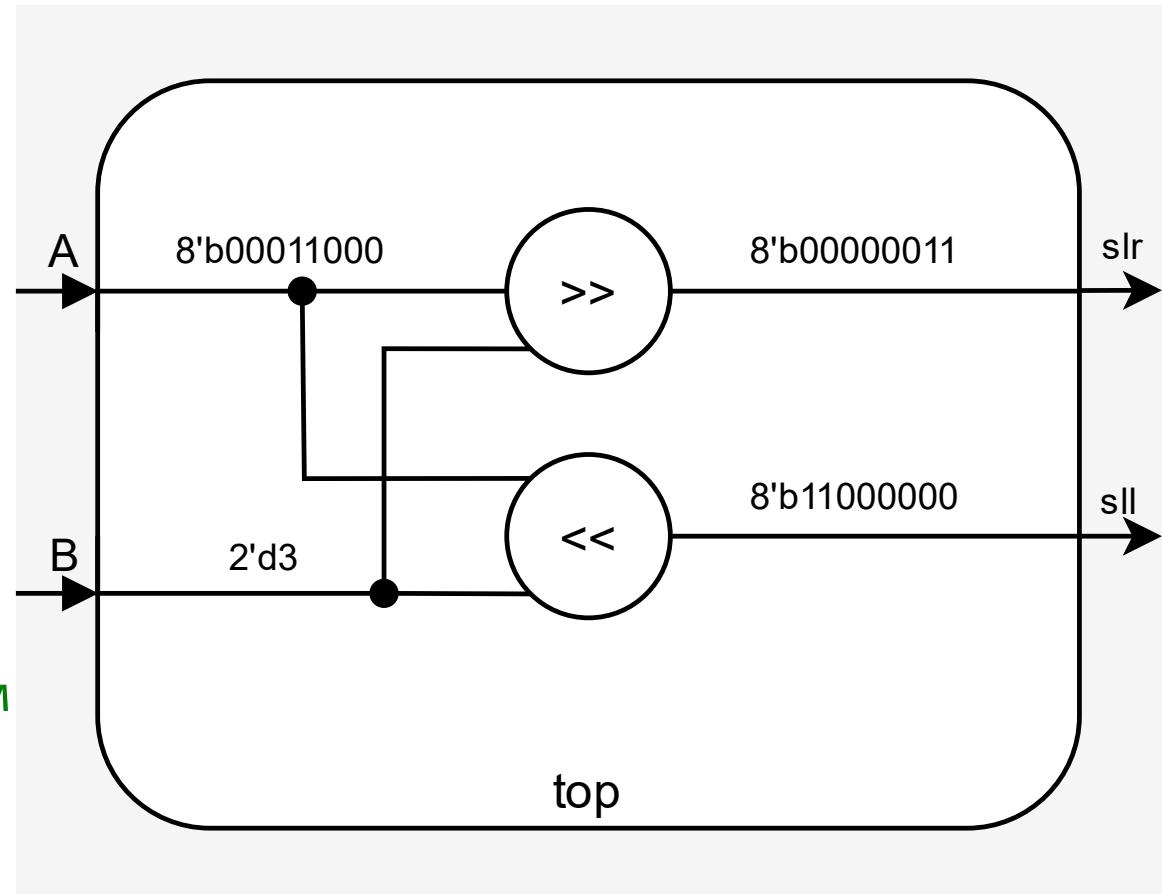


Сдвиг

```

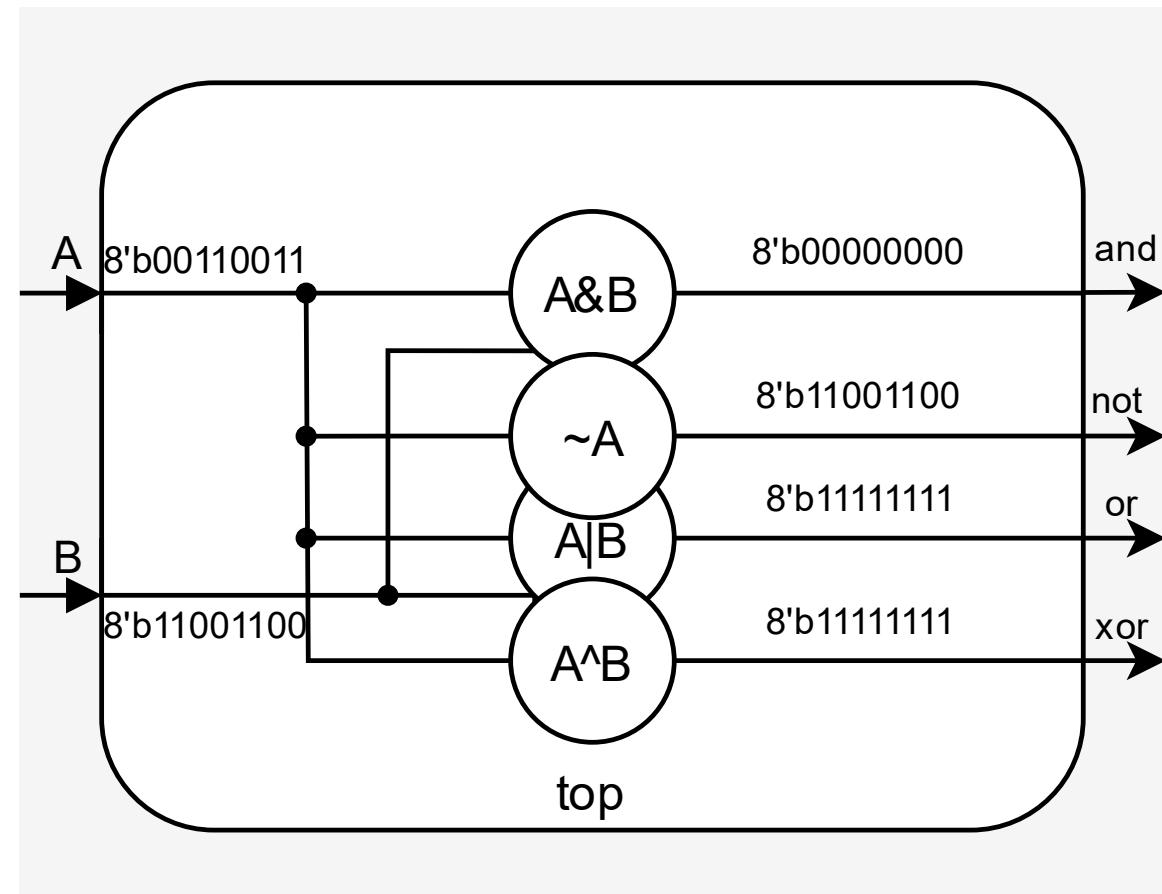
module shift (
    input [7:0] A,B,
    output[7:0] sll,slr,sar
);
//логический сдвиг влево операнда А на
//значение операнда В
assign sll = A<<B;
//логический сдвиг вправо операнда А на
//значение операнда В
assign slr = A>>B;
//арифметический сдвиг вправо( с сохранением
//знака)
assign sar = A >>>3;

```



Побитовые операции

```
module bit_logic (
    input [7:0] A,B,
    output[7:0] out_and,out_or,out_xor,
out_not
);
    assign out_and = A & B;
    assign out_or = A | B;
    assign out_xor = A ^ B;
    assign out_not = ~A;
endmodule
```

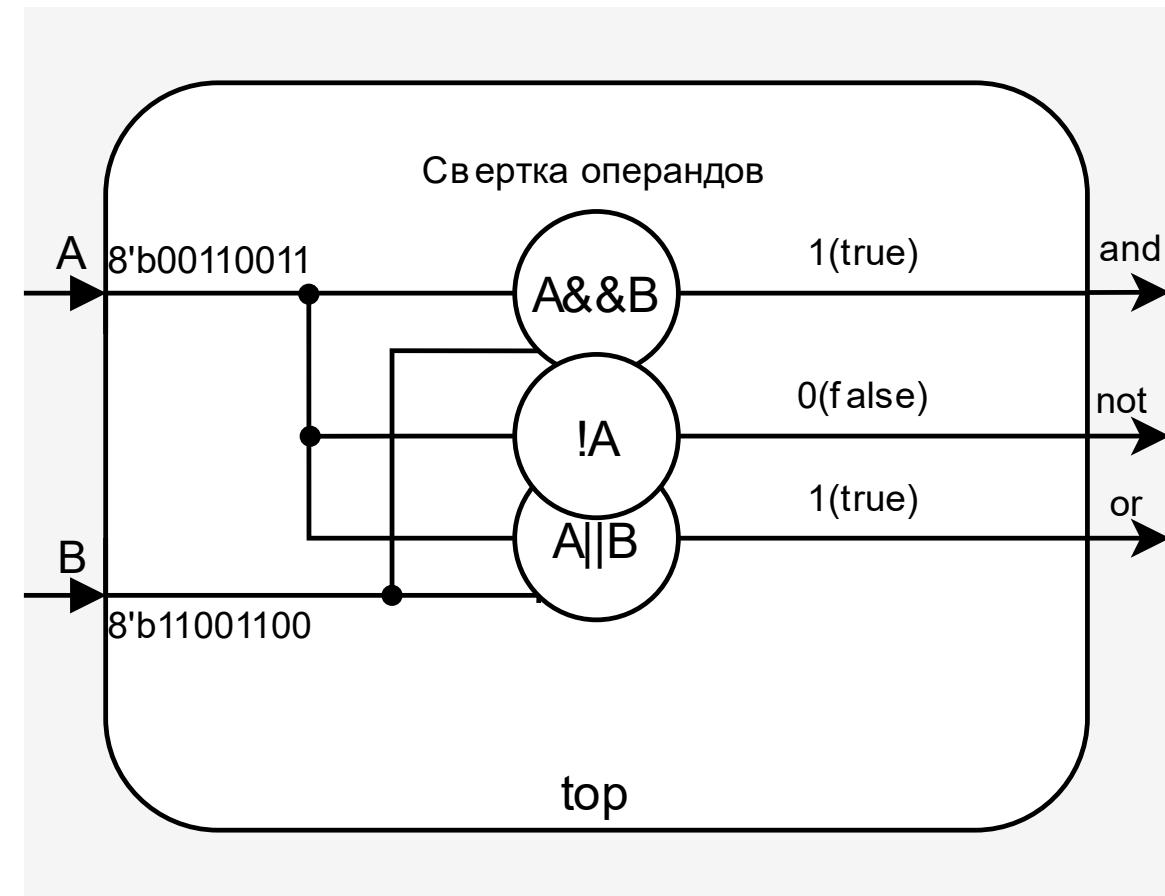


Булевые операции

```

module bool_logic (
    input [7:0] A,B,
    output out_and,out_or, out_not
);
// шина сворачивается в “1”, если есть
// хотя бы одна “1” вшине, иначе
// сворачивается в “0”. (|A , |B)
    assign out_and = A && B;
    assign out_or = A || B;
    assign out_not = !A;
endmodule

```

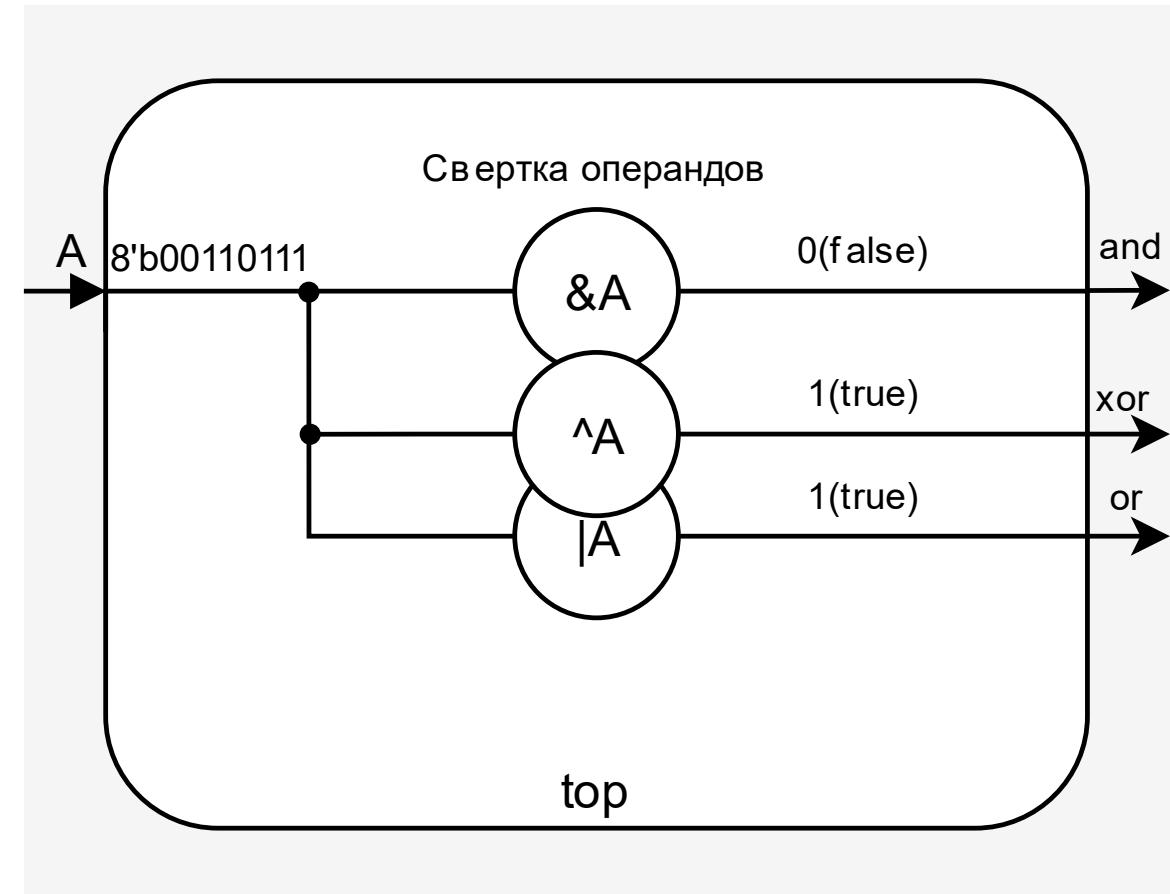


Операции свертки

```

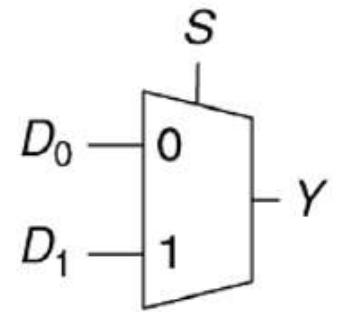
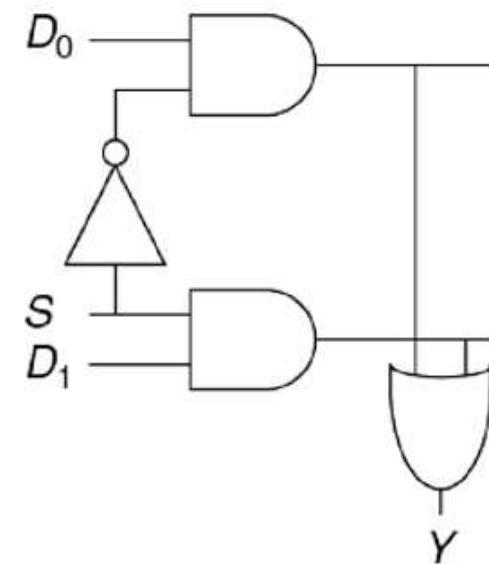
module red_logic (
    input [7:0] A,
    output red_and,red_or, red_xor
);
    // Операция выполняется между битами в
    //шине
    assign red_and = &A;
    assign red_or = |A;
    assign red_xor = ^A;
endmodule

```



Мультиплексор

```
module mux(  
    input D0,D1,S,  
    output Y  
);  
    assign Y = S ? D1 : D0; // тернарный  
    оператор  
endmodule
```



Операторы сравнения

```
module operators(  
    input[7:0] A, B,  
    output eq, neq, higher_eq, lower_eq, higher, lower  
);  
    assign eq      = (A == B);  
    assign neq     = (A != B);  
    assign higher_eq = (A >= B);  
    assign lower_eq = (A <= B);  
    assign higher   = (A > B);  
    assign lower    = (A < B);  
  
endmodule
```

Тип reg

```
module top(  
    input clk,  
    input [7:0] a,  
    input b,  
    output reg[3:0] q  
);  
// тип reg используется только в процедурных блоках always !  
    always @(*) begin  
        q = !a[7:4];  
    end  
endmodule
```

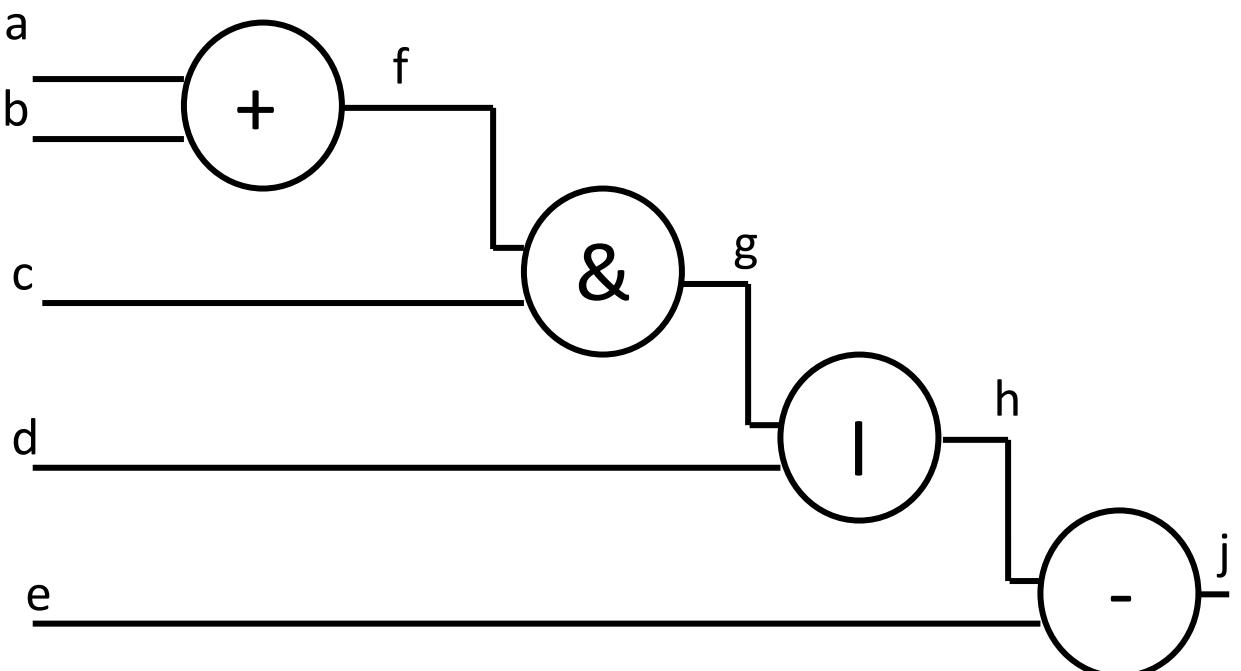
Процедурный always блок

```
wire [3:0] a,b,c,d,e;
```

```
reg [3:0] f,g,h,j;
```

// тип reg используется только
в процедурных блоках always !

```
always @(*) begin
    f = a + b;
    g = f & c;
    h = g | d;
    j = h - e;
end
```



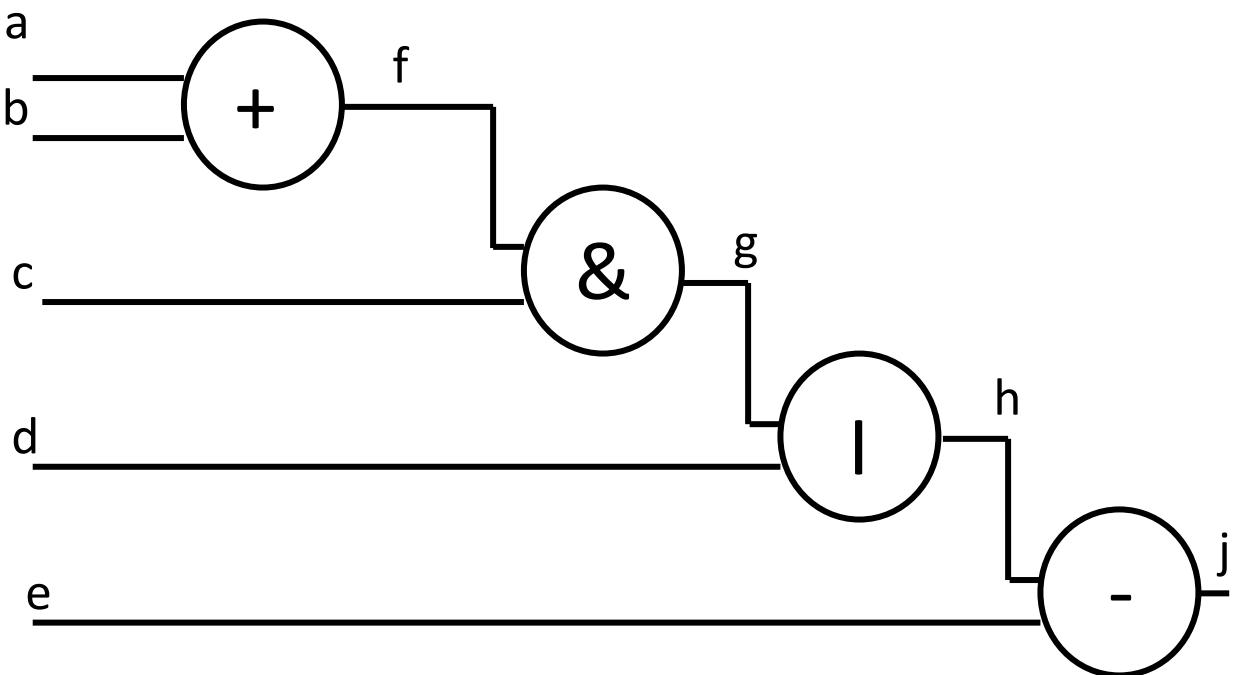
Процедурный always блок

```
wire [3:0] a,b,c,d,e;
```

```
reg [3:0] f,g,h,j;
```

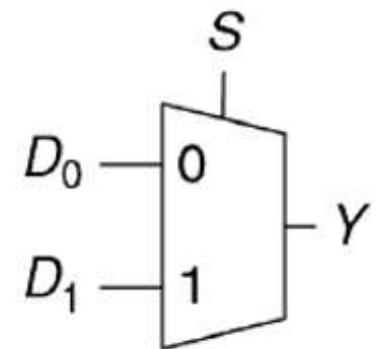
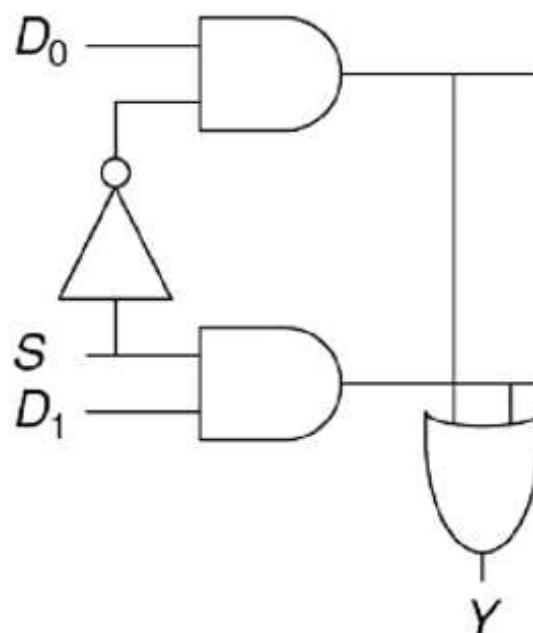
// тип reg используется только
в процедурных блоках always !

```
always @(*) begin
    j = (((a + b) & c) | d ) - e;
end
```



Мультиплексор через условия в блоке always

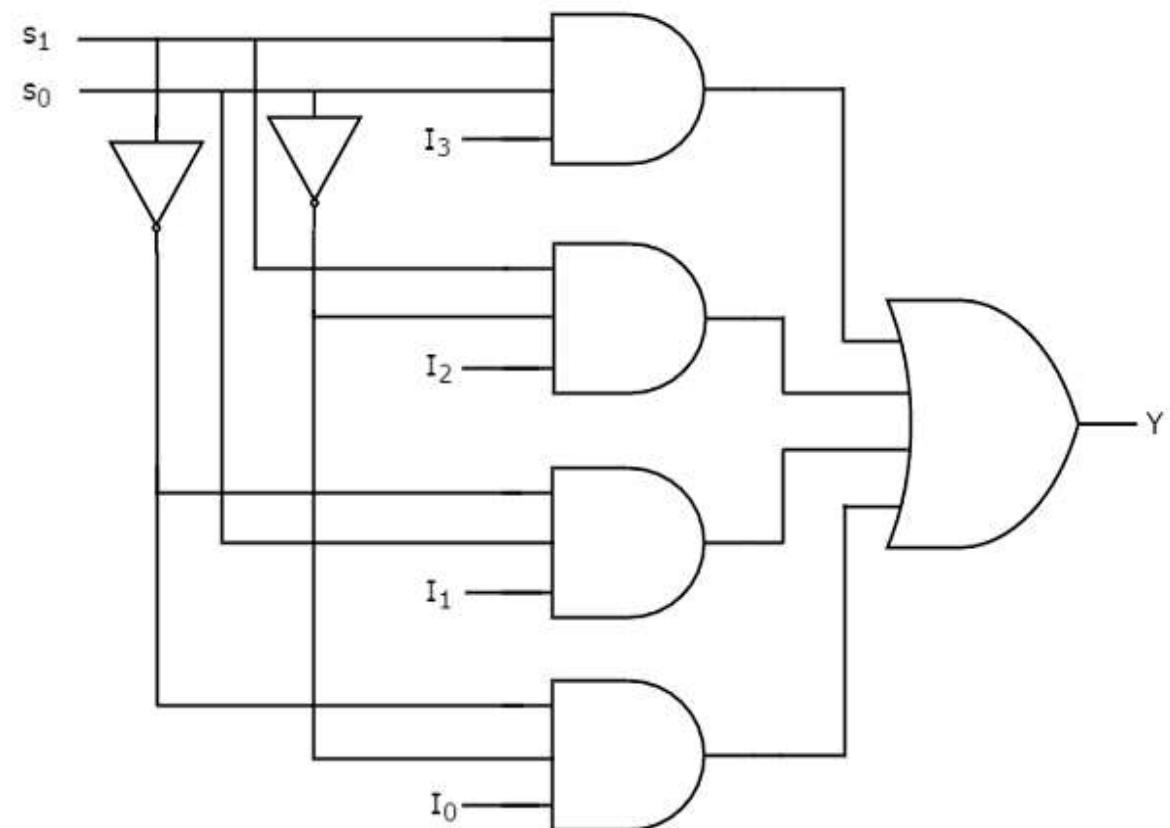
```
module mux(  
    input D0,D1,S,  
    output reg Y  
);  
always@(D0 or D1 or S) begin  
    if(S) begin  
        Y = D1;  
    end else begin  
        Y = D0;  
    end  
end  
endmodule
```



Оператор Case

Case позволяет создавать шифраторы и дешифраторы

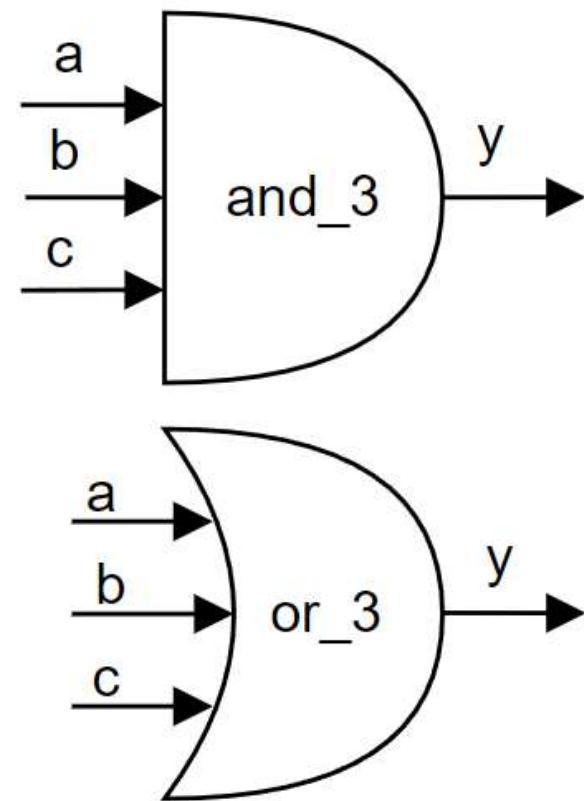
```
wire [1:0] option;  
wire [7:0] a,b,c,d;  
reg [7:0] e;  
always @(a or b or c or d or option) begin  
    case (option)  
        0: e = a;  
        1: e = b;  
        2: e = c;  
        3: e = d;  
        default:  
    endcase  
end
```



Иерархия модулей

```
module and_3(
    input a,b,c,
    output y
);
    assign y = a & b & c;
endmodule

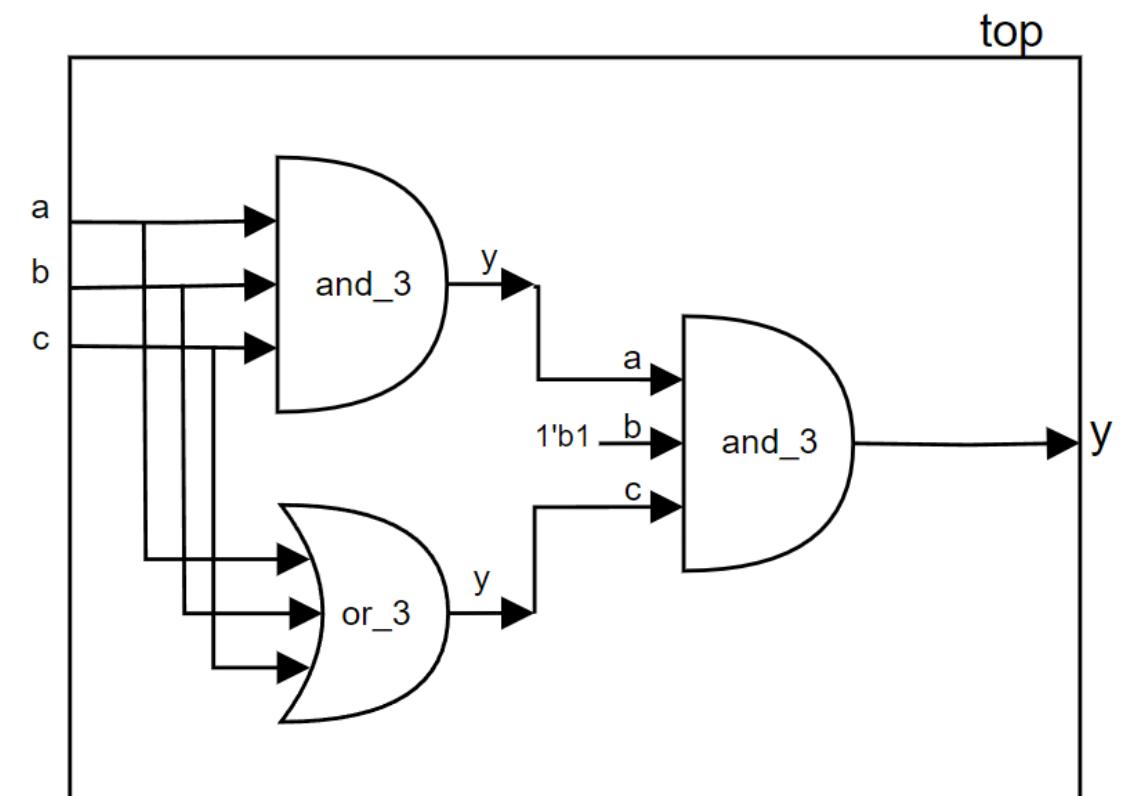
module or_3(
    input a,b,c,
    output y
);
    assign y = a | b | c;
endmodule
```



Иерархия модулей

```
module top(
    input a,b,c,
    output y
);
    wire net_1, net_2;
    or_3 mod1 (a,b,c,net_1);
    and_3 mod2 (a,b,c,net_2);
    and_3 mod3 (net_1,net_2,1'b1,y);

endmodule
```



Практика. Описание комбинационной и последовательностной логики на языке Verilog

45 минут

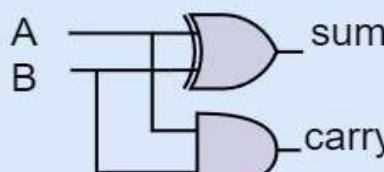
- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- **Практика. Описание комбинационной и последовательностной логики на языке Verilog(45 минут)**
- Введение в архитектуру RISC-V.
- Использование assembler RISC-V
- Введение в микроархитектуру RISC-V.
- Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Введение в архитектуру RISC-V.

- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- Описание комбинационной и последовательностной логики на языке Verilog
- **Введение в архитектуру RISC-V.**
- Использование assembler RISC-V
- Введение в микроархитектуру RISC-V.
- Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Уровни абстракции

- Процесс вычисления или выполнения программы может быть представлены на нескольких уровнях *абстракции*
- Каждому уровню абстракции соответствует своё средство проектирования

	Слой абстракции	Средство описания
Уровень программы	Алгоритм, Программа	Языки высокого уровня (C, C++) $a = b + c;$
	Архитектура команд(ISA)	Язык ассемблера add x4,x3,x2
	Машинный код	Двоичный код 0x00310233
Уровень аппаратуры	Микроархитектура	Блок-схемы, языки описания аппаратуры assign sum = A ^ B; assign carry = A & B;
	Реализация на аппаратном уровне	Электрические схемы 

Процессор

- Задача процессора (Central Processing Unit, CPU) - выполнять команды(инструкции)
- Команды процессора - примитивные операции, которые он может выполнять:
 - команды выполняются одна за другой последовательно
 - каждая команда выполняет какую-то небольшую часть работы (RISC)
 - команда выполняет операцию над operandами
 - некоторые команды могут менять последовательность выполнения команд
- Последовательность команд, хранящаяся в памяти - программа

Архитектура набора команд

- Процессоры делятся на “семьи”, в каждой из которых свой набор команд
- Каждый набор конкретного процессора реализует архитектуру набора команд (*Instruction Set Architecture, ISA*). Примеры ISA:
 - ARM, Intel x86, MIPS, RISC-V, IBM Power и т.д.
 - ISA определяет команды с точностью до двоичной кодировки, поэтому процессоры из одной *семьи* могут выполнять *одни и те же программы*
 - Язык ассемблера (или просто ассемблер, англ. *Assembly Language*) - язык программирования, прямо соответствующий ISA, но понятный человеку

RISC-V



- Пятое поколение архитектур набора команд RISC, созданое в 2010 году исследователями из калифорнийского университета в Беркли
- Спецификация ISA доступна для свободного и бесплатного использования - Linux в мире архитектур
- Предназначена для использования как в коммерческих, так и академических целях
- Поддерживается общая растущая программная экосистема
- Архитектура имеет стандартную версию, а также несколько расширений системы команд
- Подходит для вычислительных систем всех уровней: от микроконтроллеров до суперкомпьютеров
- Стандарт поддерживается некоммерческой организацией “RISC-V Foundation”, которая работает в тесном партнерстве с “The Linux Foundation”

Переменные в ассемблере - регистры

- В отличие от языков высокого уровня в ассемблере отсутствуют переменные
- Вместо переменных команды оперируют с регистрами
 - ограниченный набор ячеек хранения чисел, встроенных прямо в аппаратуру
 - в архитектурах RISC арифметические операции могут выполняться только с регистрами
 - с памятью возможны только операции записи и считывания (в отличие от CISC)
- Преимущество работы с регистрами - скорость доступа к ним
- Недостатки - ограниченное число регистров - 32 в RISC-V

Регистры RISC-V

- 32 регистра для основного набора команд
- каждый регистр имеет размер 32 бита = слово (**word**)
- x0 всегда равен 0
- 32 регистра для вещественных операций в расширении “F”
- в версия RV64 регистры имеют размер 64 бита (**double word**)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Команды RISC-V

- Каждая команда имеет код операции (opcode) и операнды

add x1, x2, x3 # $x_1 = x_2 + x_3$

add - код операции - сложение

x1 - регистр результата

x2, x3 - регистры-операнды

- используется в ассемблере для комментариев

- Эквивалент в языке С:

$$a = b + c$$

$$x_1 \Leftrightarrow a, x_2 \Leftrightarrow b, x_3 \Leftrightarrow c$$

Арифметические команды

- **add** $rd, rs1, rs2$
 - сложение $rd = rs1 + rs2$
- **sub** $rd, rs1, rs2$
 - вычитание $rd = rs1 - rs2$
- **and** $rd, rs1, rs2$
 - побитовое И $rd = rs1 \& rs2$
- **or** $rd, rs1, rs2$
 - побитовое ИЛИ $rd = rs1 | rs2$
- **xor** $rd, rs1, rs2$
 - побитовое исключающее ИЛИ $rd = rs1 \text{ xor } rs2$
- **sll** $rd, rs1, rs2$
 - сдвиг влево $rd = rs1 \ll rs2$

Обращение к памяти

- 1 байт = 8 бит
- 4 байта = 1 слово (**word**)
- адреса в памяти - адреса в байтах
- в RISC-V байты в словах расположены в соответствии с **little endian**, т.е. байты с меньшим адресом расположены в младших битах (см. картинку)

Младший байт в слове

Память			
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

31:24 23:16 15:8 7:0

Команды обращения к памяти

- **lw rd, addr**
 - считывание слова в регистр rd из памяти по адресу addr
- **sw rd, addr**
 - запись слова из регистра rd в память по адресу addr
- также доступны обращения меньшими размерами:
 - halfword - 2 байта: **lh, sh**
 - byte - 1 байт: **lb, sb**
- адрес addr может быть указан несколькими способами, самый простой
 $addr = offset(r1)$

где r1 - регистр, содержащий адрес обращения

offset - дополнительное смещение, указанное в виде числа

lw x2, 4(x3) # считывание слова из адреса = x3 + 4

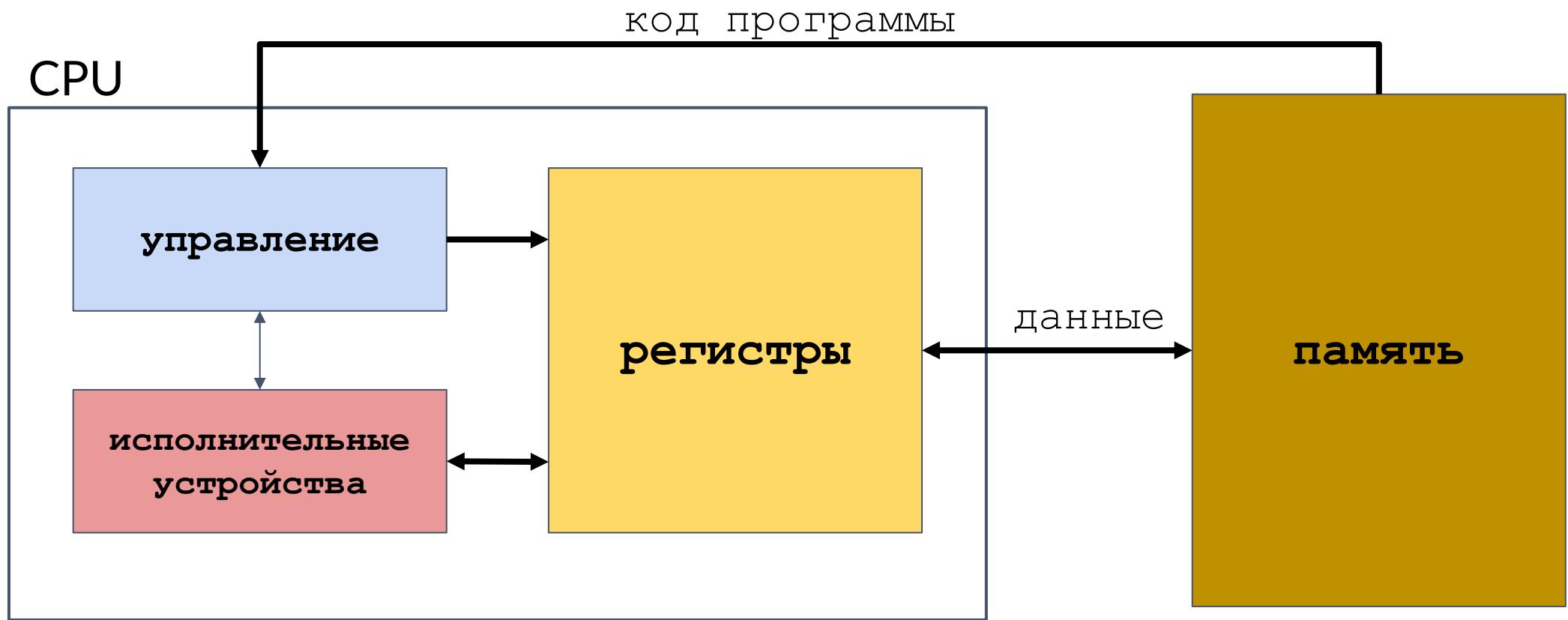
Числовые константы

- В коде ассемблера в качестве operandов можно использовать числа или непосредственные значения (**immediate operand**) или константы
- Для использования констант в архитектуре предусмотрены специальные команды:
 - **addi rd, rs1, imm**
 - сложение $rd = rs1 + imm$, например
 - **addi x2, x3, -4 # x2 = x3 - 4**
 - **li rd, imm**
 - Запись константы в регистр $rd = rs1$
- числа по умолчанию указываются в десятичной системе и со знаком
- для использований шестнадцатеричных чисел нужно добавить “0x”, например, $0x10 = 16$

Ветвления в программе. Переходы

- Ветвления подразумевают, что в зависимости от результатов некоторых вычислений нужно выполнять разные действия
- В языках программирования используется оператор if
- Аналог оператора if в ассемблере - операции условного перехода (*branch*)
- **beq rs1, rs2, label # branch if equal**
 - если $rs1 == rs2$, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду
- **bne rs1, rs2, label # branch if not equal**
 - если $rs1 != rs2$, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду
- Также есть безусловные переходы jump
- **j label**

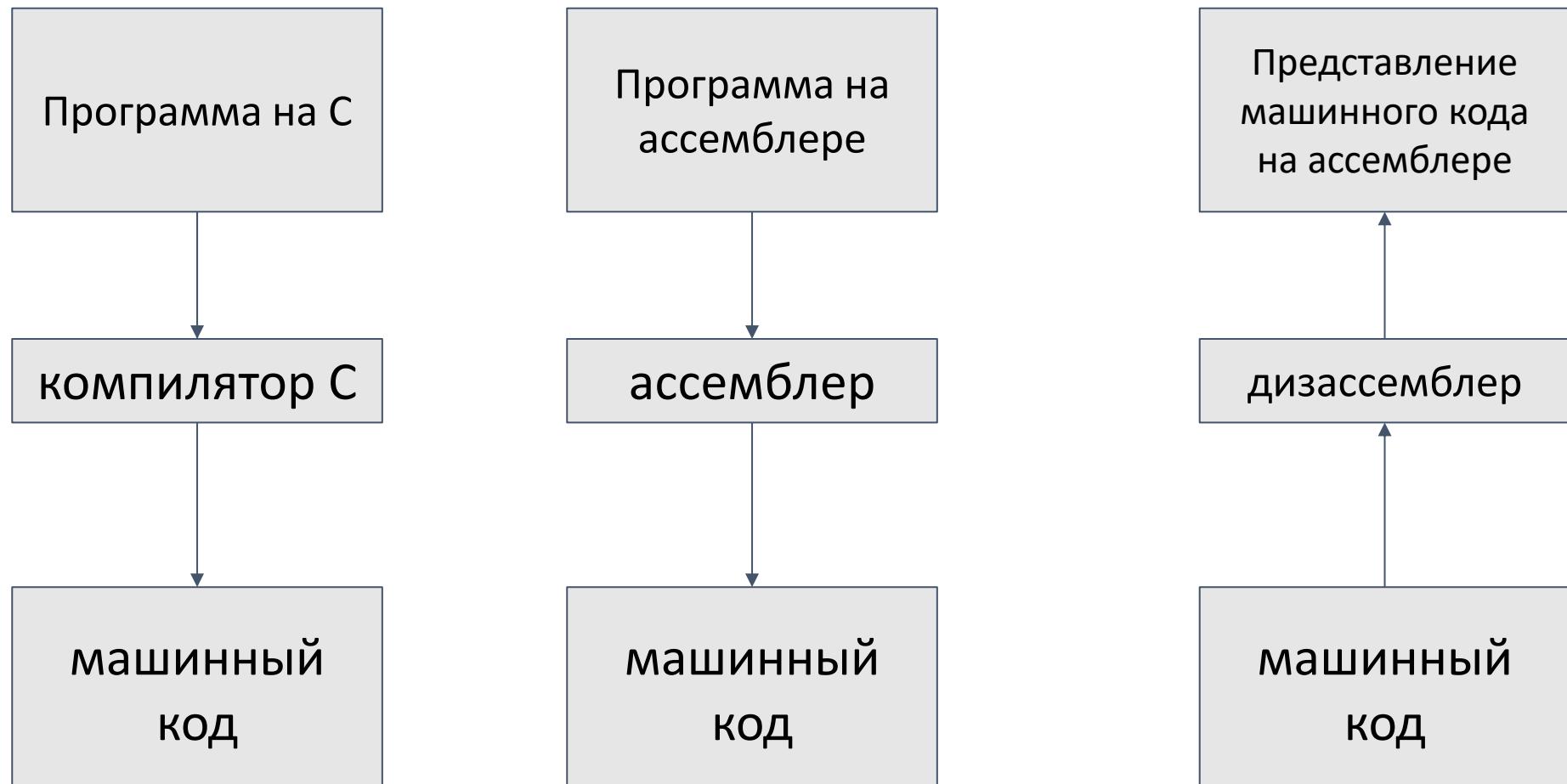
Архитектура с точки зрения программиста



Программирование на языке ассемблера

- Зачем программировать на ассемблере, если есть языки высокого уровня?
 - ассемблер до сих пор используется в системном ПО (например, ОС), чтобы получить доступ к специальным аппаратным ресурсам
 - ассемблер используется при разработке аппаратуры:
 - для написания тестовых программ
 - для изучения особенностей работы аппаратуры при выполнении программ используют дизассемблирование, т.е. получение из двоичного кода ассемблерной программы

Программирование на языке ассемблера



Псевдоинструкции

- Команды ассемблера, которые упрощают читаемость, но при сборке в двоичный код заменяются на другие
- **nop** $\Leftrightarrow \text{addi } x0, x0, 0$ # no operation
- **mv rd, rs** $\Leftrightarrow \text{addi } rd, rs, 0$ # copy register
- **beqz rs, offset** $\Leftrightarrow \text{beq } rs, x0, \text{offset}$ # branch if = zero

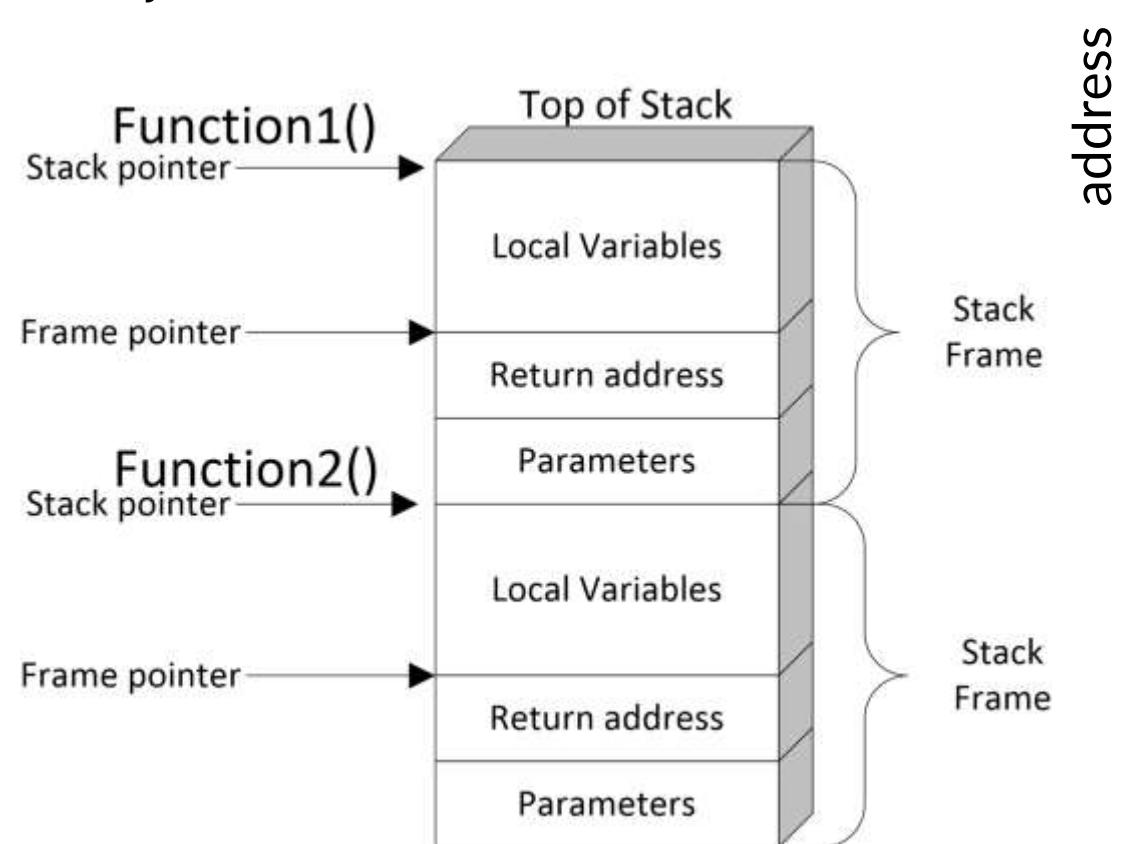
Стэк вызова

Стек вызовов (call stack) — структура данных, хранящая информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания

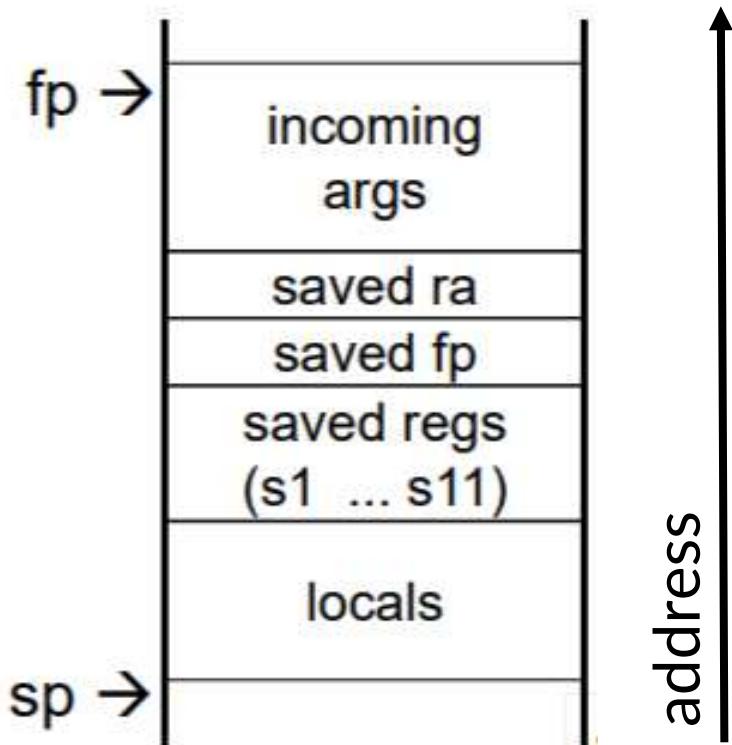
Стек обычно хранится в памяти и в нем сохраняются:

- аргументы вызванной функции (если не помещаются в регистрах)
- адрес возврата или другие указатели
- локальные переменные самой функции (если не помещаются в регистрах)
- значения регистров вызывающей функции

```
int function2(int a, ...)  
{  
    function1(b,c, ...);  
}
```



Регистры стэка вызова



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Команды JAL, JALR, RET

- Команды JAL/JALR выполняют безусловный переход с сохранением текущего PC для возможности возврата в тот же участок программы
- **jal rd, label** # jump and link
 - сделать переход на участок кода, помеченный *label* (immediate address), и записать *(pc+4)* в регистр *rd*
- **jalr rd, offset(rs1)** # jump and link register
 - сделать переход на участок кода по адресу *(rs1 + offset)*, и записать *(pc+4)* в регистр *rd*
- **ret** # return from subroutine: **jalr x0, x1, 0**

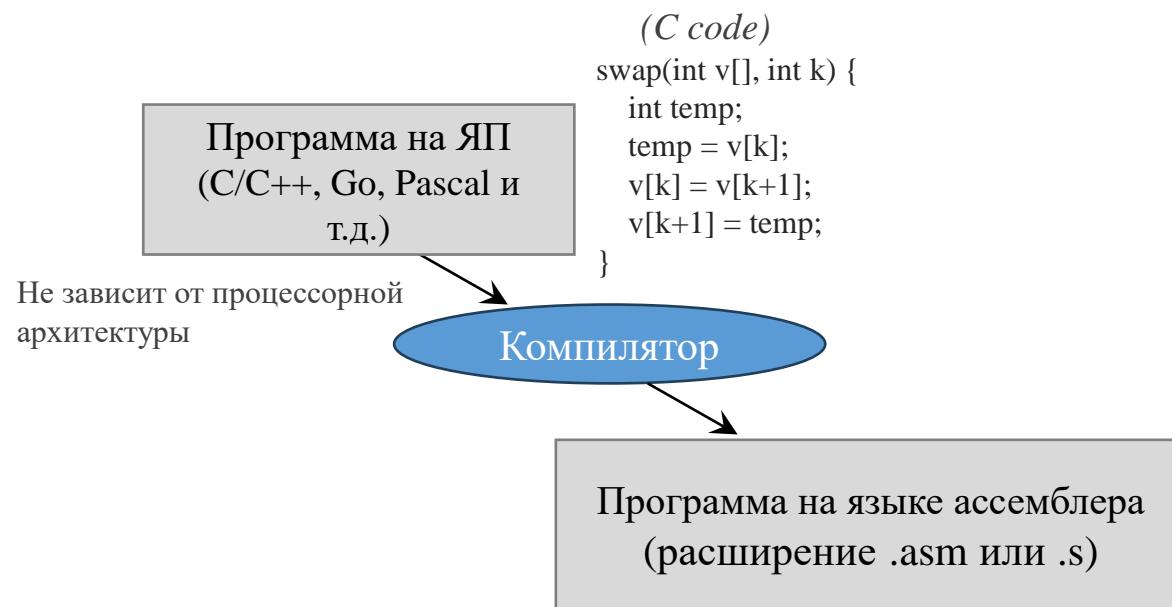
Преобразование высокоуровневых программ в машинный код

Программа на ЯП
(C/C++, Go, Pascal и
т.д.)

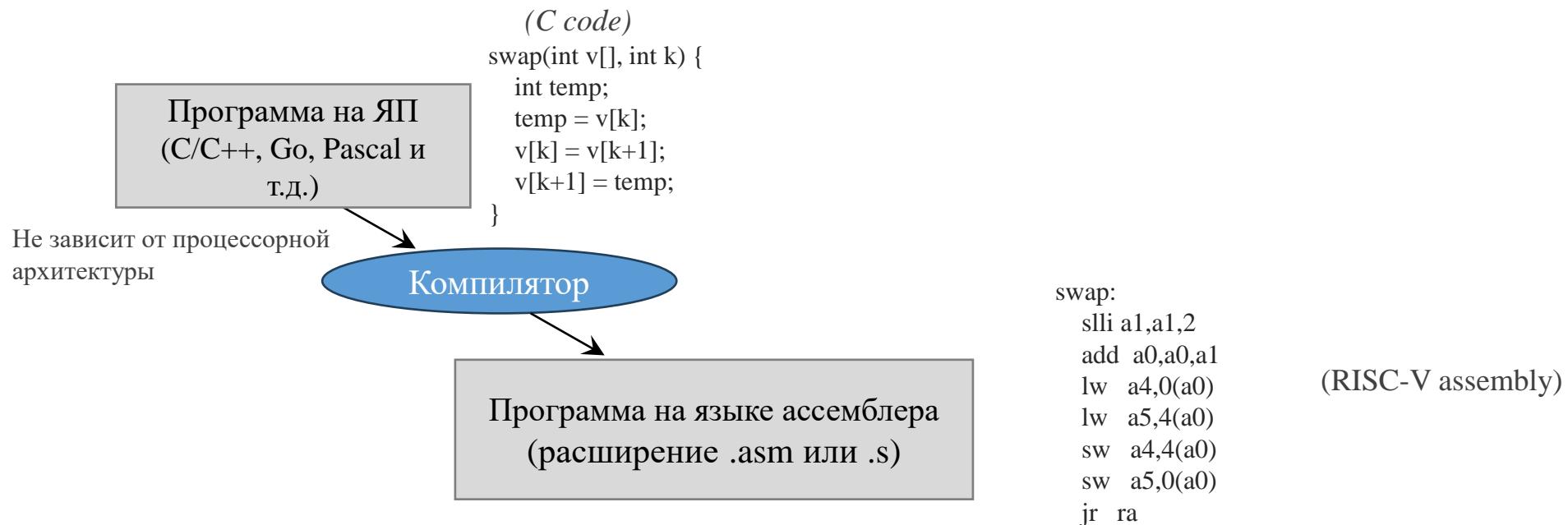
(C code)
swap(int v[], int k) {
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}

Не зависит от процессорной
архитектуры

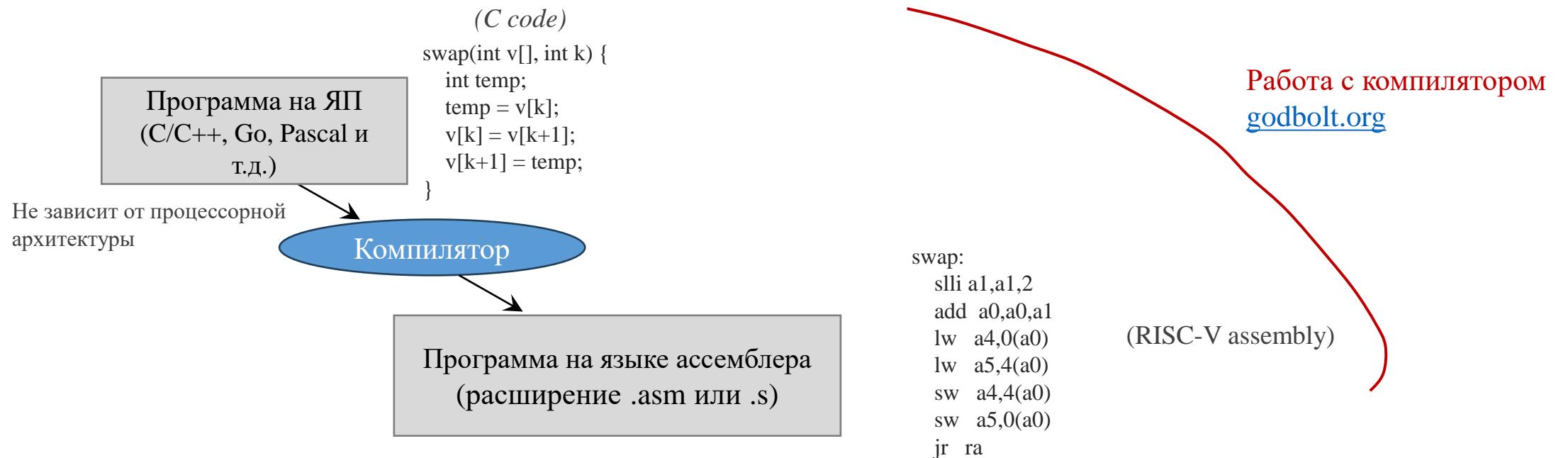
Преобразование высокоуровневых программ в машинный код



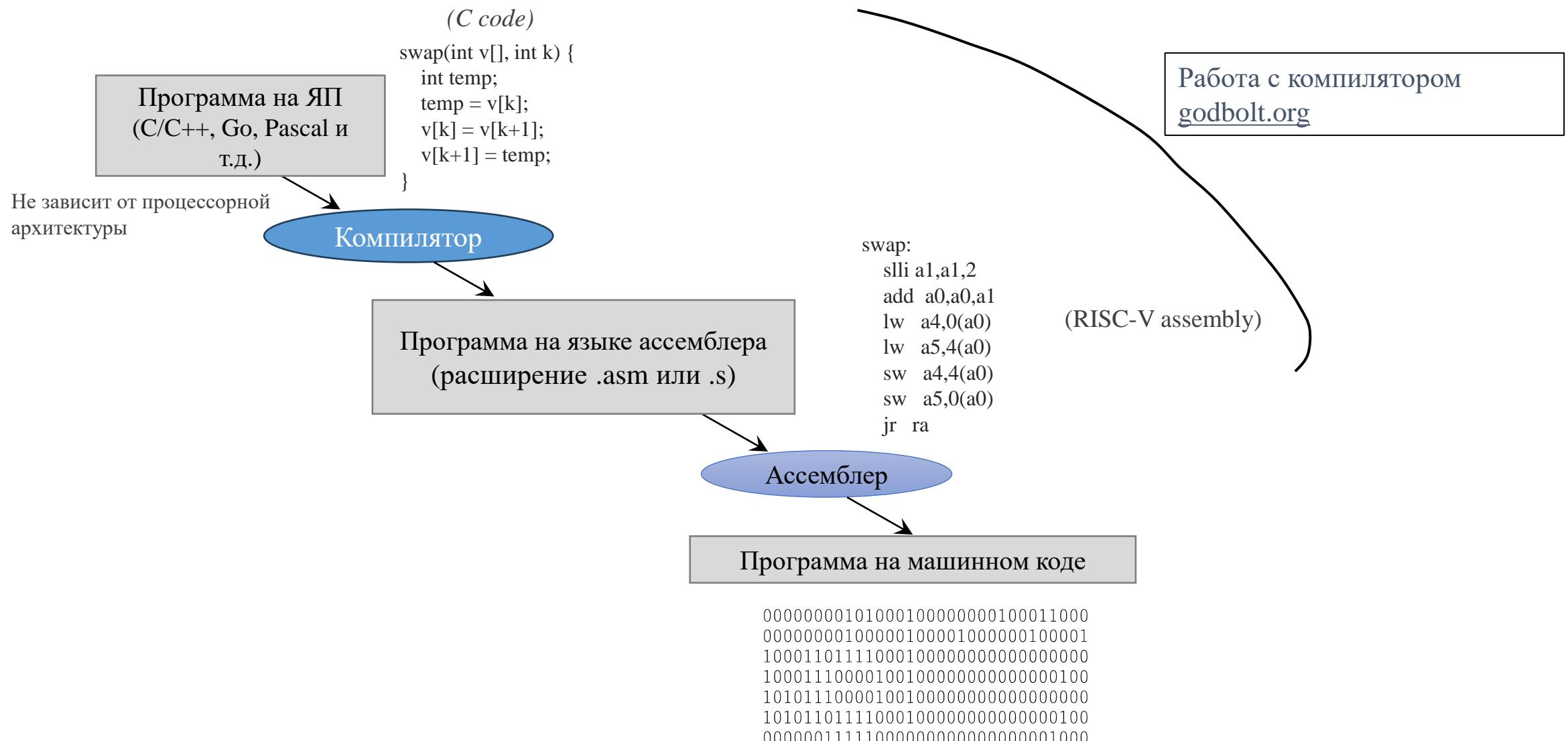
Преобразование высокоуровневых программ в машинный код



Преобразование высокоуровневых программ в машинный код



Преобразование высокоуровневых программ в машинный код



Преобразование высокоуровневых программ в машинный код

The screenshot shows the Godbolt compiler explorer interface with two code editors side-by-side.

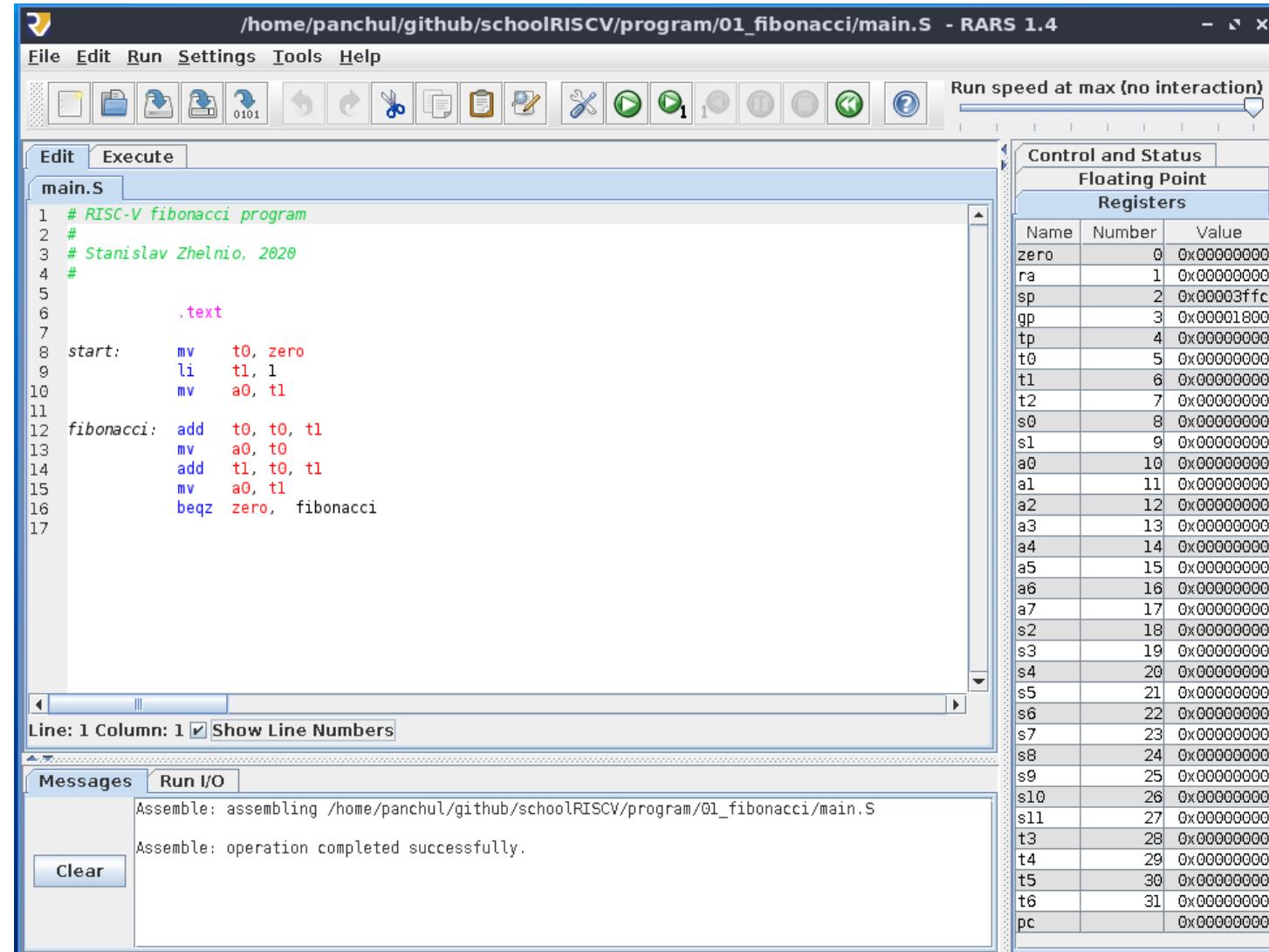
Left Editor (C source #1):

```
1 #include <stdio.h>
2 // Type your code here, or load an example.
3 int square(int num) {
4     return num * num;
5 }
6
7 int sum(int a, int b){
8     return a + b;
9 }
10
11 void swap(int v[], int k) {
12     int temp;
13     temp = v[k];
14     v[k] = v[k+1];
15     v[k+1] = temp;
16 }
```

Right Editor (RISC-V (32-bits) gcc 13.2.0 (Editor #1)):

```
1 square:
2     mul    a0,a0,a0
3     ret
4 sum:
5     add    a0,a0,a1
6     ret
7 swap:
8     slli   a1,a1,2
9     add    a0,a0,a1
10    lw     a4,0(a0)
11    lw     a5,4(a0)
12    sw     a4,4(a0)
13    sw     a5,0(a0)
14    ret
```

Симулятор assembler RISC-V



Практика. Использование assembler RISC-V

45 минут

- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- Описание комбинационной и последовательностной логики на языке Verilog
- Введение в архитектуру RISC-V.
- **Практика. Использование assembler RISC-V(45 минут)**
- Введение в микроархитектуру RISC-V.
- Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Введение в микроархитектуру RISC-V

- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- Описание комбинационной и последовательностной логики на языке Verilog
- Введение в архитектуру RISC-V.
- Практика. Использование assembler RISC-V(45 минут)
- **Введение в микроархитектуру RISC-V.**
- Использование assembler RISC-V и отладка кода с помощью ПЛИС.

Что такое schoolRISCV?

- Учебное ядро изначально разработанное Станиславом Жельнио
 - github.com/zhelnio/schoolRISCV
- Оригинально написано на Verilog
- Проект появился из предшественника schoolMIPS
- Реализует минимальный набор инструкций:
 - add, or, srl, sltu, sub, addi, lui, beq, bne

Что такое schoolRISCV?

- Учебное ядро изначально разработанное Станиславом Жельнио
 - github.com/zhelnio/schoolRISCV
- Оригинально написано на **Verilog**
- Проект появился из предшественника schoolMIPS
- Реализует минимальный набор инструкций: **add, or, srl, sltu, sub, addi, lui, beq, bne**
- Однотактная реализация
- Адресация памяти команд по словам (4-байта)
- Отсутствует память данных (инструкции load/store)
- Частично соответствует стандарту архитектуры RISC-V (RV32I)

Поддерживает только 9 инструкций из необходимых 40 в стандарте (Учебный проект), и этого уже достаточно, чтобы посчитать квадратный корень.

Основные компоненты и файлы

- Тракт данных
- Счетчик команд **PC**
- Память инструкций **Instruction Memory**
- Декодер инструкций **Instruction Decoder**
- Регистровый файл **Register File**
- Арифметико-логическое устройство **ALU**
- Сумматоры адреса **pcPlus4** и **pcBranch**
- Мультиплексоры **pcSrc**, **wdSrc** и **aluSrc**
- Устройство управления
- **sr_cpu.svh** — заголовочный файл с константами
- **sr_cpu.sv** — основной модуль процессора
- **sr_decode.sv** — дешифратор инструкций и непосредственных значений (immediate)
- **sr_control.sv** — управляющий модуль
- **sr_alu.sv** — модуль арифметико-логического устройства
- **sr_register_file.sv** — модуль 32 регистров общего назначения

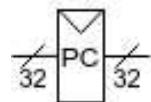
Что такое микроархитектура?

Микроархитектура – это аппаратная реализация архитектуры в виде логической схемы, состоящей из функциональных блоков, реализующих все команды.

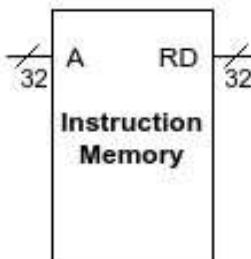
Возможны разные реализации одной архитектуры:

однотактная многотактная конвейерная

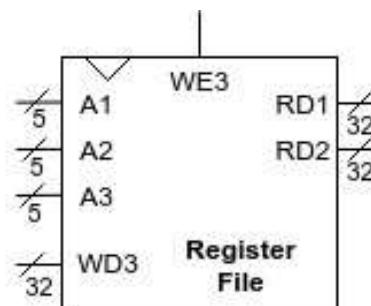
Архитектурное состояние системы определяют данные блоки:



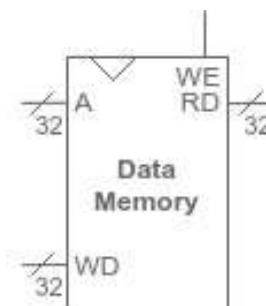
Счётчик команд



Память инструкций



Регистровый файл



Память данных

**The RISC-V Instruction Set Manual
Volume II: Privileged Architecture**
Document Version 20211203

Editors: Andrew Waterman¹, Krste Asanović^{1,2}, John Hauser

¹SiFive Inc.,

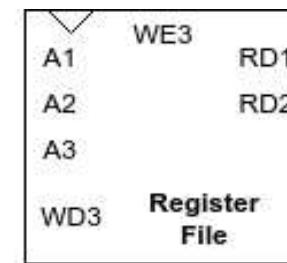
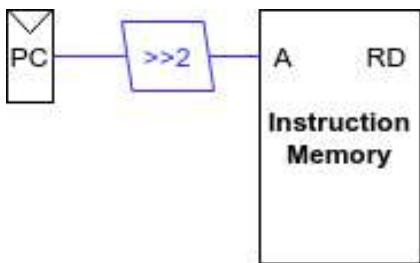
²CS Division, EECS Department, University of California, Berkeley

andrew@sifive.com, krste@berkeley.edu, jh.riscv@jhauser.us

December 4, 2021

<https://riscv.org/specifications/>

ADDI: выборка инструкции



ADDI: спецификация

Integer Register-Immediate Instructions

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM	
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd*, *rs1*, 0 is used to implement the MV *rd*, *rs1* assembler pseudoinstruction.

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (*inst*[*y*]) produces each bit of the immediate value.

31	30	20 19	12	11	10	5	4	1	0
— <i>inst</i> [31] —					<i>inst</i> [30:25]	<i>inst</i> [24:21]	<i>inst</i> [20]	I-immediate	

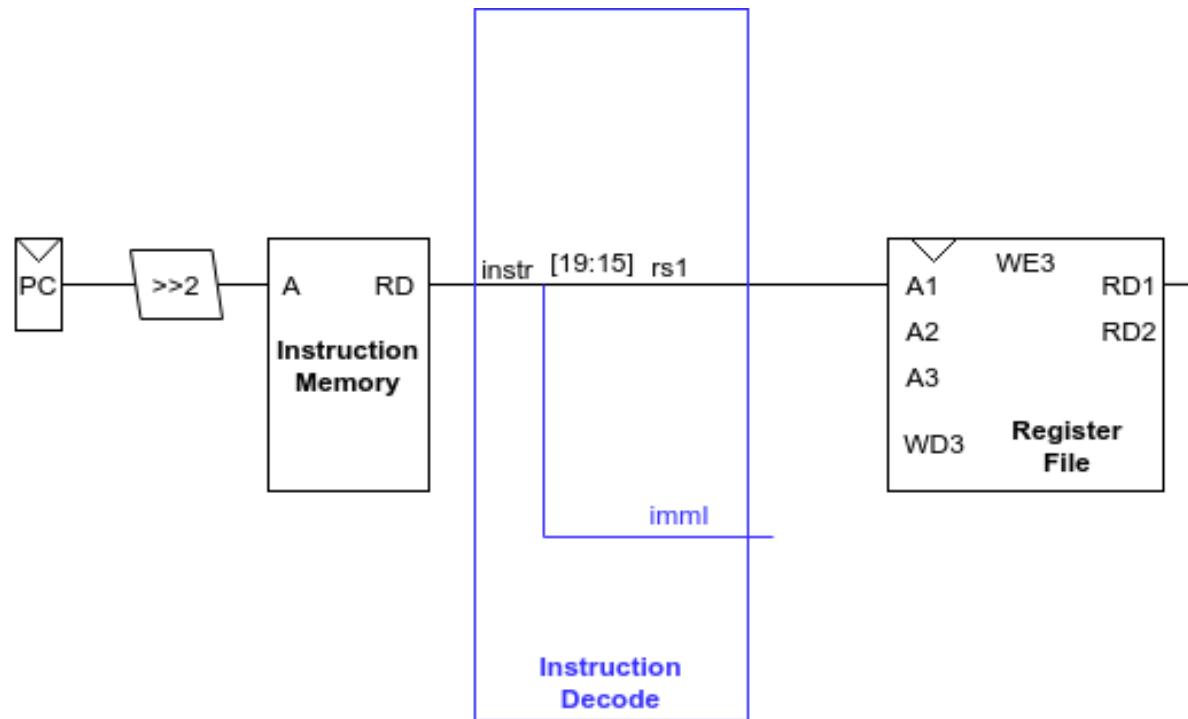
ADDI: считывание операнда из регистрового файла



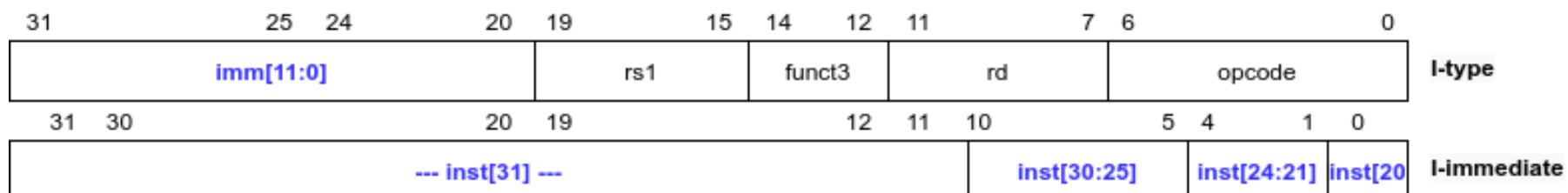
ADDI: I-type, adds the sign-extended 12-bit immediate to register rs1: $rd = rs1 + imm$

31	25	24	20	19	15	14	12	11	7	6	0	I-type
	imm[11:0]			rs1	funct3		rd		opcode			

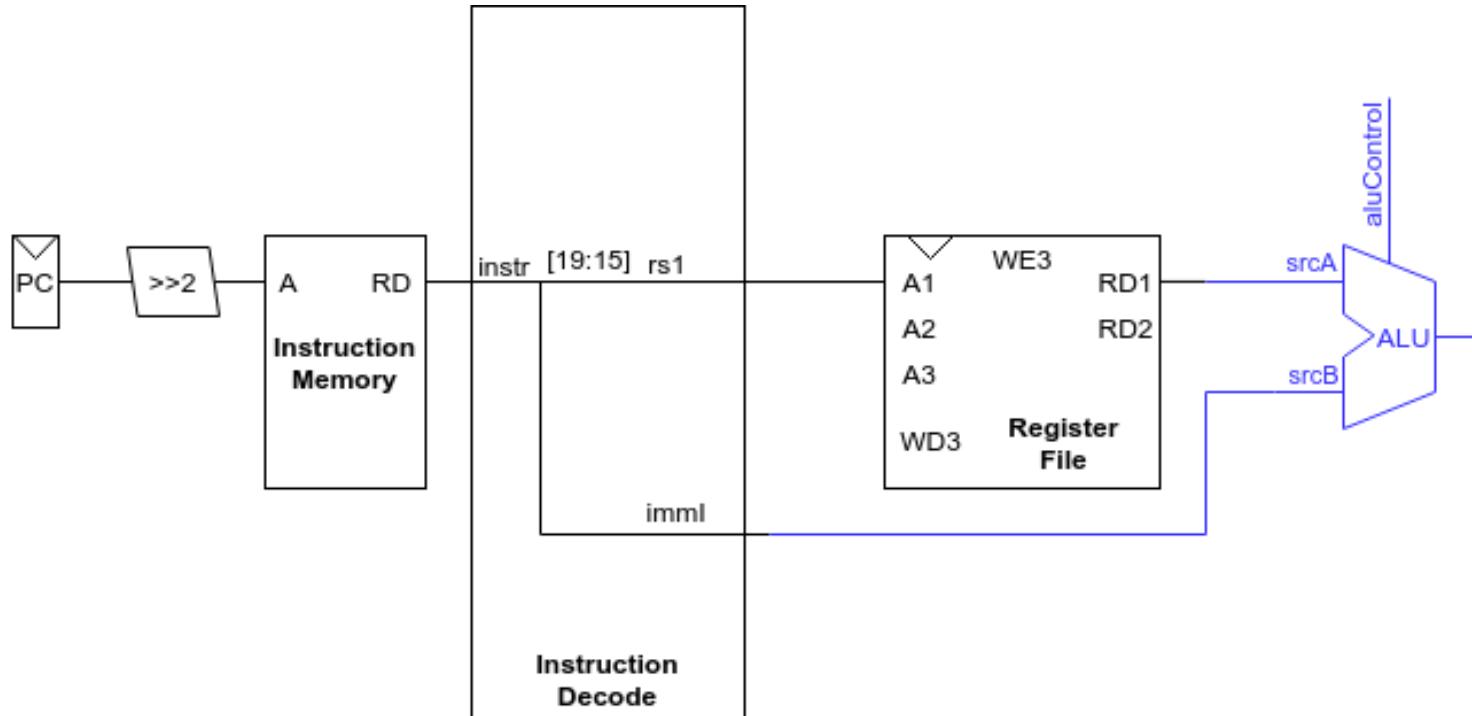
ADDI: декодирование константы из тела инструкции



ADDI: I-type, adds the sign-extended 12-bit immediate to register rs1: $rd = rs1 + imm12$



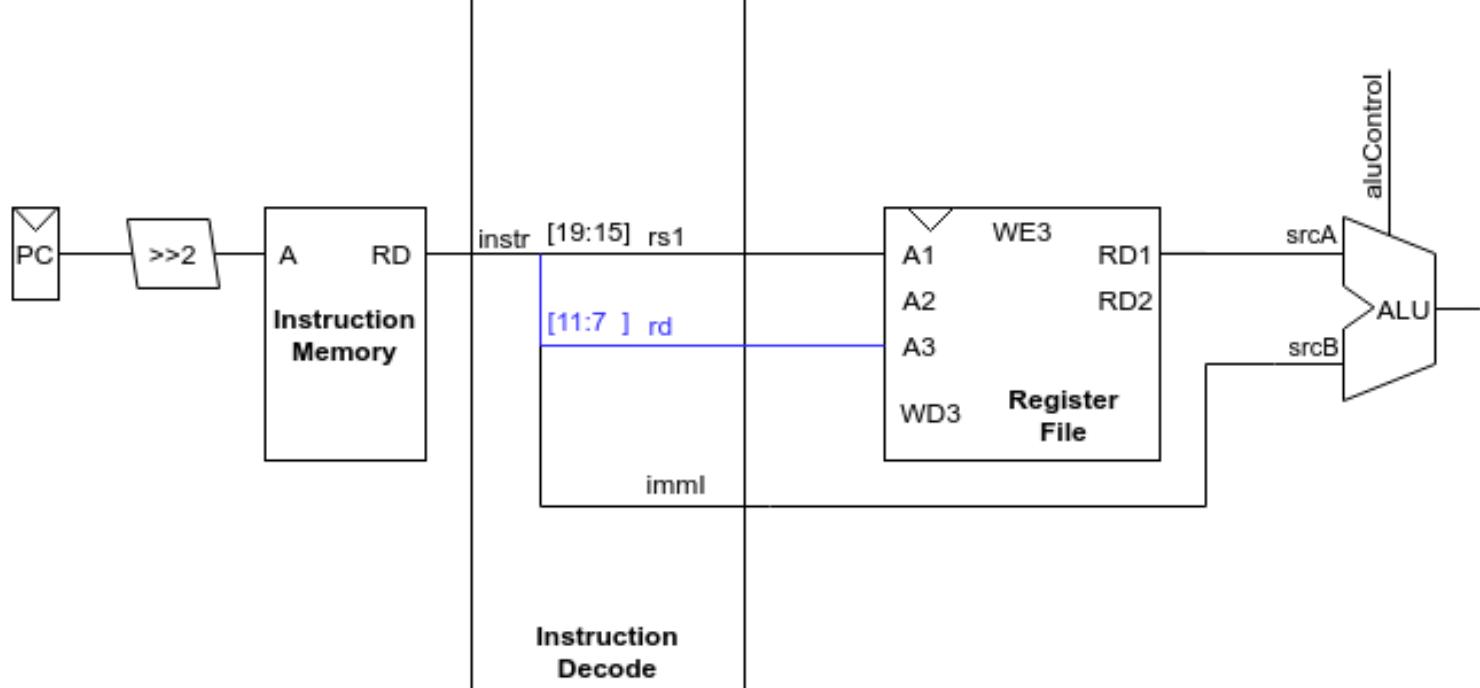
ADDI: вычисление результата арифметической операции



ADDI: I-type, adds the sign-extended 12-bit immediate to register rs1: $rd = rs1 + imm11[11:0]$

31	25	24	20	19	15	14	12	11	7	6	0	I-type
imm[11:0]				rs1	funct3		rd		opcode			

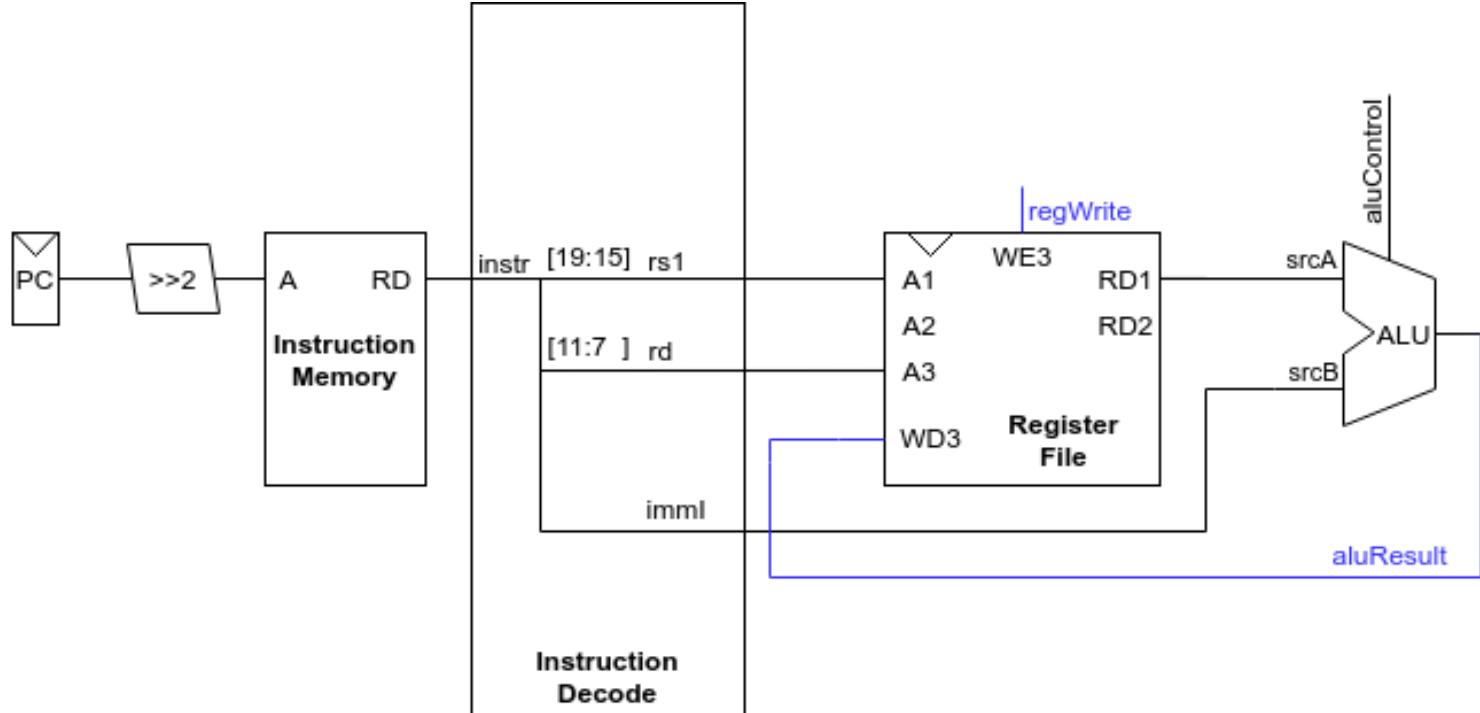
ADDI: декодирование регистра назначения



ADDI: I-type, adds the sign-extended 12-bit immediate to register rs1: $rd = rs1 + imm1$

31	25	24	20	19	15	14	12	11	7	6	0	I-type
imm[11:0]				rs1	funct3	rd			opcode			

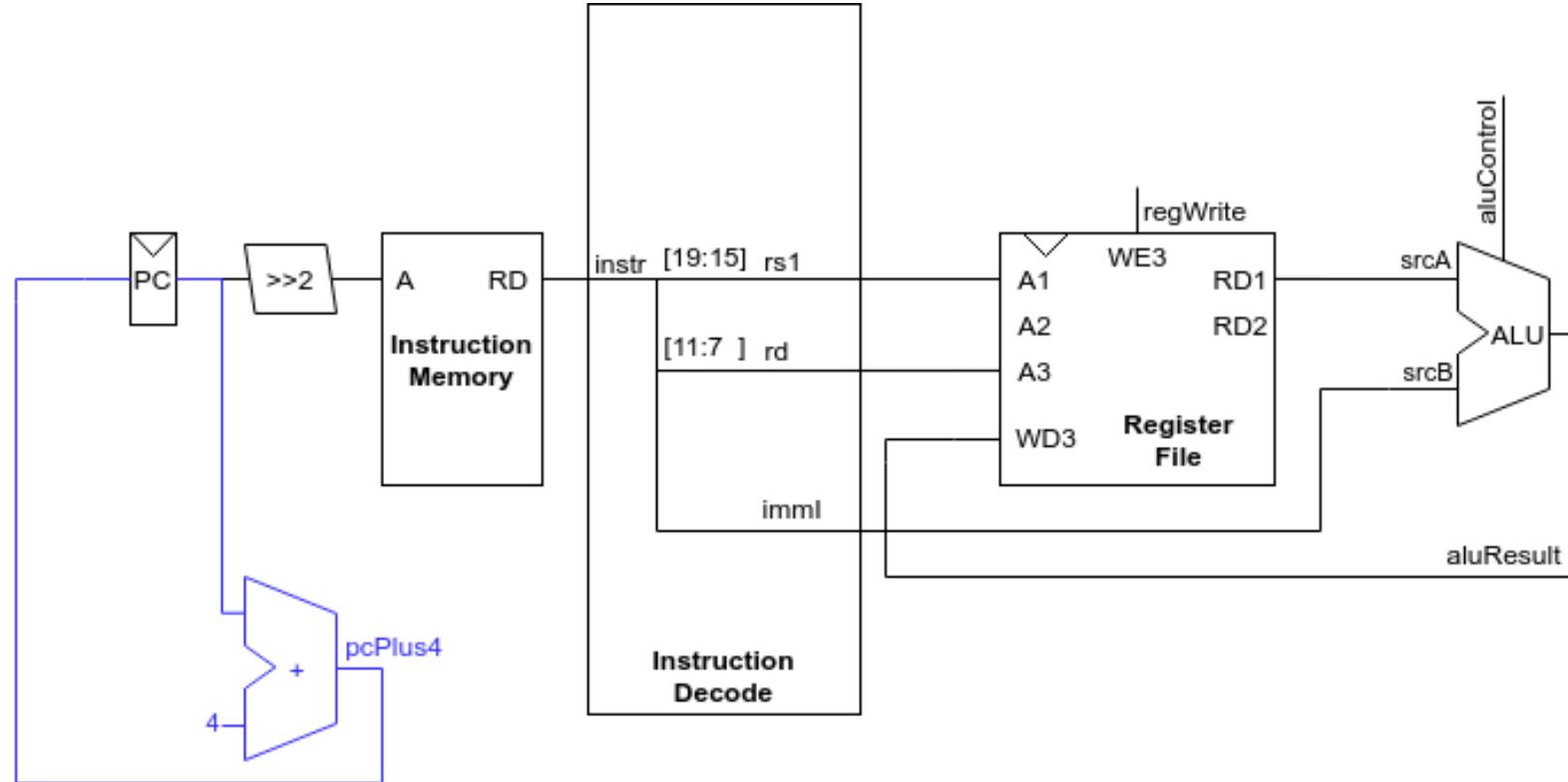
ADDI: запись результата в регистр назначения



ADDI: I-type, adds the sign-extended 12-bit immediate to register rs1: $rd = rs1 + imm1$

31	25	24	20	19	15	14	12	11	7	6	0	I-type
imm[11:0]			rs1	funct3		rd			opcode			

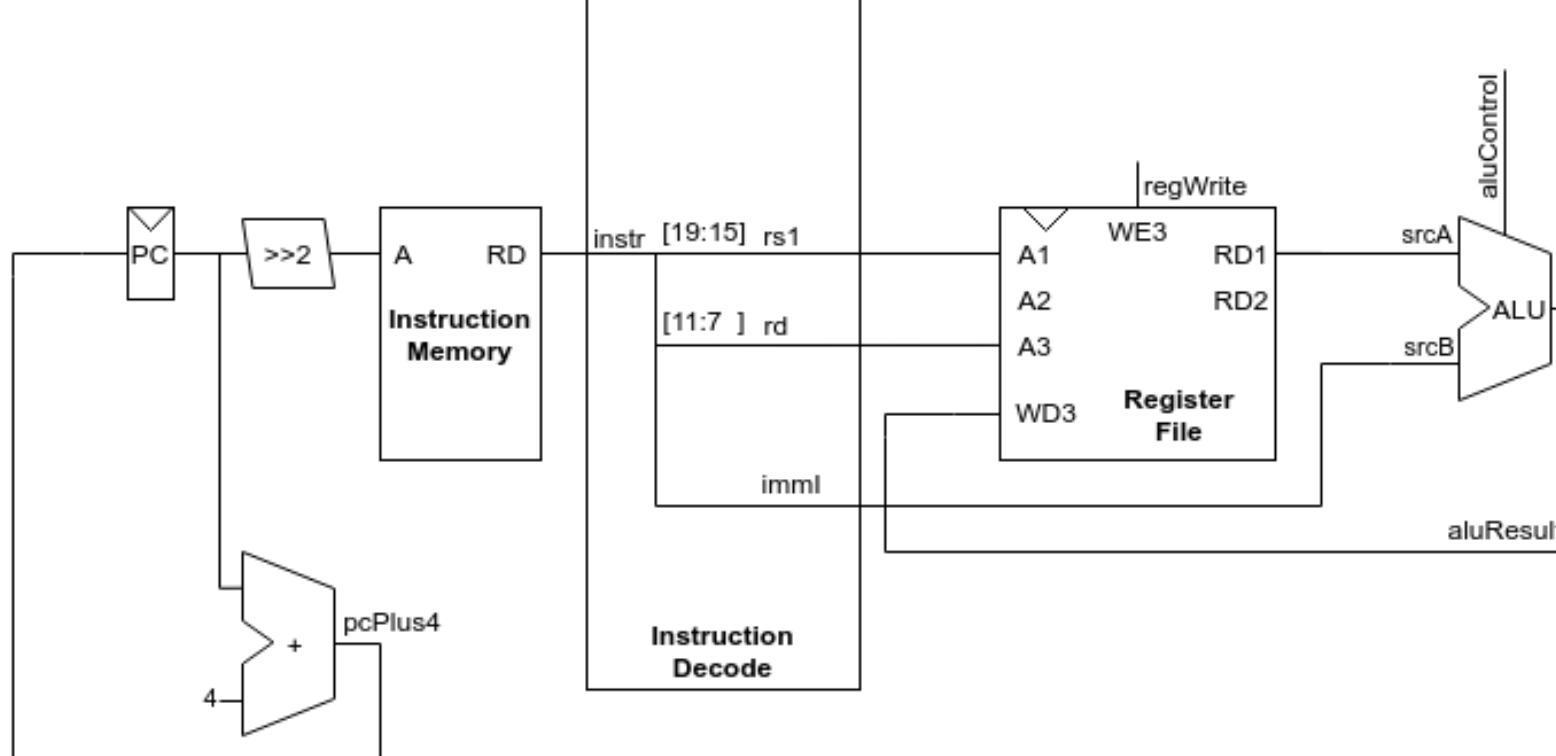
ADDI: вычисление адреса следующей инструкции



ADDI: I-type, adds the sign-extended 12-bit immediate to register $rs1$: $rd = rs1 + imm1$

31	25	24	20	19	15	14	12	11	7	6	0	I-type
imm[11:0]				rs1	funct3		rd		opcode			

ADDI: итоговая схема



ADDI: I-type, adds the sign-extended 12-bit immediate to register $rs1$: $rd = rs1 + imml$

31	25	24	20	19	15	14	12	11	7	6	0
	imm[11:0]			rs1	funct3		rd		opcode		

I-type

31	30	20	19	12	11	10	5	4	1	0
		---	inst[31]	---			inst[30:25]	inst[24:21]	inst[20]	

I-immediate

ADD: спецификация

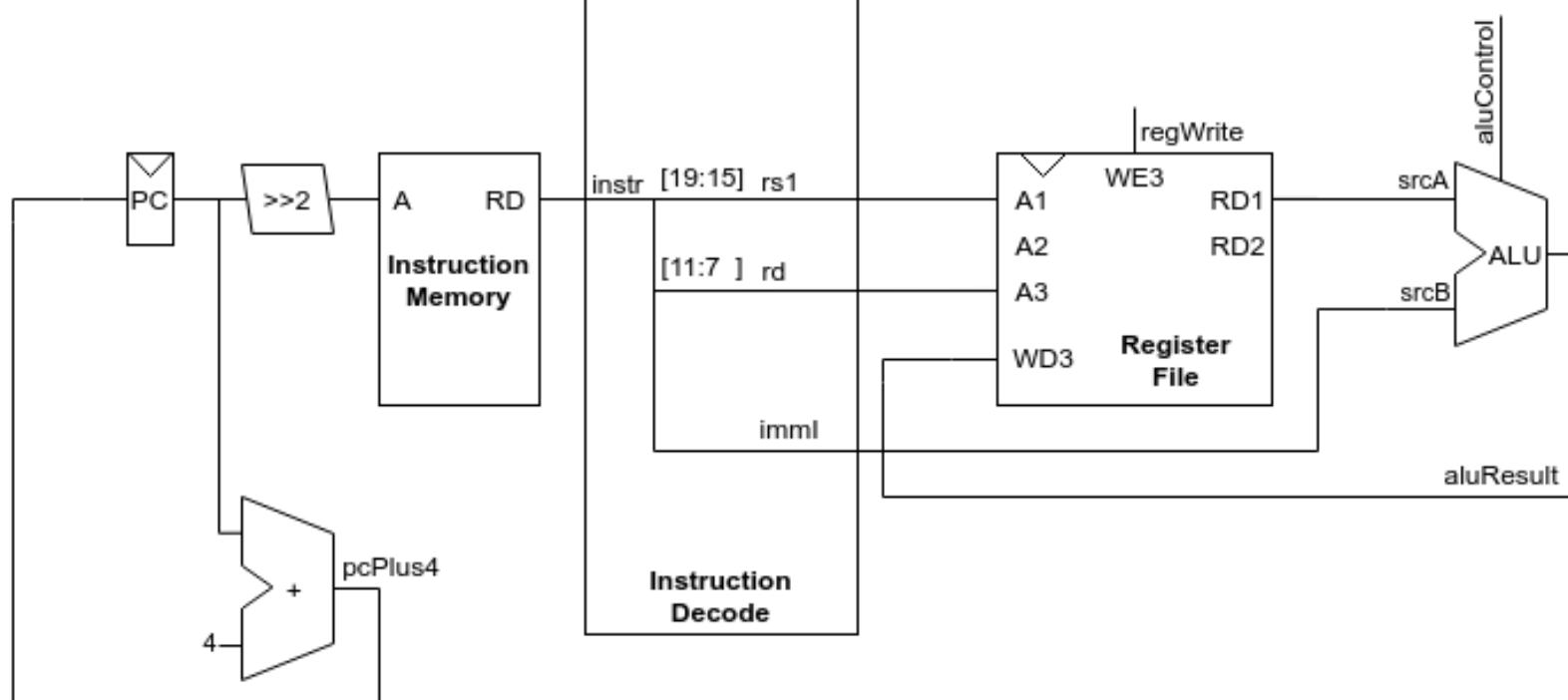
Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note,

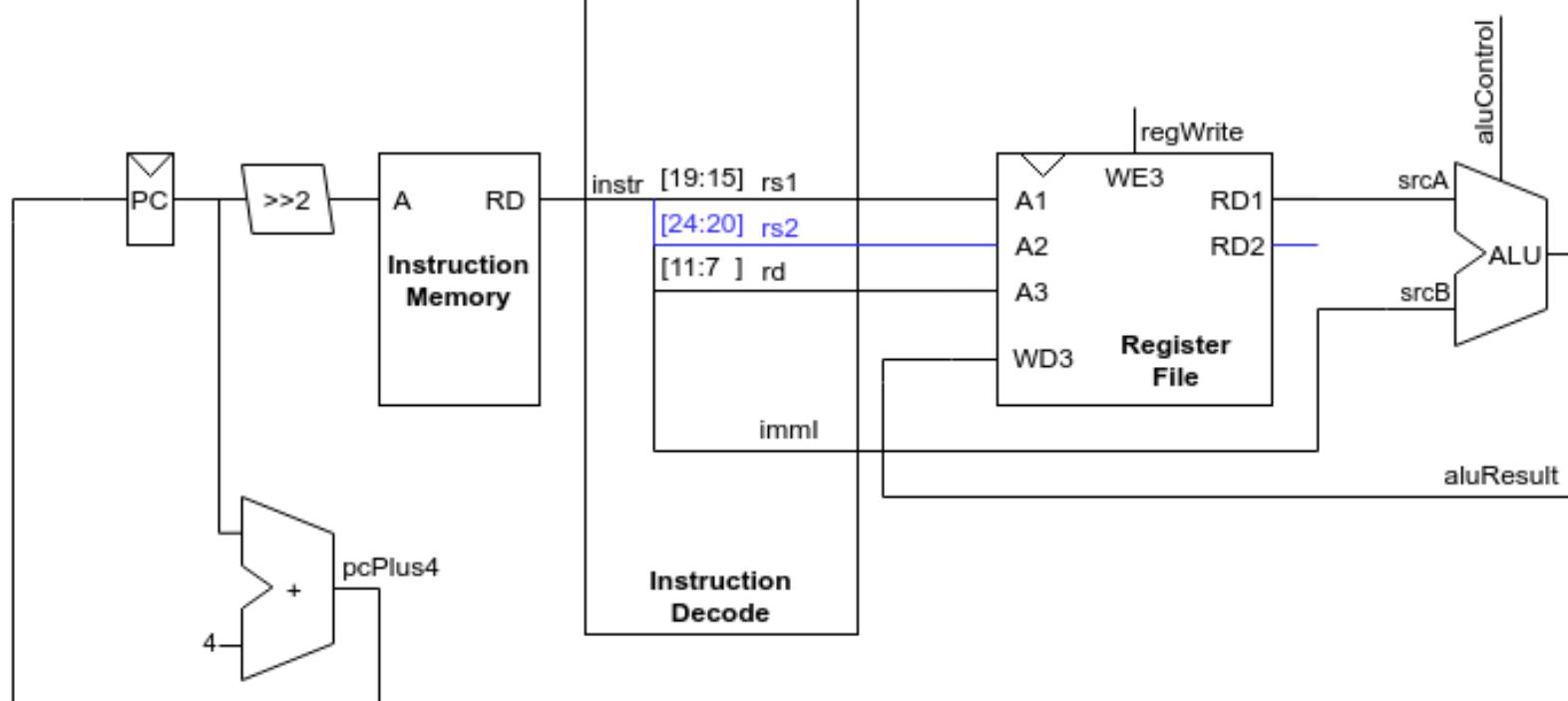
ADD: выборка операнда rs1



ADD: R-type, performs the addition of rs1 and rs2: $rd = rs1 + rs2$

31	25	24	20	19	15	14	12	11	7	6	0	R-type
funct7	rs2		rs1	funct3	rd				opcode			

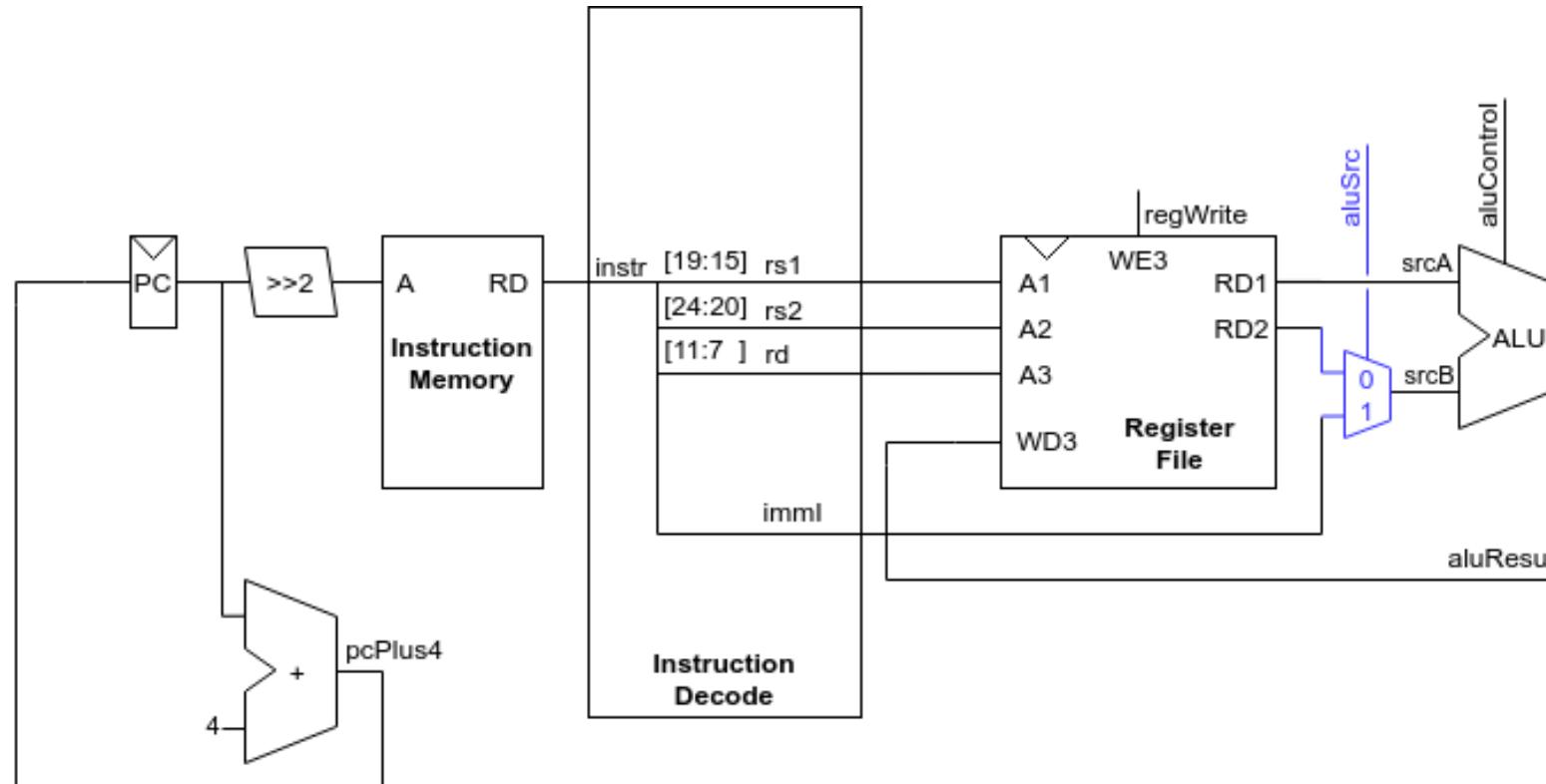
ADD: выборка операнда rs2



ADD: R-type, performs the addition of rs1 and rs2: $rd = rs1 + rs2$

31	25	24	20	19	15	14	12	11	7	6	0	R-type
funct7		rs2		rs1	funct3		rd		opcode			

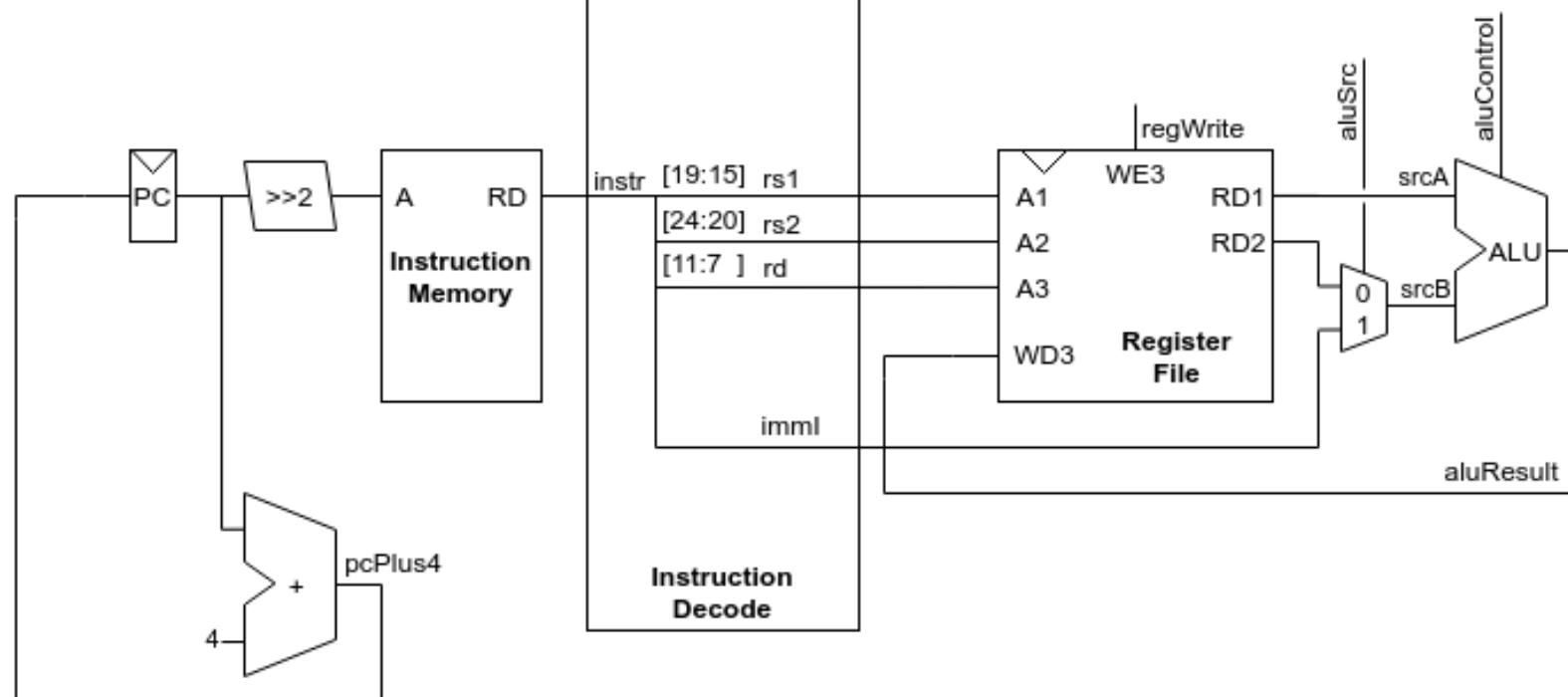
ADD: передача второго операнда в АЛУ



ADD: R-type, performs the addition of rs1 and rs2: $rd = rs1 + rs2$

31	25	24	20	19	15	14	12	11	7	6	0	R-type
funct7		rs2		rs1	funct3		rd		opcode			

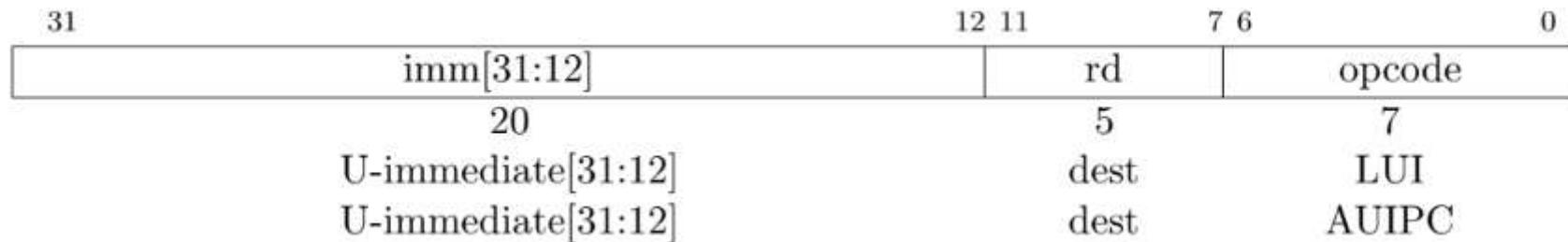
ADD: итоговая схема



ADD: R-type, performs the addition of rs1 and rs2: $rd = rs1 + rs2$

31	25	24	20	19	15	14	12	11	7	6	0	R-type
funct7		rs2		rs1	funct3		rd		opcode			

LUI: спецификация

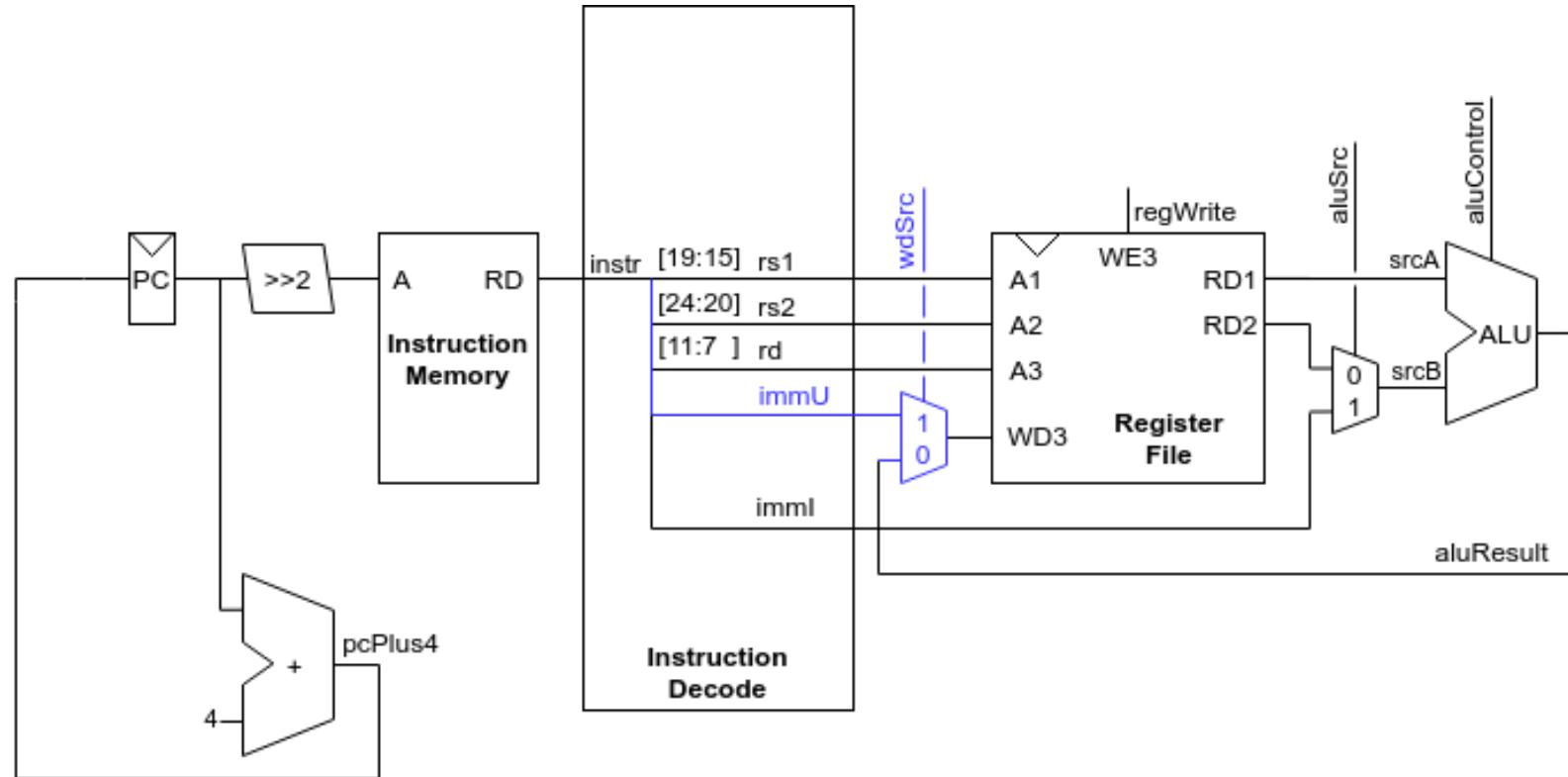


LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

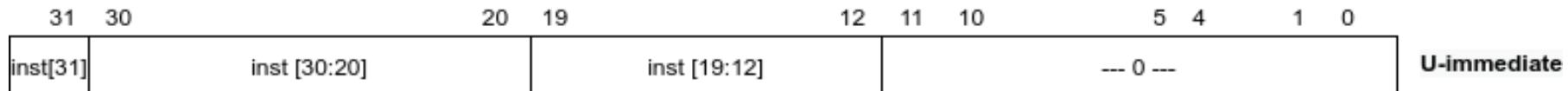
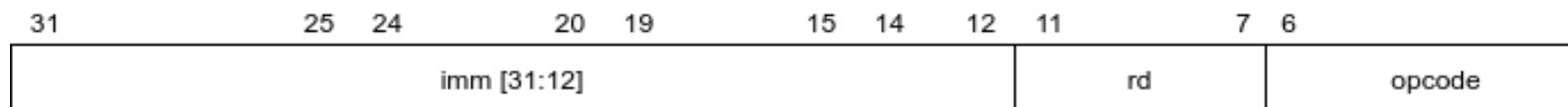
...

inst[31]	inst[30:20]	inst[19:12]	— 0 —	U-immediate
----------	-------------	-------------	-------	-------------

LUI: декодирование и передача константы



LUI: U-type, load upper immediate: $rd = immU$



BEQ: спецификация

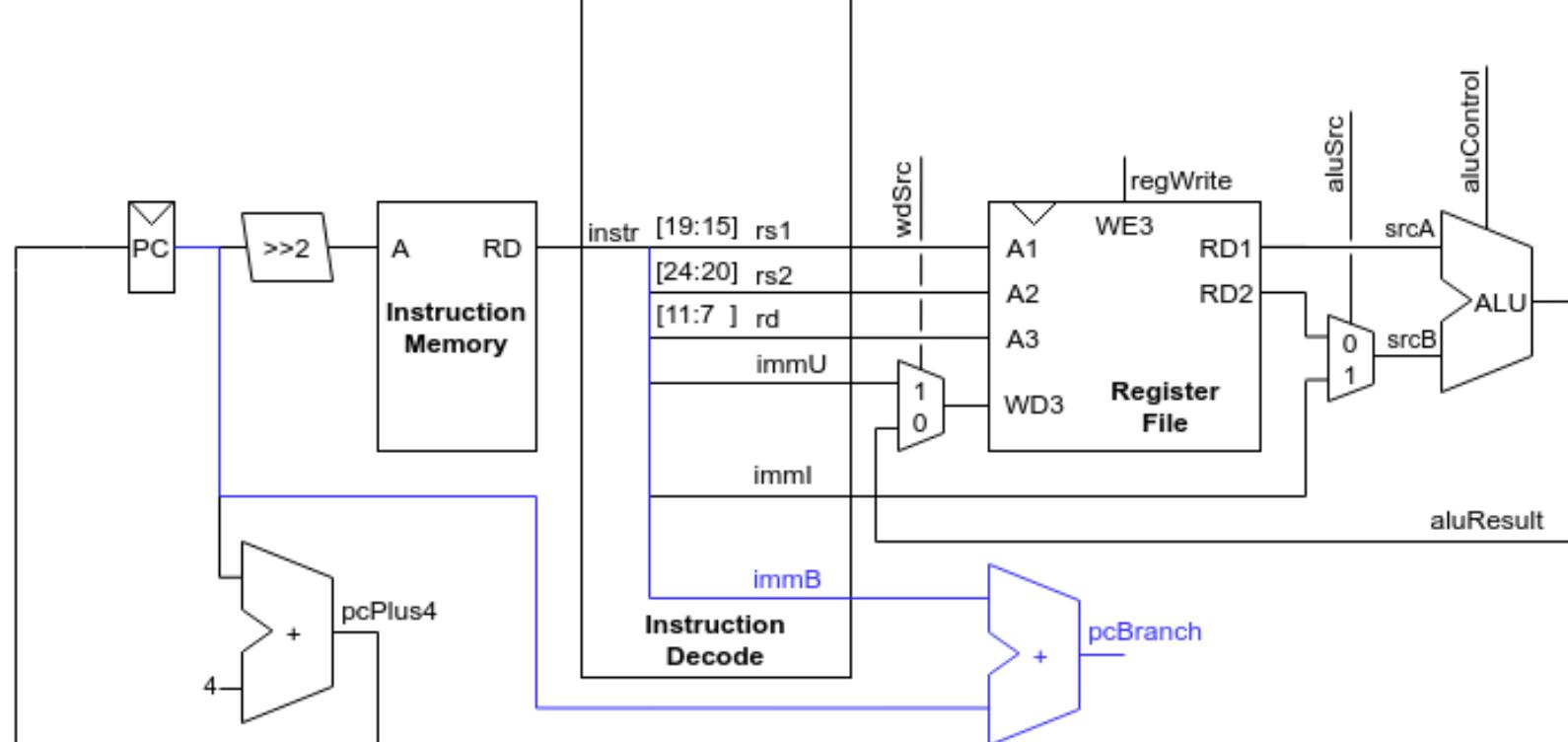
Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is $\pm 4\text{ KiB}$.

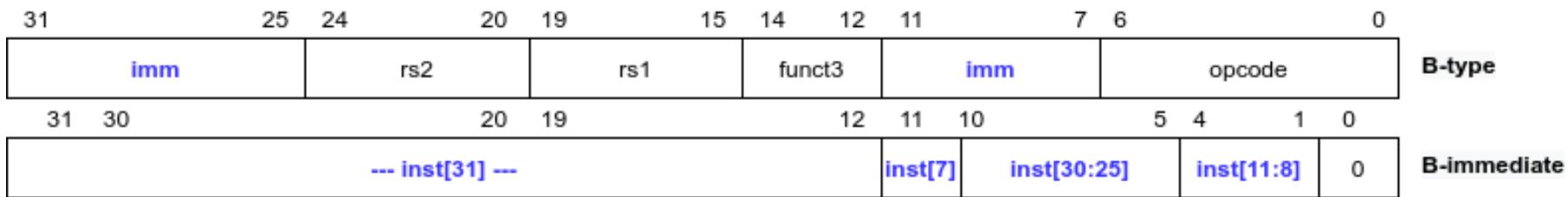
31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]		opcode	
1	6	5	5	3	4	1		7	
offset[12 10:5]		src2	src1	BEQ/BNE		offset[11 4:1]		BRANCH	
offset[12 10:5]		src2	src1	BLT[U]		offset[11 4:1]		BRANCH	
offset[12 10:5]		src2	src1	BGE[U]		offset[11 4:1]		BRANCH	

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

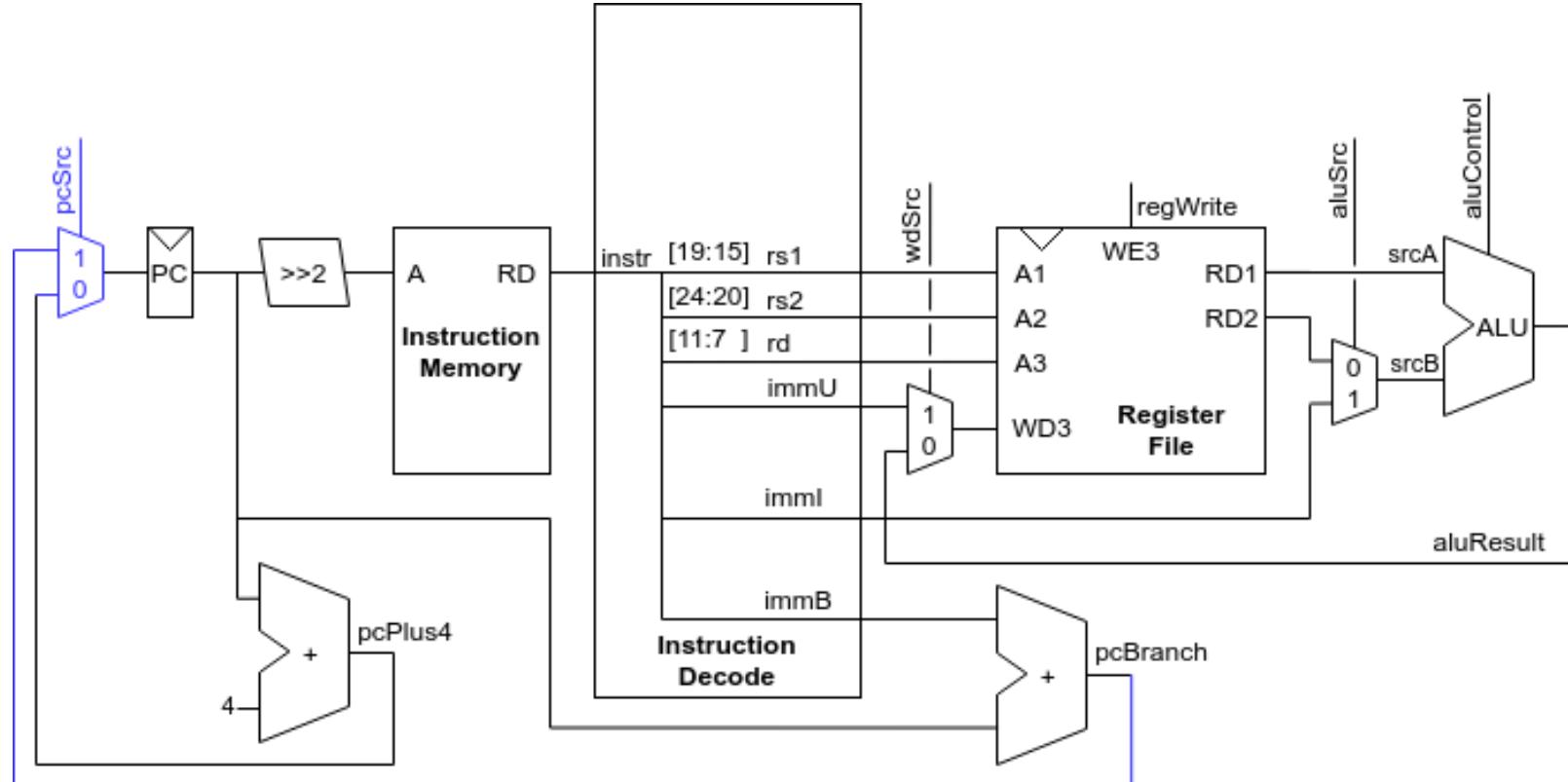
BEQ: вычисление адреса условного перехода



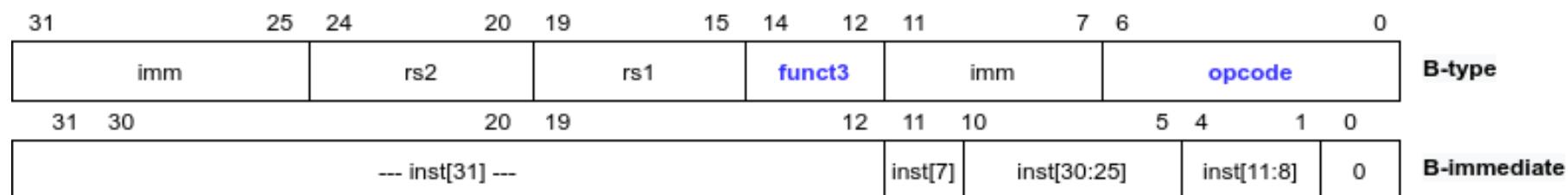
BEQ: B-type, branch on equal: if ($rs1 == rs2$) $PC = PC + immB$



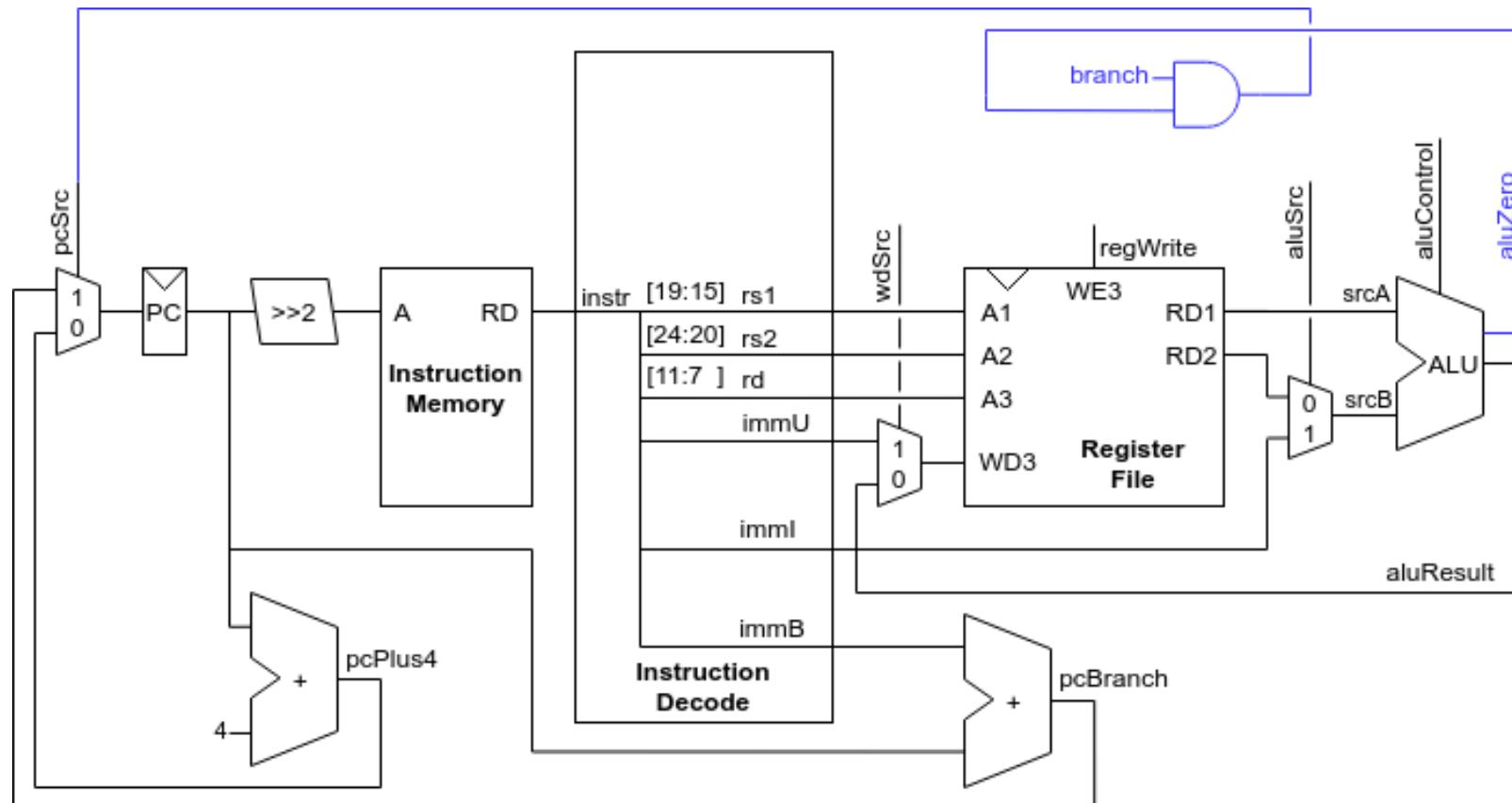
BEQ: выбор адреса



BEQ: B-type, branch on equal: if ($rs1 == rs2$) $PC = PC + immB$



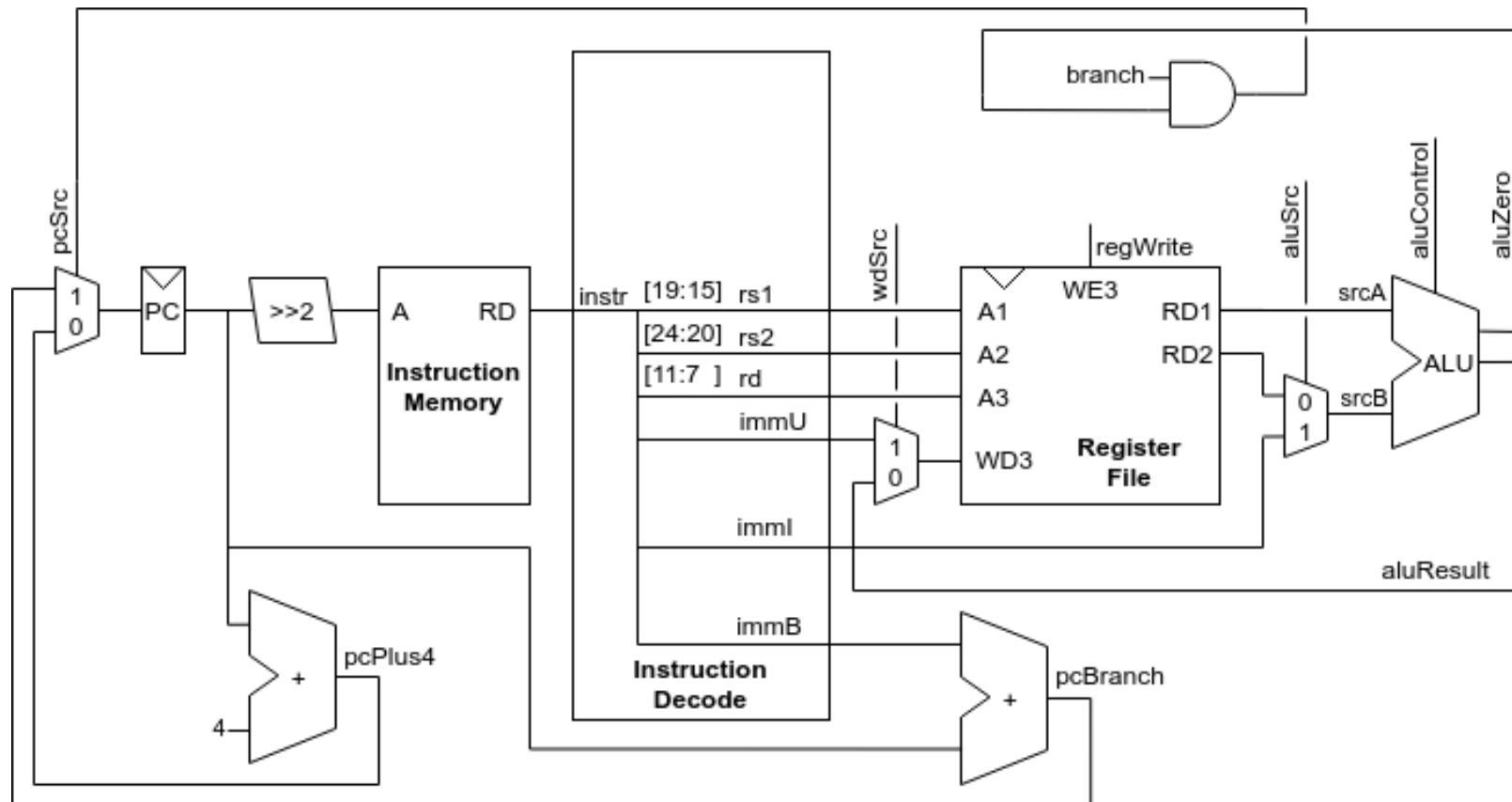
BEQ: определение необходимости перехода



BEQ: B-type, branch on equal: if ($rs1 == rs2$) $PC = PC + immB$

31	25	24	20	19	15	14	12	11	7	6	0	B-type	
imm	rs2		rs1		funct3		imm		opcode				
31 30	20 19		12 11 10		5 4	1	0					B-immediate	
--- inst[31] ---										inst[7]	inst[30:25]	inst[11:8]	0

BEQ: итоговая схема



BEQ: B-type, branch on equal: if ($rs1 == rs2$) $PC = PC + immB$

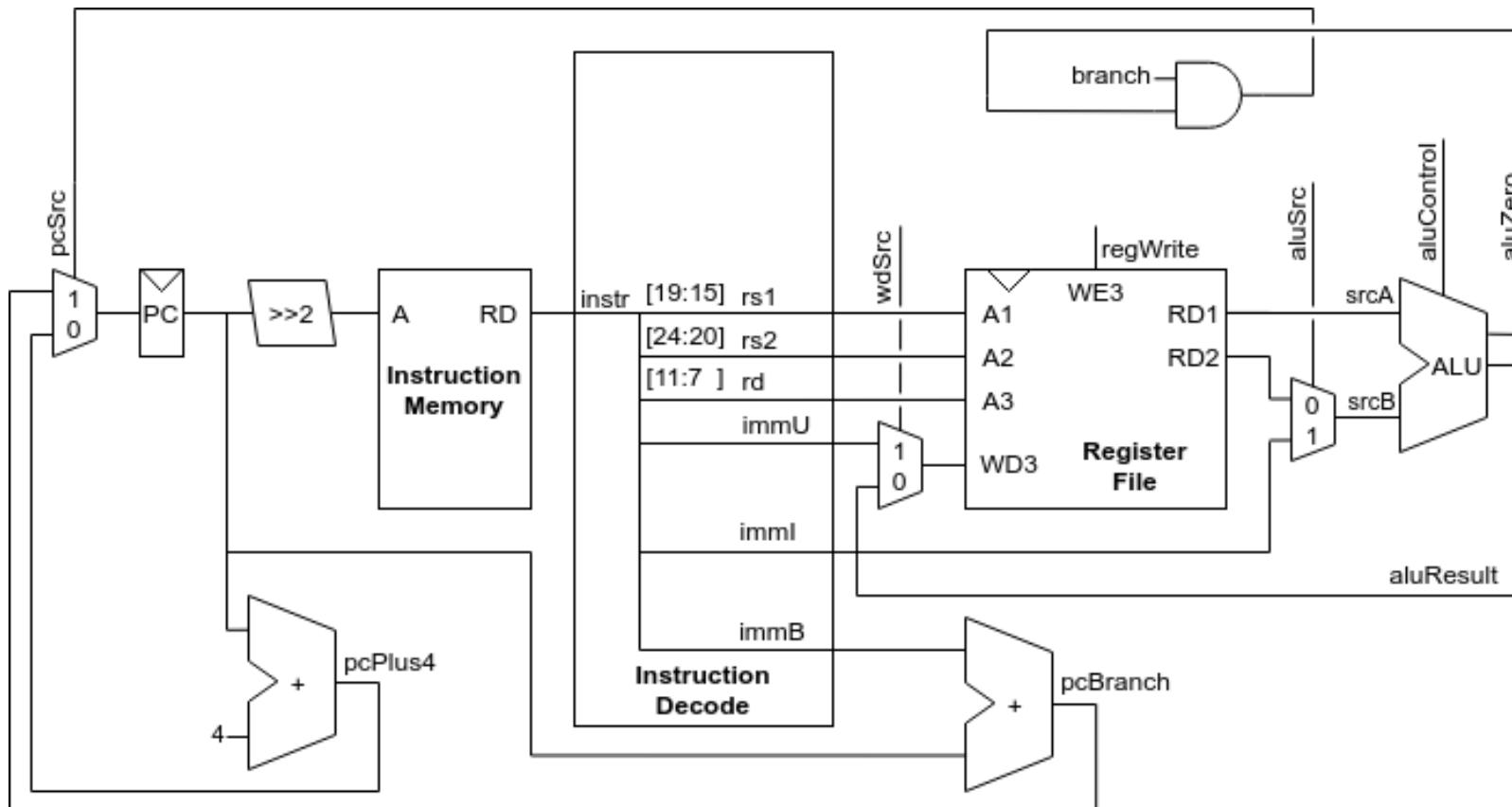
31	25	24	20	19	15	14	12	11	7	6	0
imm	rs2		rs1		funct3		imm		opcode		

B-type

31	30	20	19	12	11	10	5	4	1	0
--- inst[31] ---										

B-immediate

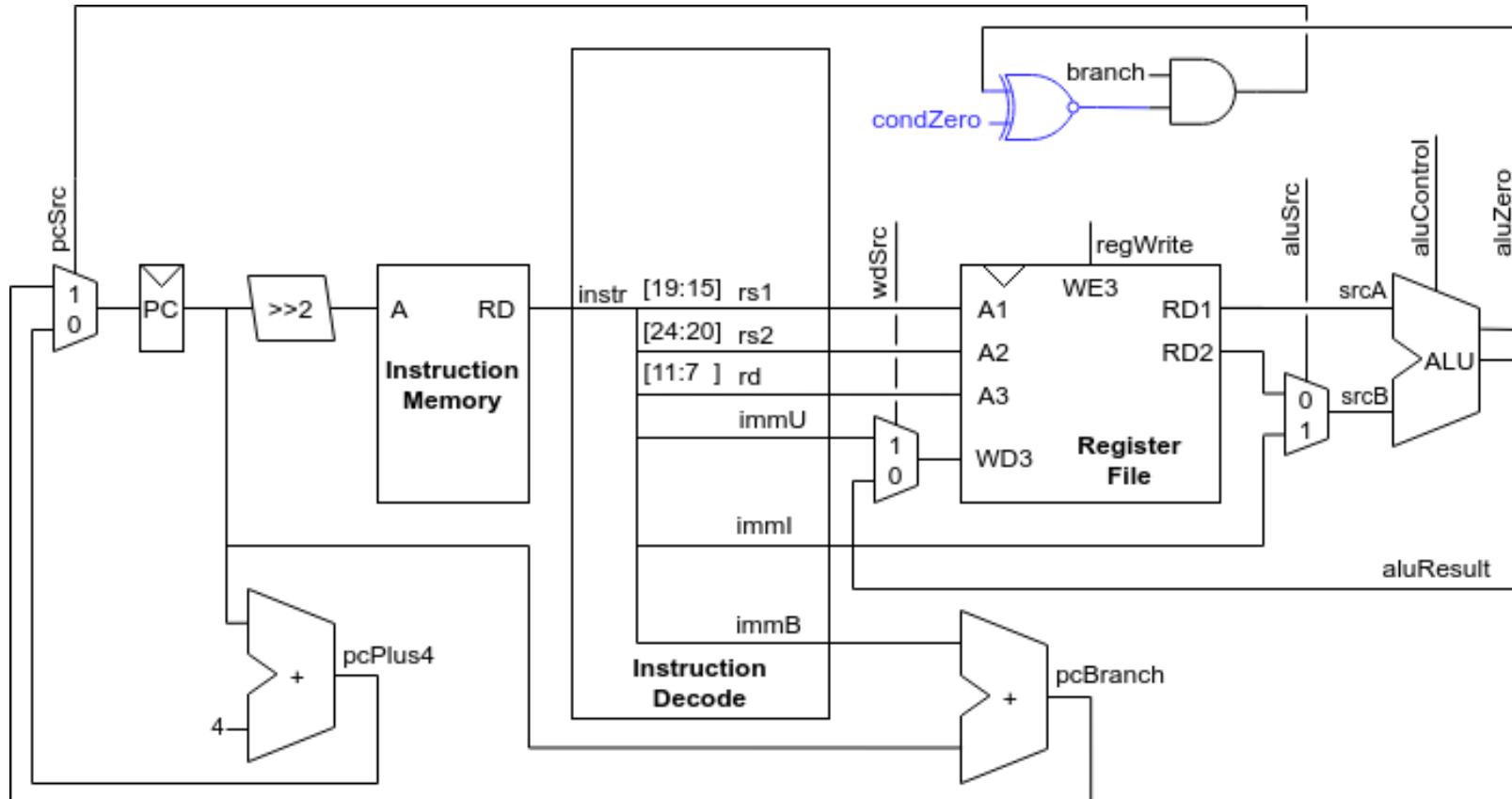
BNE: BEQ наоборот



BNE: B-type, branch on non equal: if ($rs1 \neq rs2$) $PC = PC + immB$

31	25	24	20	19	15	14	12	11	7	6	0		
imm	rs2		rs1		funct3		imm		opcode			B-type	
31	30		20	19		12	11	10	5	4	1	0	
--- inst[31] ---									inst[7]	inst[30:25]	inst[11:8]	0	B-immediate

BNE: новая цепь управления



BNE: B-type, branch on non equal: if ($rs1 \neq rs2$) $PC = PC + immB$

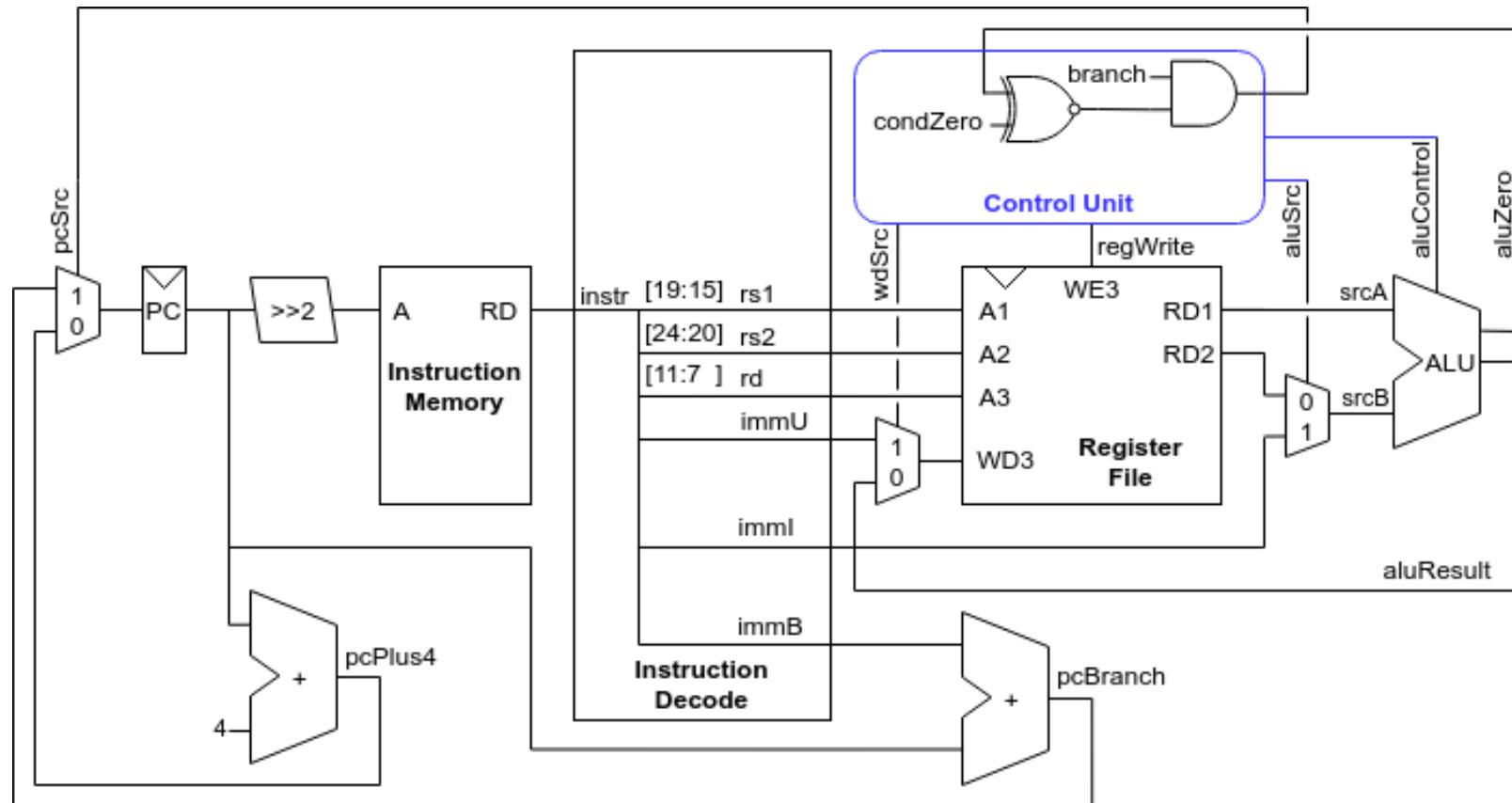
31	25	24	20	19	15	14	12	11	7	6	0
imm	rs2	rs1	funct3	imm	opcode						

B-type

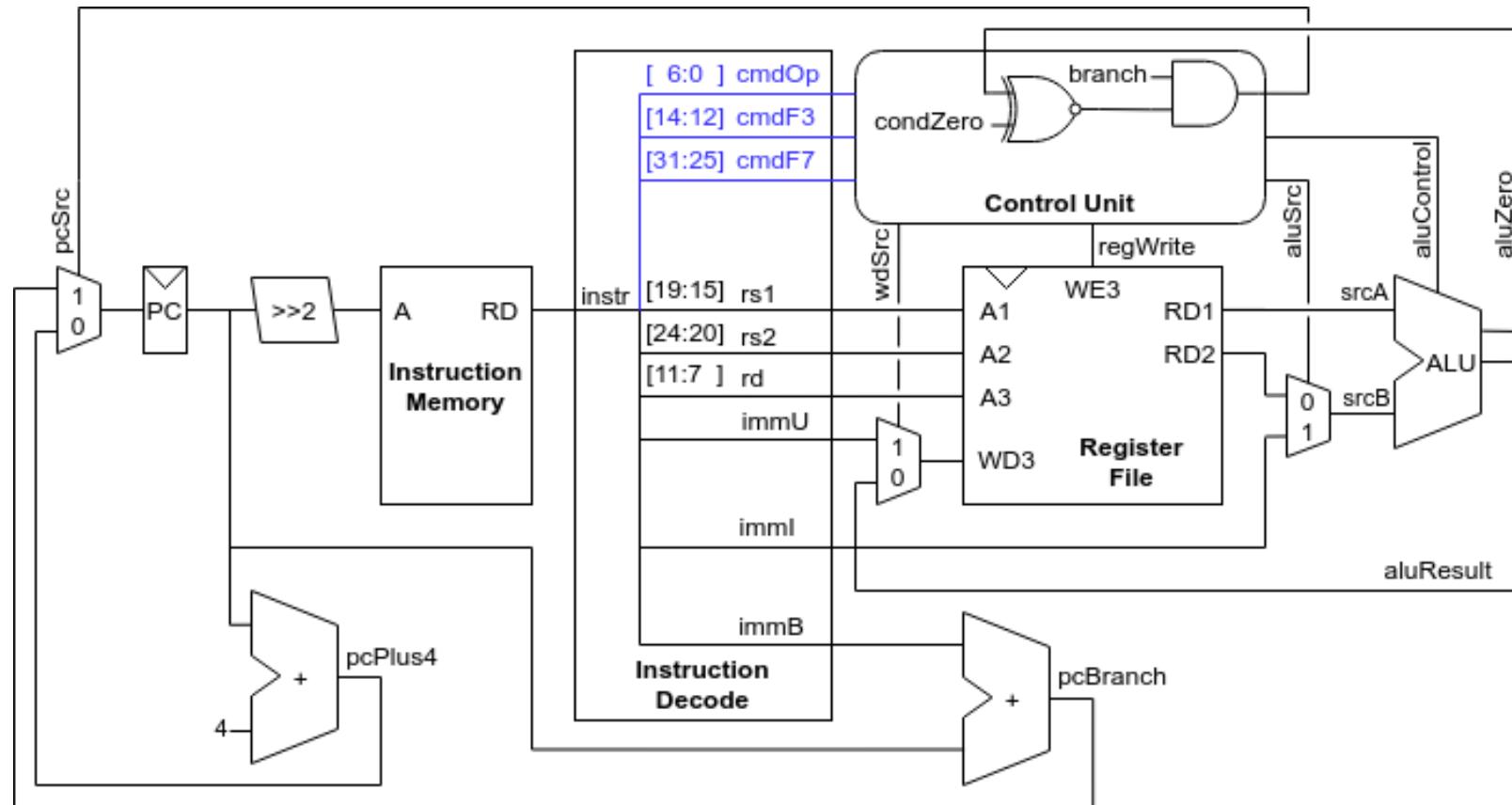
31	30	20	19	12	11	10	5	4	1	0	
--- inst[31] ---						inst[7]	inst[30:25]			inst[11:8]	0

B-immediate

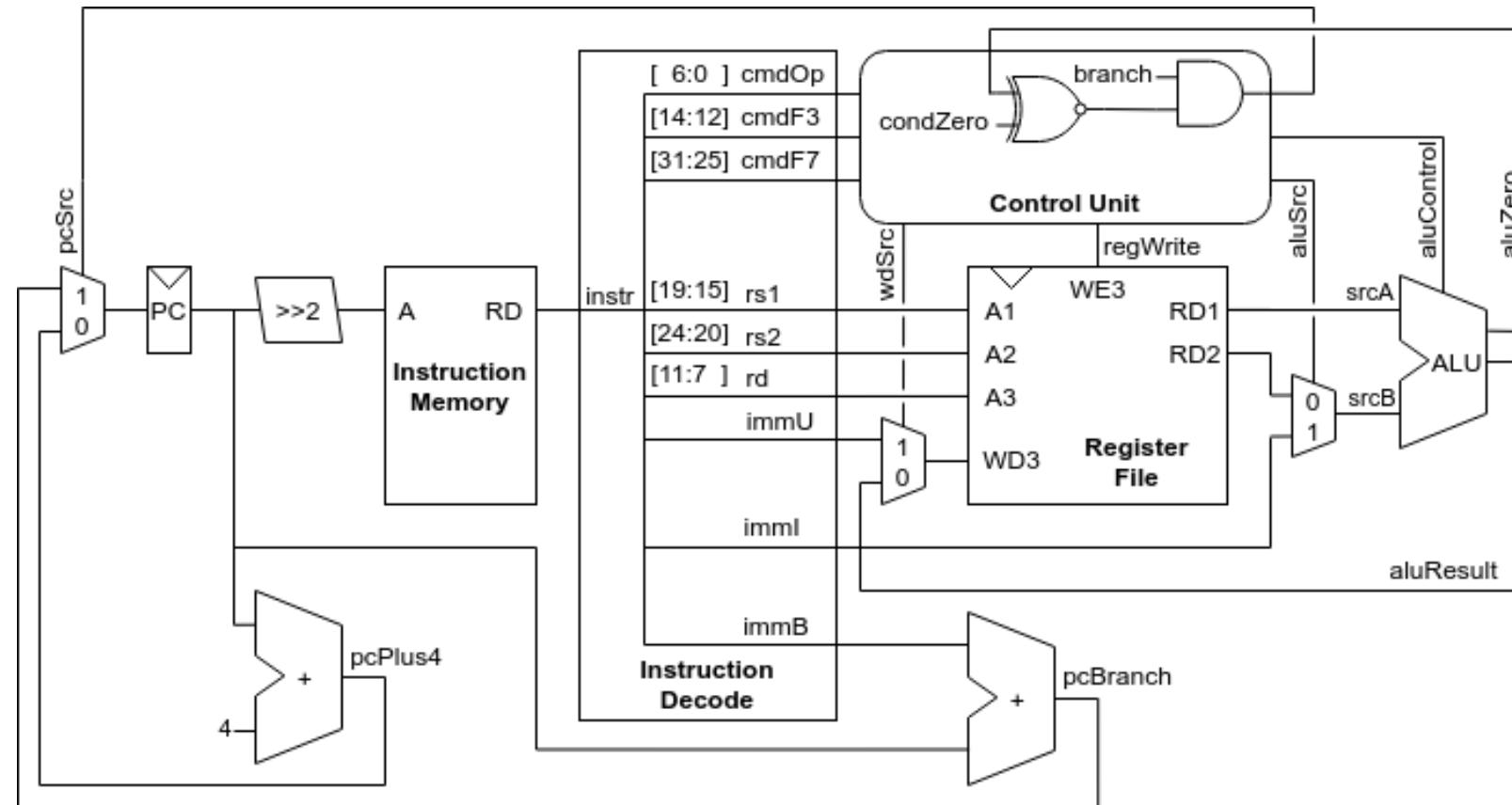
Устройство управления



Декодирование типа инструкции



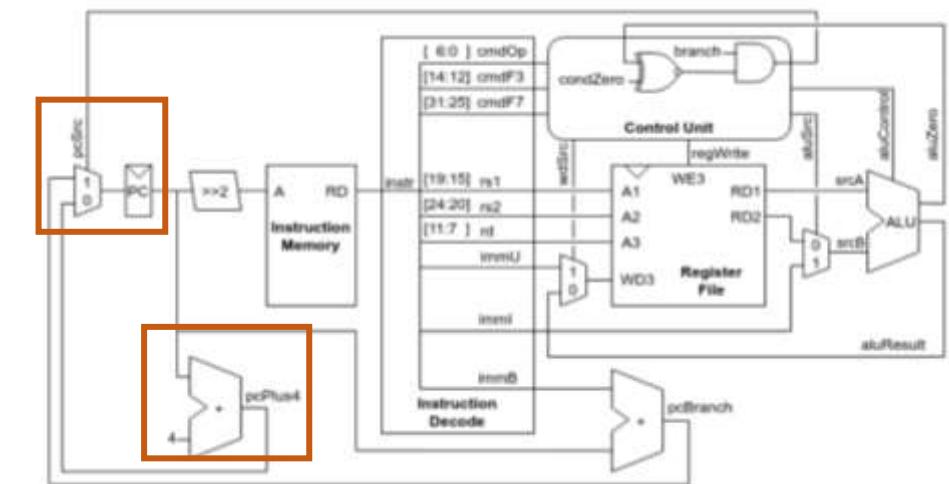
Итоговая схема процессора



Реализация: PC, сумматоры и мультиплексор адреса

```
// sm_register.v
module sm_register
(
    input                  clk,
    input                  rst,
    input      [ 31 : 0 ] d,
    output reg [ 31 : 0 ] q
);
    always @ (posedge clk or negedge rst)
        if(~rst)
            q <= 32'b0;
        else
            q <= d;
endmodule
```

```
// sr_cpu.v
wire [31:0] pc;
wire [31:0] pcBranch = pc + immB;
wire [31:0] pcPlus4  = pc + 4;
wire [31:0] pcNext   = pcSrc ? pcBranch : pcPlus4;
sm_register r_pc(clk ,rst_n, pcNext, pc);
```

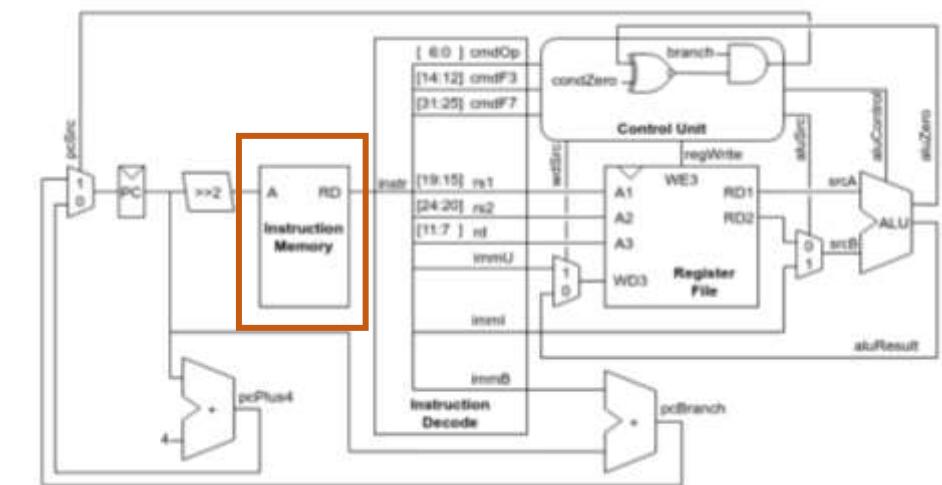


Реализация: память инструкций

```
// sm_rom.v
module sm_rom
#(
    parameter SIZE = 64
)
(
    input [31:0] a,
    output [31:0] rd
);
reg [31:0] rom [SIZE - 1:0];
assign rd = rom [a];

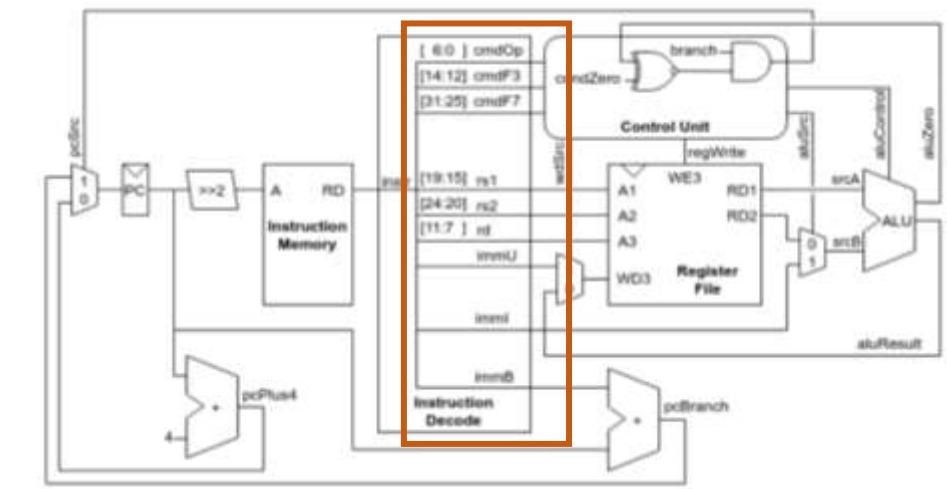
initial begin
    $readmemh ("program.hex", rom);
end
endmodule

// sm_top.v
sm_rom reset_rom(imAddr, imData);
```



Реализация: декодер инструкций (начало)

```
// sr_cpu.v
module sr_decode
(
    input      [31:0] instr,
    output     [ 6:0] cmdOp,
    output     [ 4:0] rd,
    output     [ 2:0] cmdF3,
    output     [ 4:0] rs1,
    output     [ 4:0] rs2,
    output     [ 6:0] cmdF7,
    output reg [31:0] immI,
    output reg [31:0] immB,
    output reg [31:0] immU
);
    assign cmdOp = instr[ 6: 0];
    assign rd   = instr[11: 7];
    assign cmdF3 = instr[14:12];
    assign rs1  = instr[19:15];
    assign rs2  = instr[24:20];
    assign cmdF7 = instr[31:25];
```

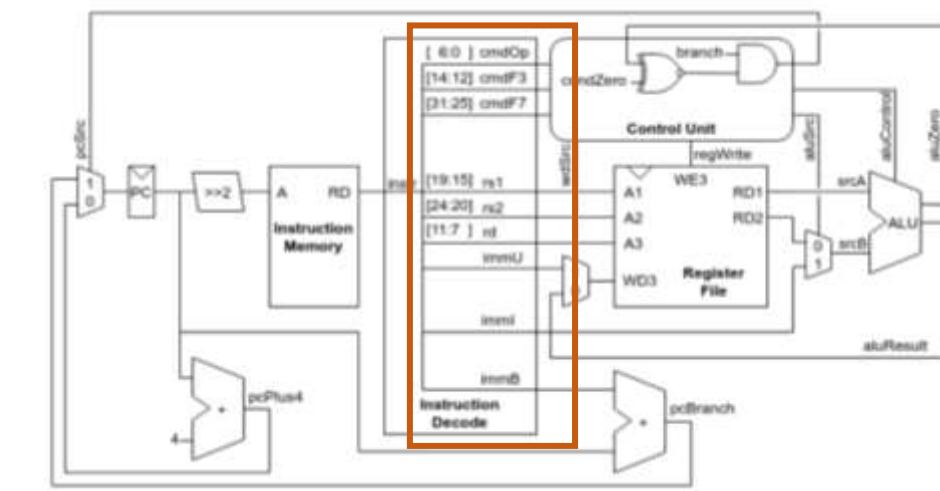


Реализация: декодер инструкций (продолжение)

```

// I-immediate
always @ (*) begin
    immI[10: 0] = instr[30:20];
    immI[31:11] = { 21 {instr[31]} };
end
// B-immediate
always @ (*) begin
    immB[ 0] = 1'b0;
    immB[ 4: 1] = instr[11:8];
    immB[10: 5] = instr[30:25];
    immB[ 11] = instr[7];
    immB[31:12] = { 20 {instr[31]} };
end
// U-immediate
always @ (*) begin
    immU[11: 0] = 12'b0;
    immU[31:12] = instr[31:12];
end
endmodule

```

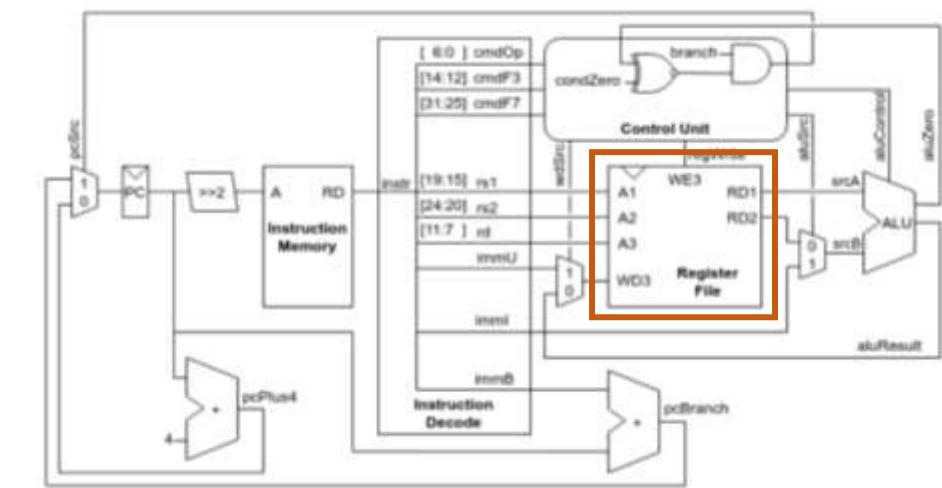


Реализация: регистровый файл

```
// sr_cpu.v
module sm_register_file
(
    input      clk,
    input [ 4:0] a0,
    input [ 4:0] a1,
    input [ 4:0] a2,
    input [ 4:0] a3,
    output [31:0] rd0,
    output [31:0] rd1,
    output [31:0] rd2,
    input [31:0] wd3,
    input       we3
);
    reg [31:0] rf [31:0];

    assign rd0 = (a0 != 0) ? rf [a0] : 32'b0;
    assign rd1 = (a1 != 0) ? rf [a1] : 32'b0;
    assign rd2 = (a2 != 0) ? rf [a2] : 32'b0;

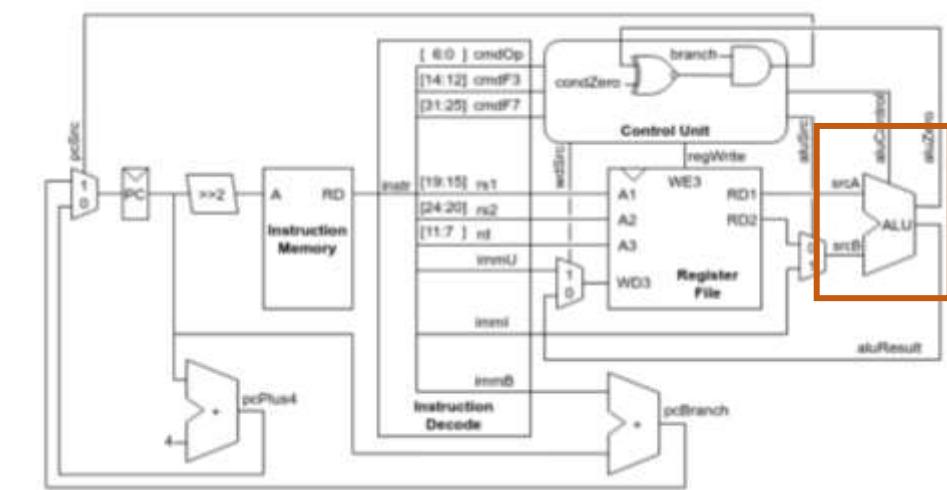
    always @ (posedge clk)
        if(we3) rf [a3] <= wd3;
endmodule
```



Реализация: операции ALU

```
// sr_cpu.vh
```

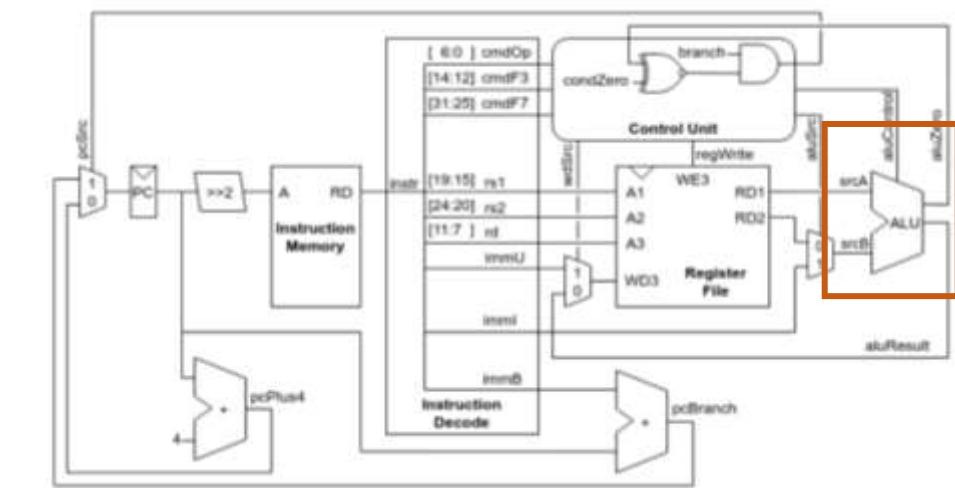
```
`define ALU_ADD      3'b000 // A + B
`define ALU_OR       3'b001 // A | B
`define ALU_SRL      3'b010 // A >> B
`define ALU_SLTU     3'b011 // A < B ? 1 : 0
`define ALU_SUB      3'b100 // A - B
```



Реализация: ALU

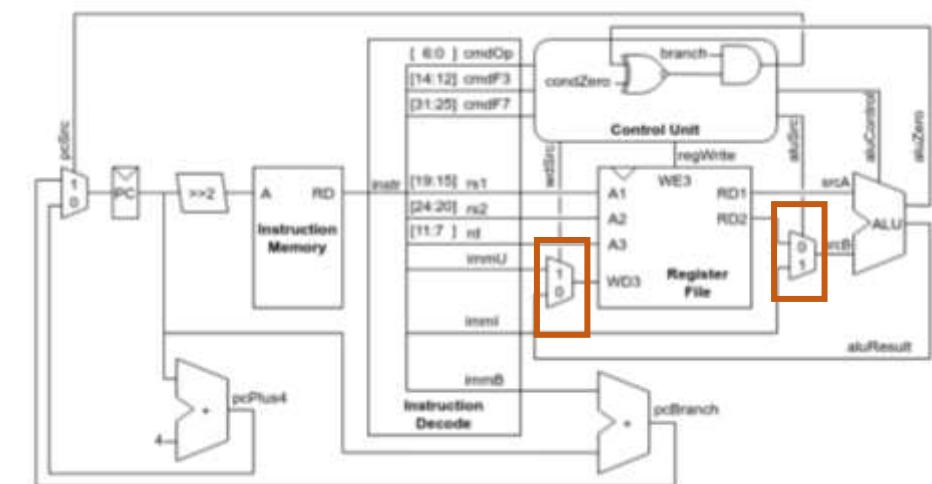
```
// sr_cpu.v
module sr_alu
(
    input [31:0] srcA,
    input [31:0] srcB,
    input [2:0] oper,
    output      zero,
    output reg [31:0] result
);
    always @ (*) begin
        case (oper)
            default : result = srcA + srcB;
            `ALU_ADD : result = srcA + srcB;
            `ALU_OR  : result = srcA | srcB;
            `ALU_SRL : result = srcA >> srcB [4:0];
            `ALU_SLTU: result = (srcA < srcB) ? 1 : 0;
            `ALU_SUB : result = srcA - srcB;
        endcase
    end

    assign zero = (result == 0);
endmodule
```



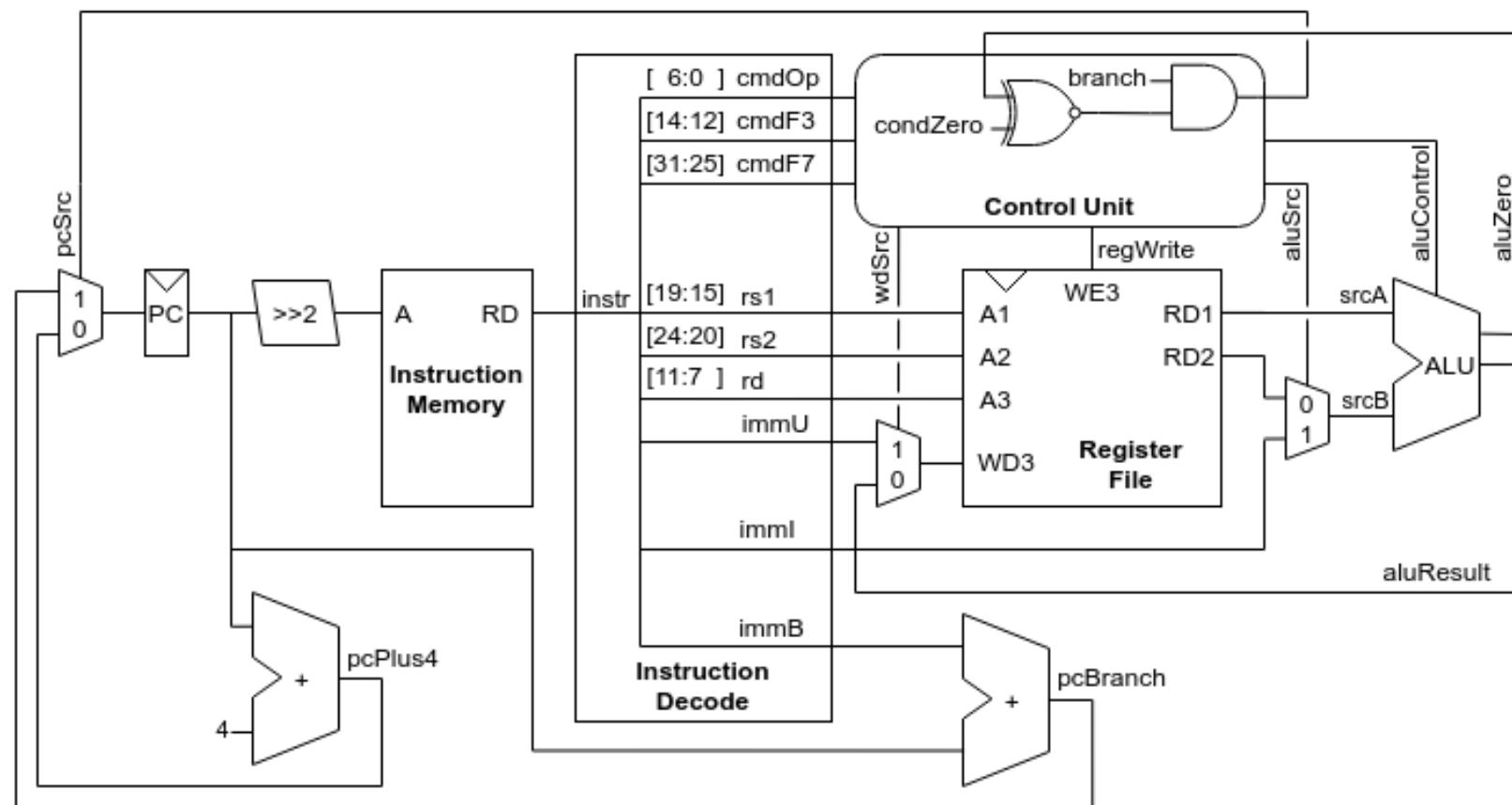
Реализация: мультиплексоры данных

```
// sr_cpu.v  
wire [31:0] srcB = aluSrc ? immI : rd2;  
  
// sr_cpu.v  
assign wd3 = wdSrc ? immU : aluResult;
```



Сигналы управления 1

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero



Код операции: спецификация

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct7					rs2		rs1		funct3		rd		opcode
					imm[11:0]				funct3		rd		opcode
					imm[11:5]		rs2		funct3		imm[4:0]		opcode
					imm[12 10:5]		rs2		funct3		imm[4:1 11]		opcode
							imm[31:12]				rd		opcode
							imm[20 10:1 11 19:12]				rd		opcode

R-type
I-type
S-type
B-type
U-type
J-type

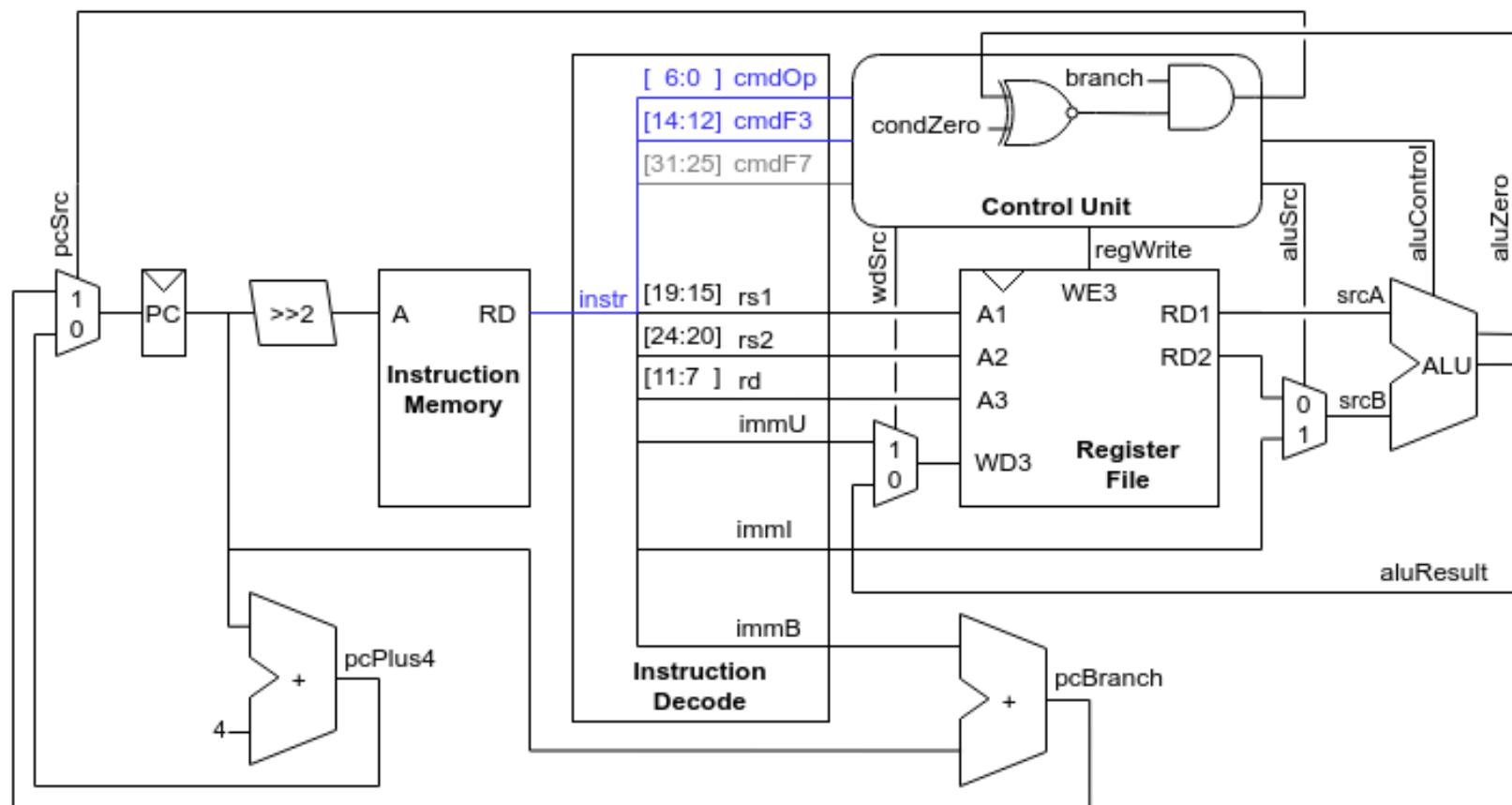
RV32I Base Instruction Set

		imm[31:12]		rd	0110111
		imm[31:12]		rd	0010111
		imm[20 10:1 11 19:12]		rd	1101111
		imm[11:0]	rs1	000	rd
	imm[12 10:5]	rs2	rs1	000	imm[4:1 11]
	imm[12 10:5]	rs2	rs1	001	imm[4:1 11]
	imm[12 10:5]	rs2	rs1	100	imm[4:1 11]
	imm[12 10:5]	rs2	rs1	101	imm[4:1 11]
	imm[12 10:5]	rs2	rs1	110	imm[4:1 11]
	imm[12 10:5]	rs2	rs1	111	imm[4:1 11]
		imm[11:0]	rs1	000	rd
		imm[11:0]	rs1	001	rd
		imm[11:0]	rs1	010	rd
		imm[11:0]	rs1	100	rd
		imm[11:0]	rs1	101	rd
	imm[11:5]	rs2	rs1	000	imm[4:0]
	imm[11:5]	rs2	rs1	001	imm[4:0]
	imm[11:5]	rs2	rs1	010	imm[4:0]
	imm[11:0]	rs1	000	rd	0010011

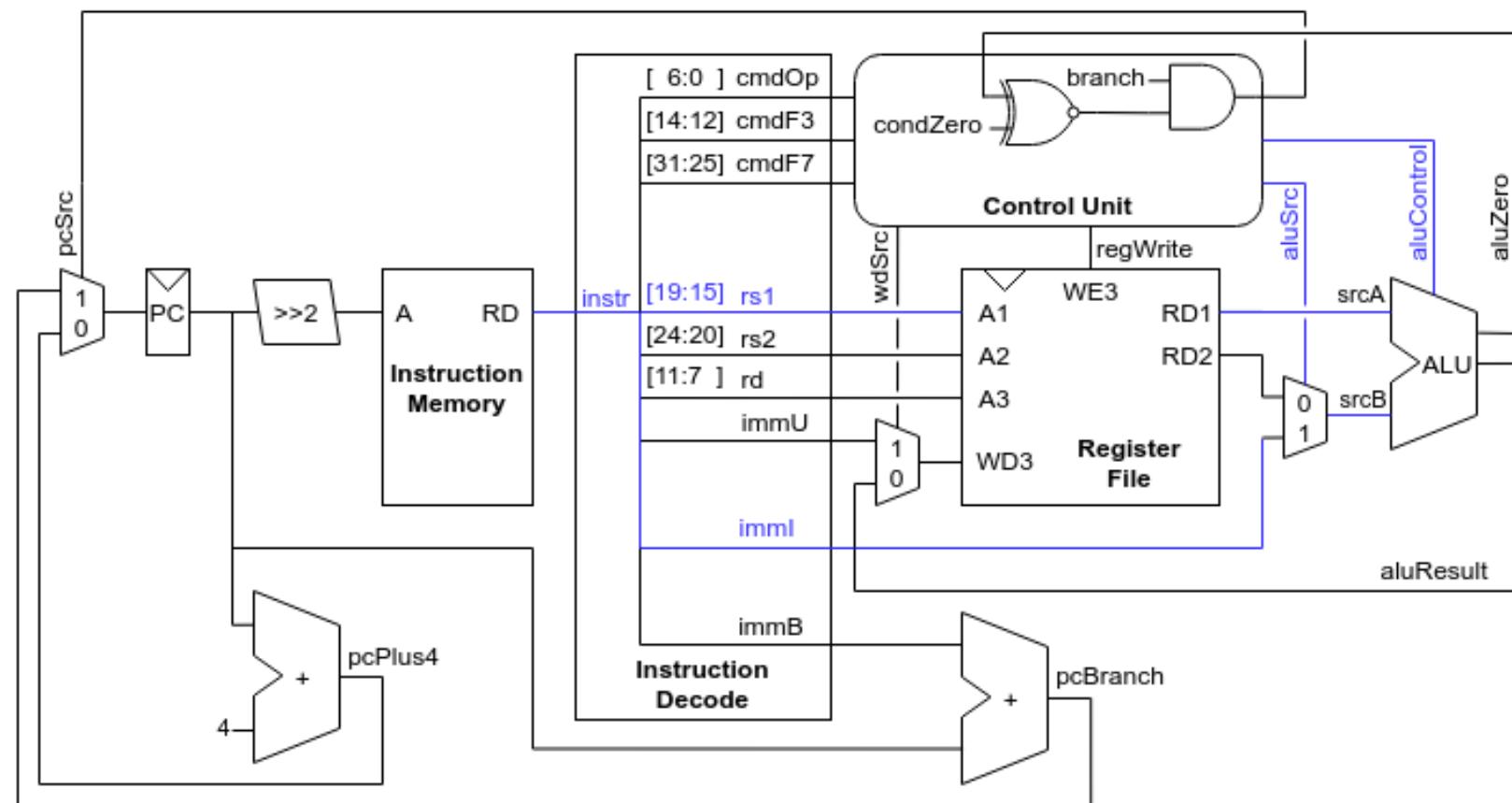
LUI
AUIPC
JAL
JALR
BEQ
BNE
BLT
BGE
BLTU
BGEU
LB
LH
LW
LBU
LHU
SB
SH
SW
ADDI

Сигналы управления 2

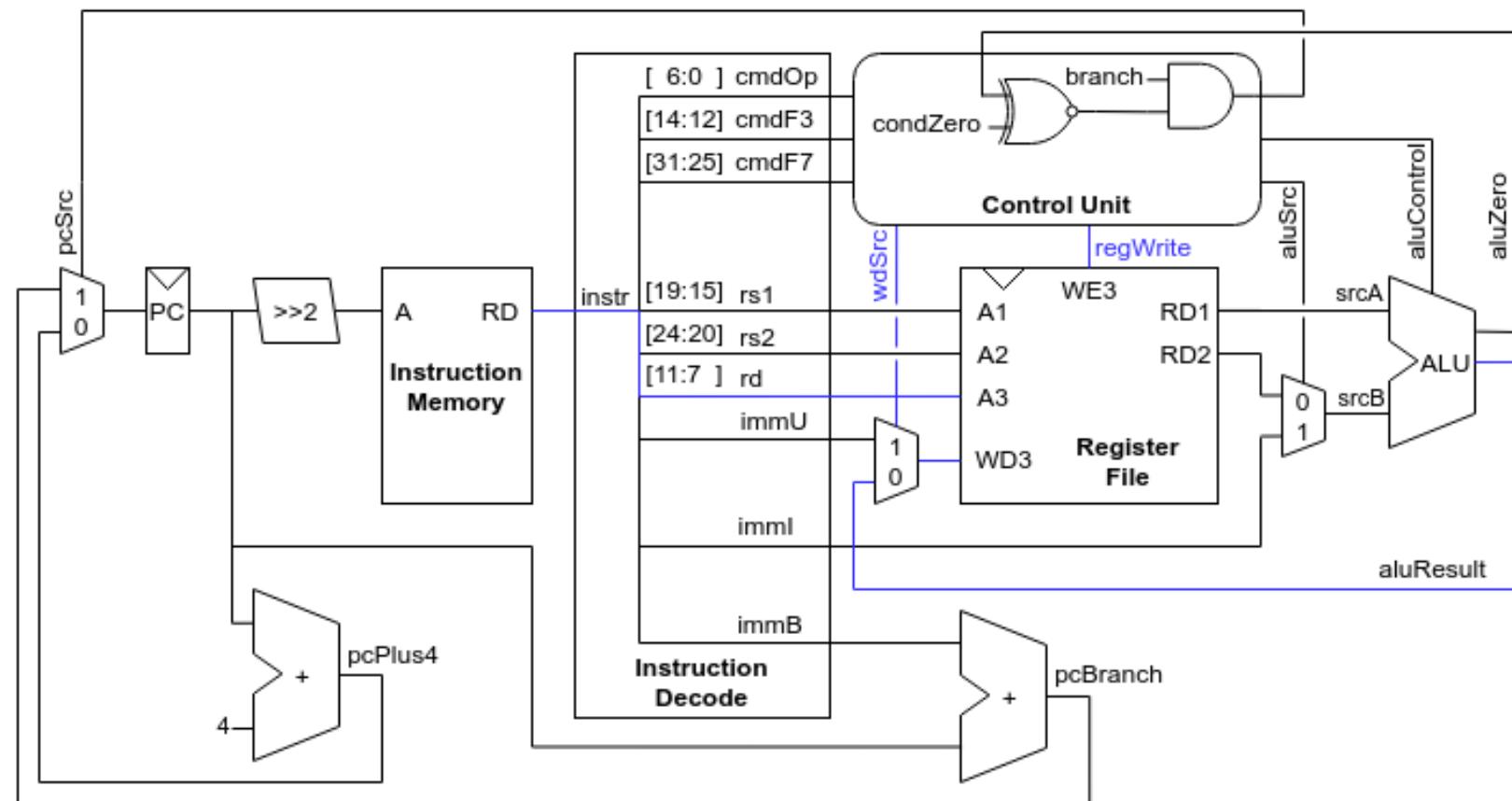
Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????							



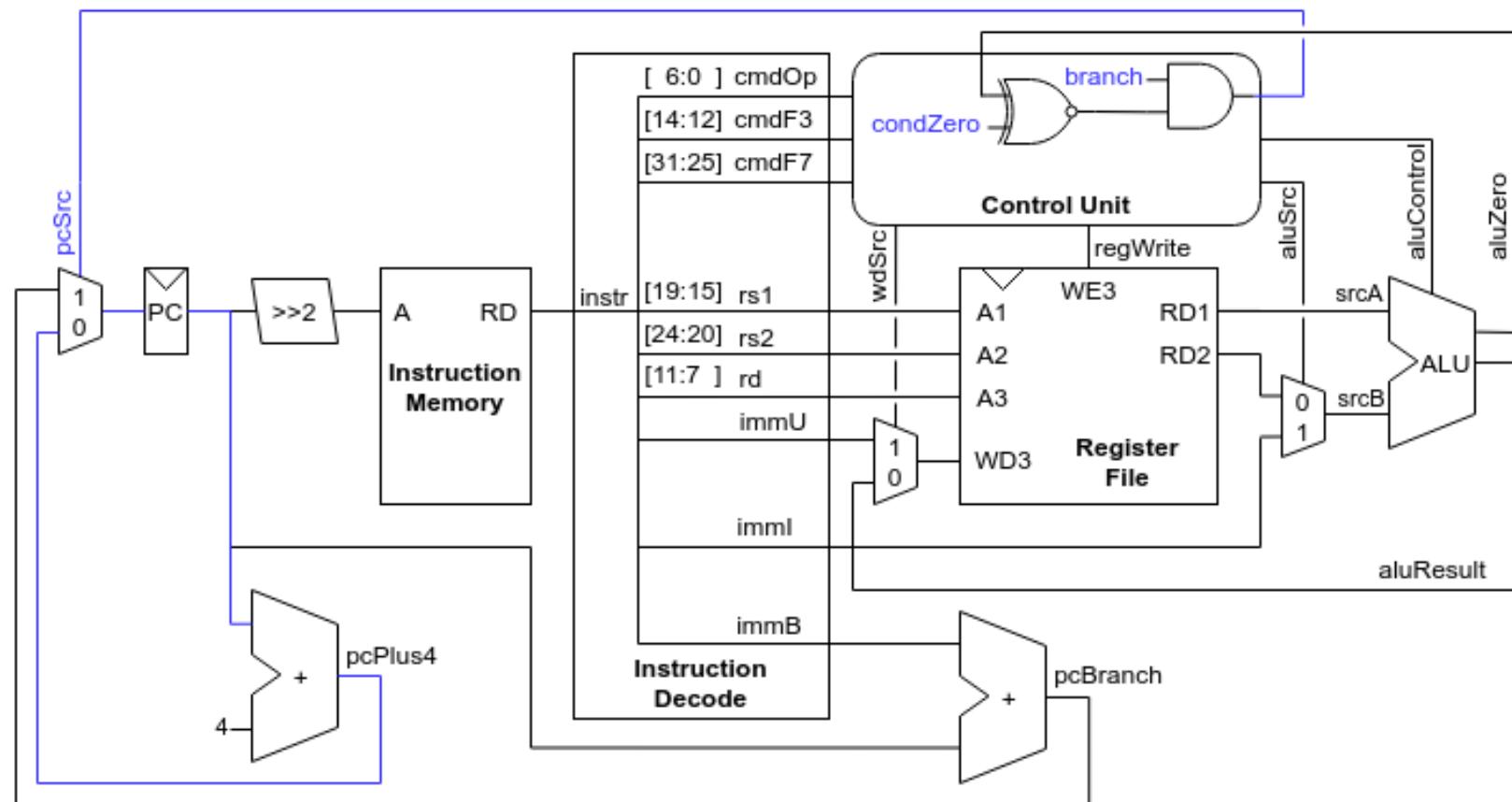
Сигналы управления 3



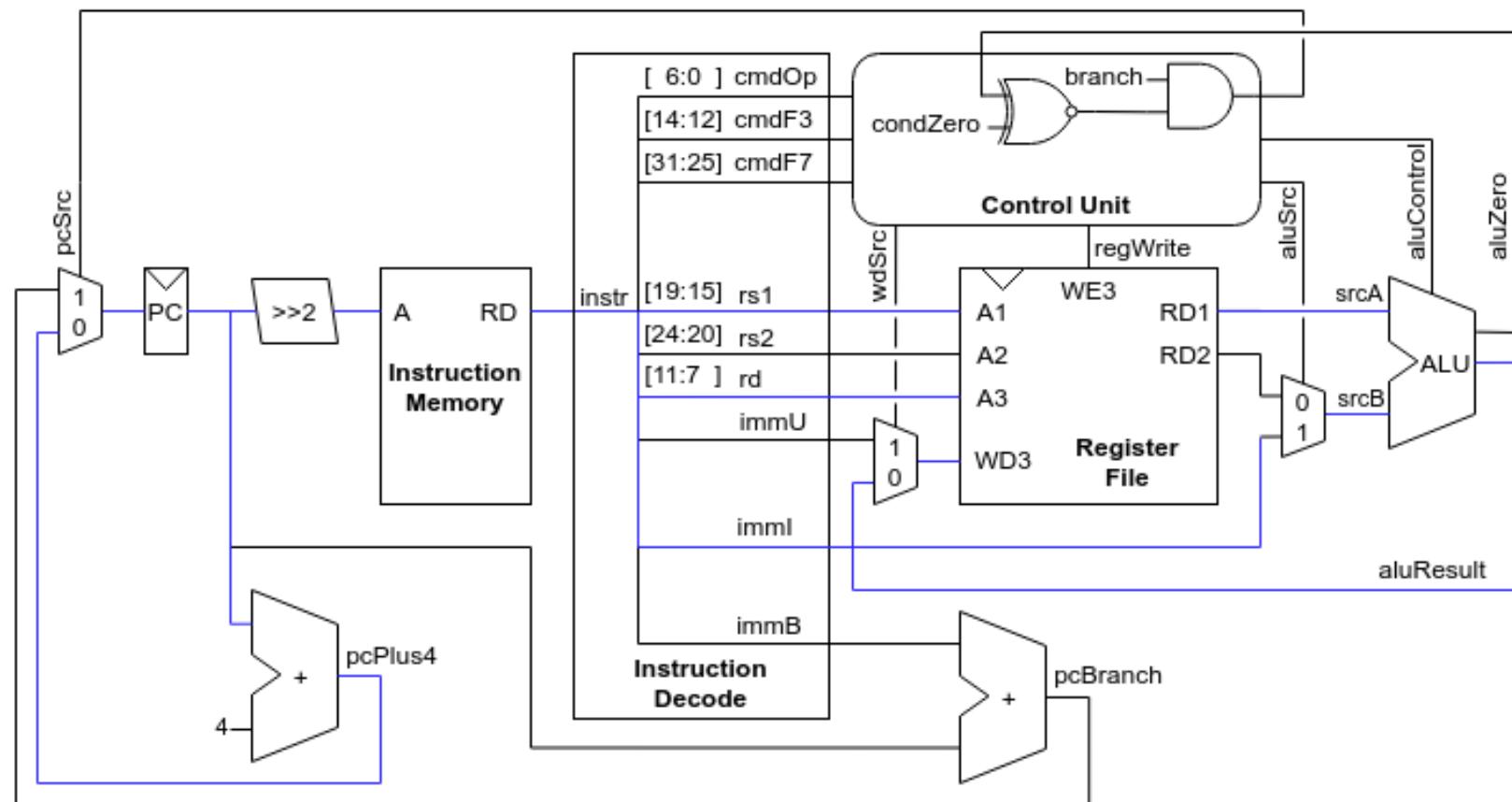
Сигналы управления 4



Сигналы управления 5

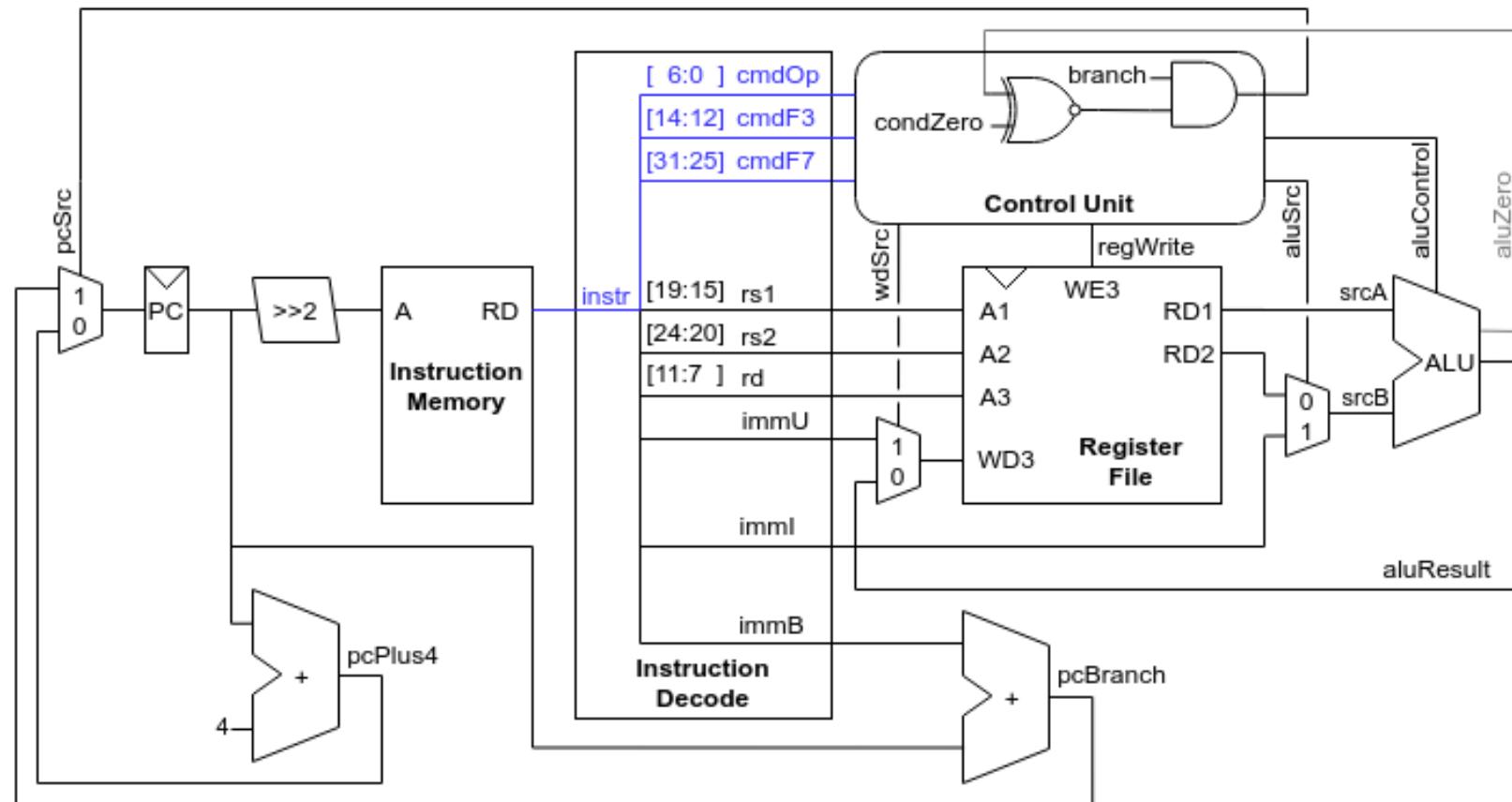


Сигналы управления 6



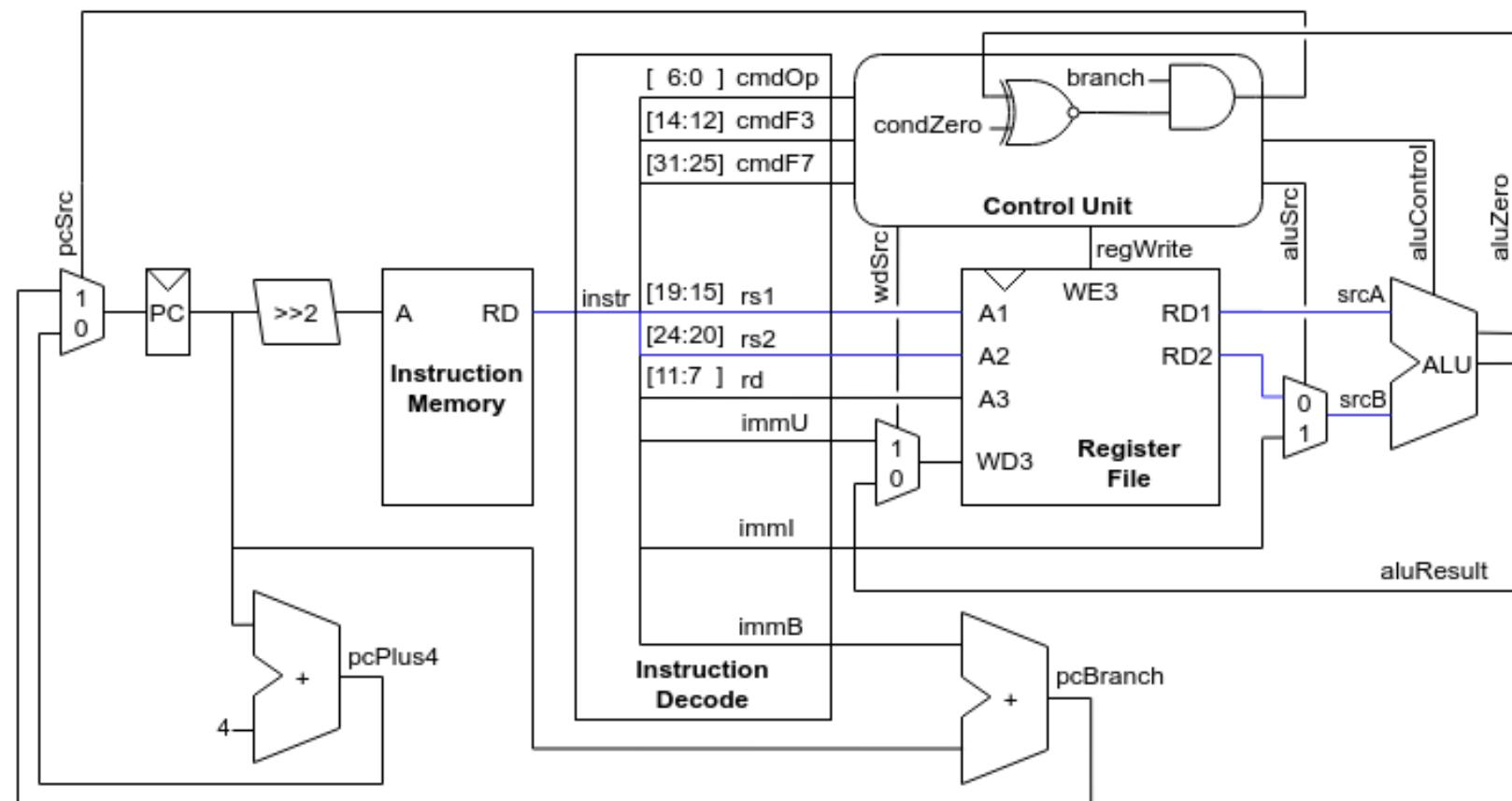
Сигналы управления 7

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000							



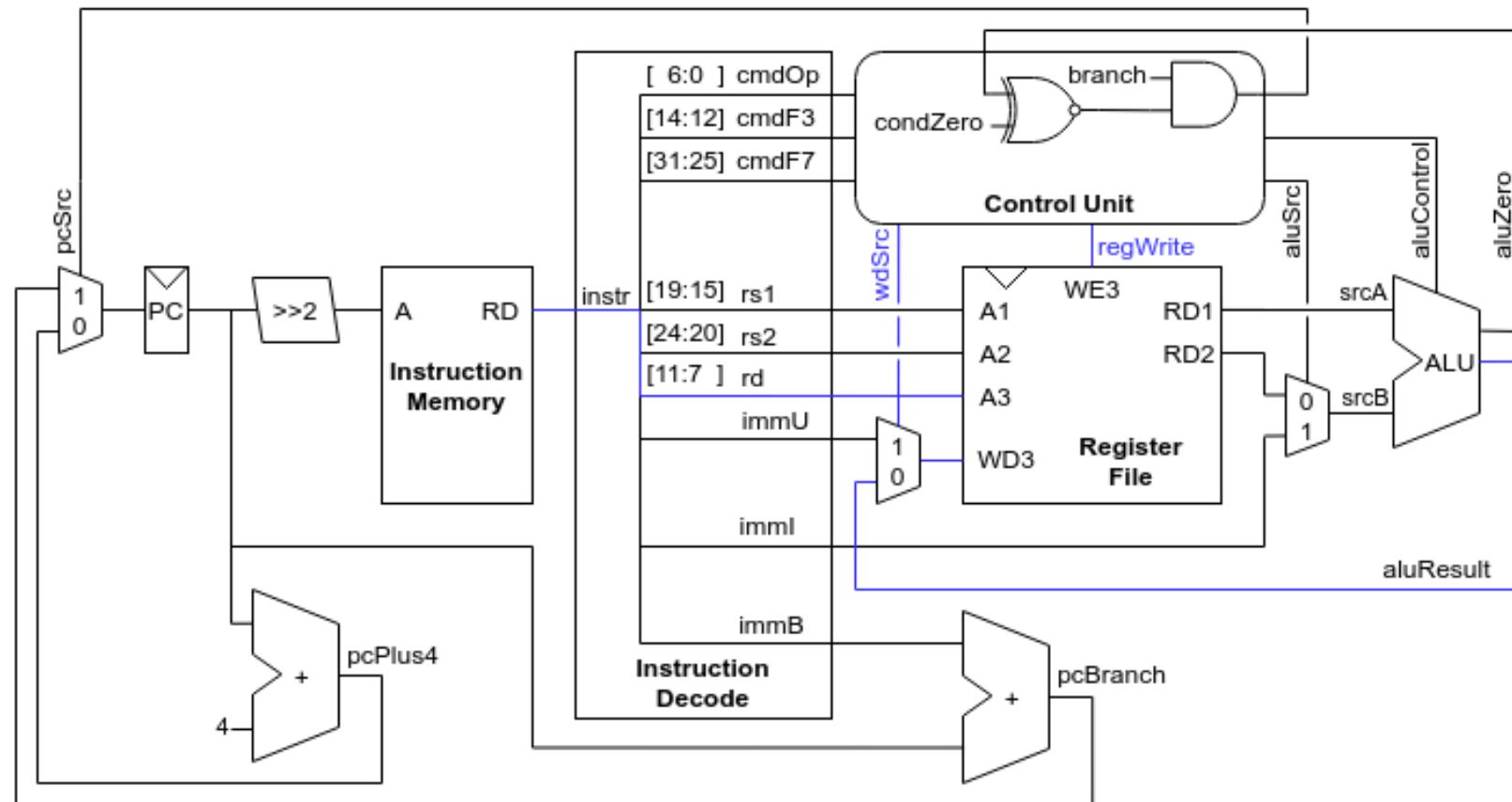
Сигналы управления 8

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000					



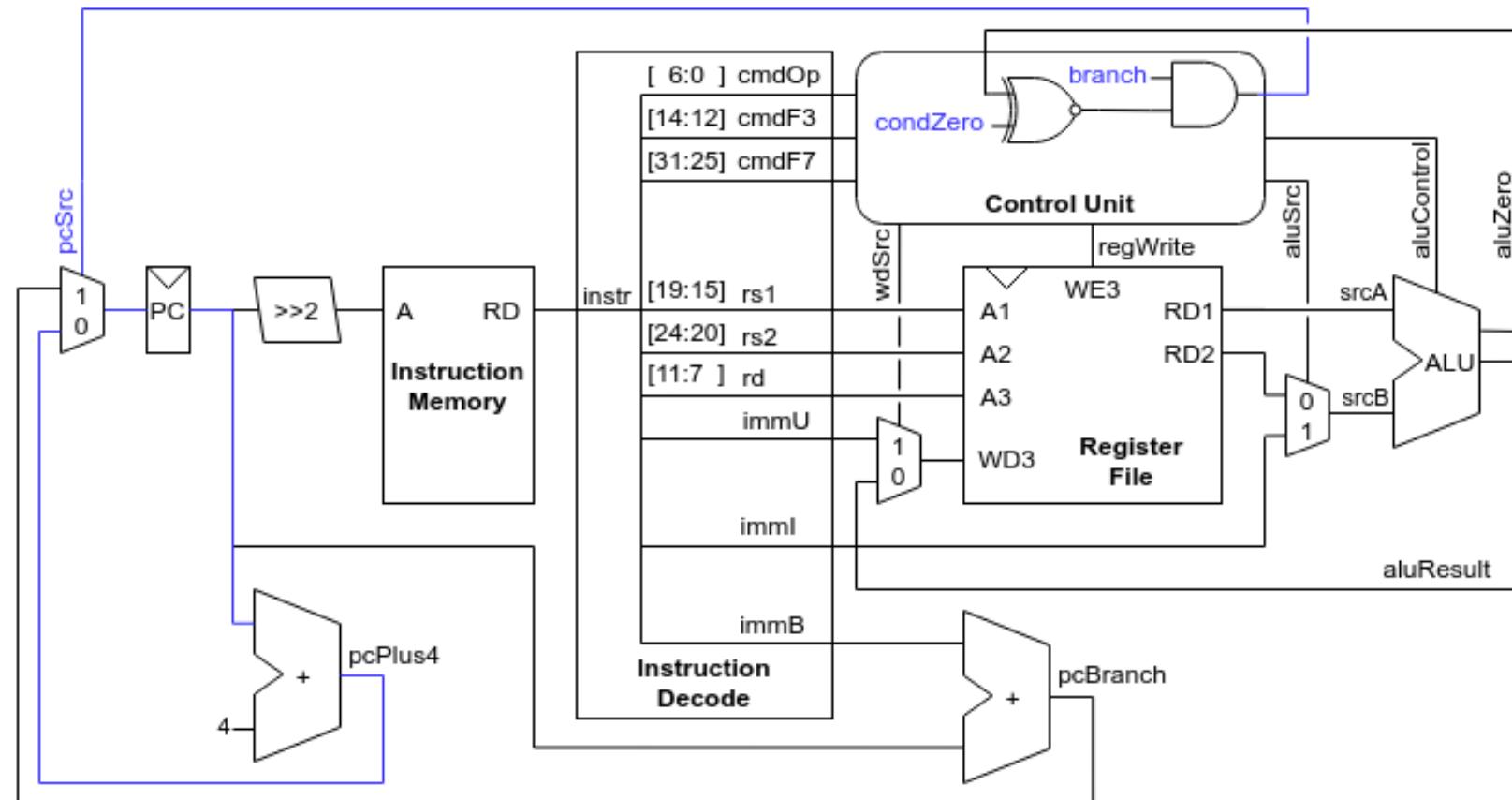
Сигналы управления 9

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1			



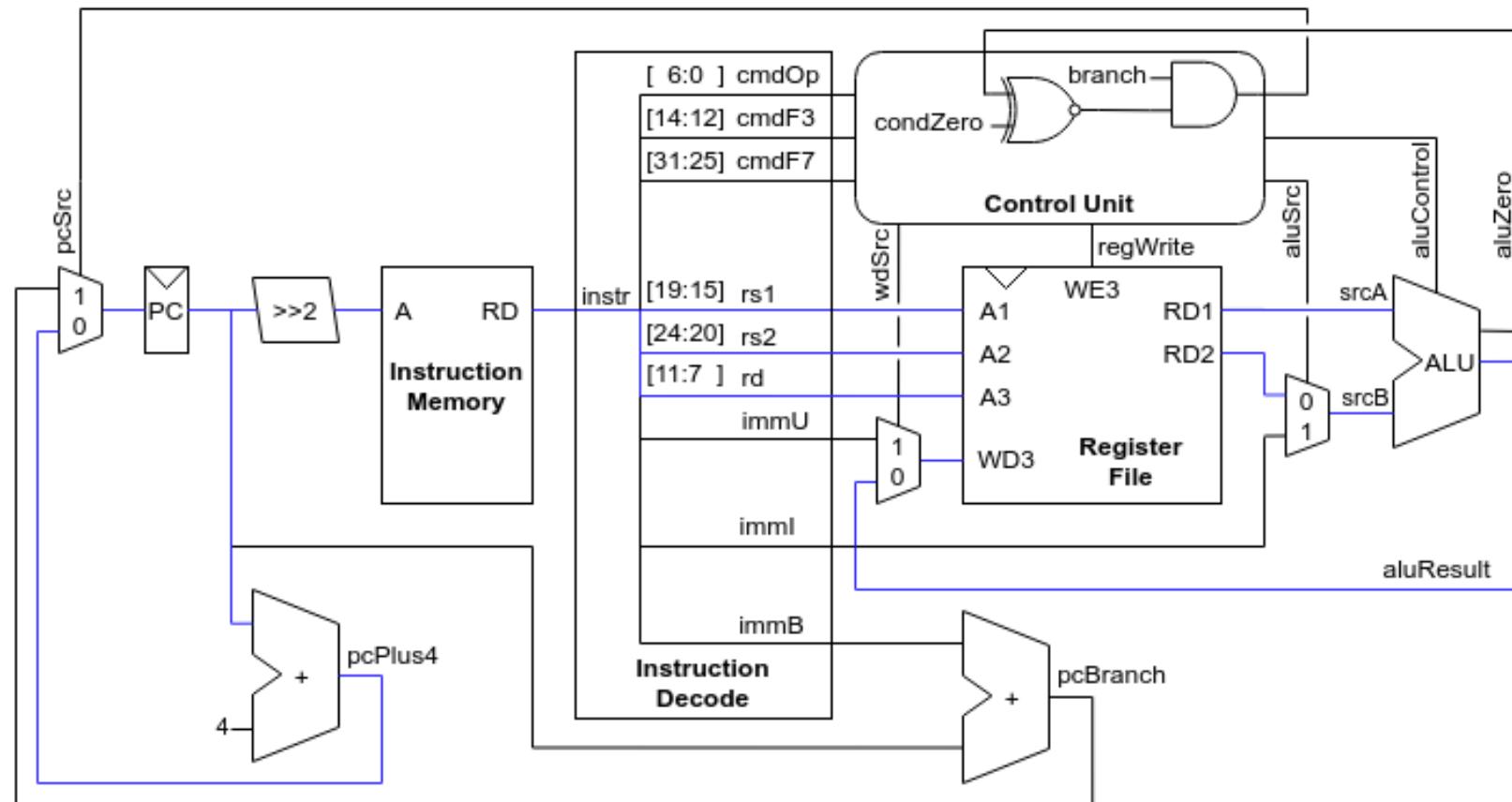
Сигналы управления 10

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0



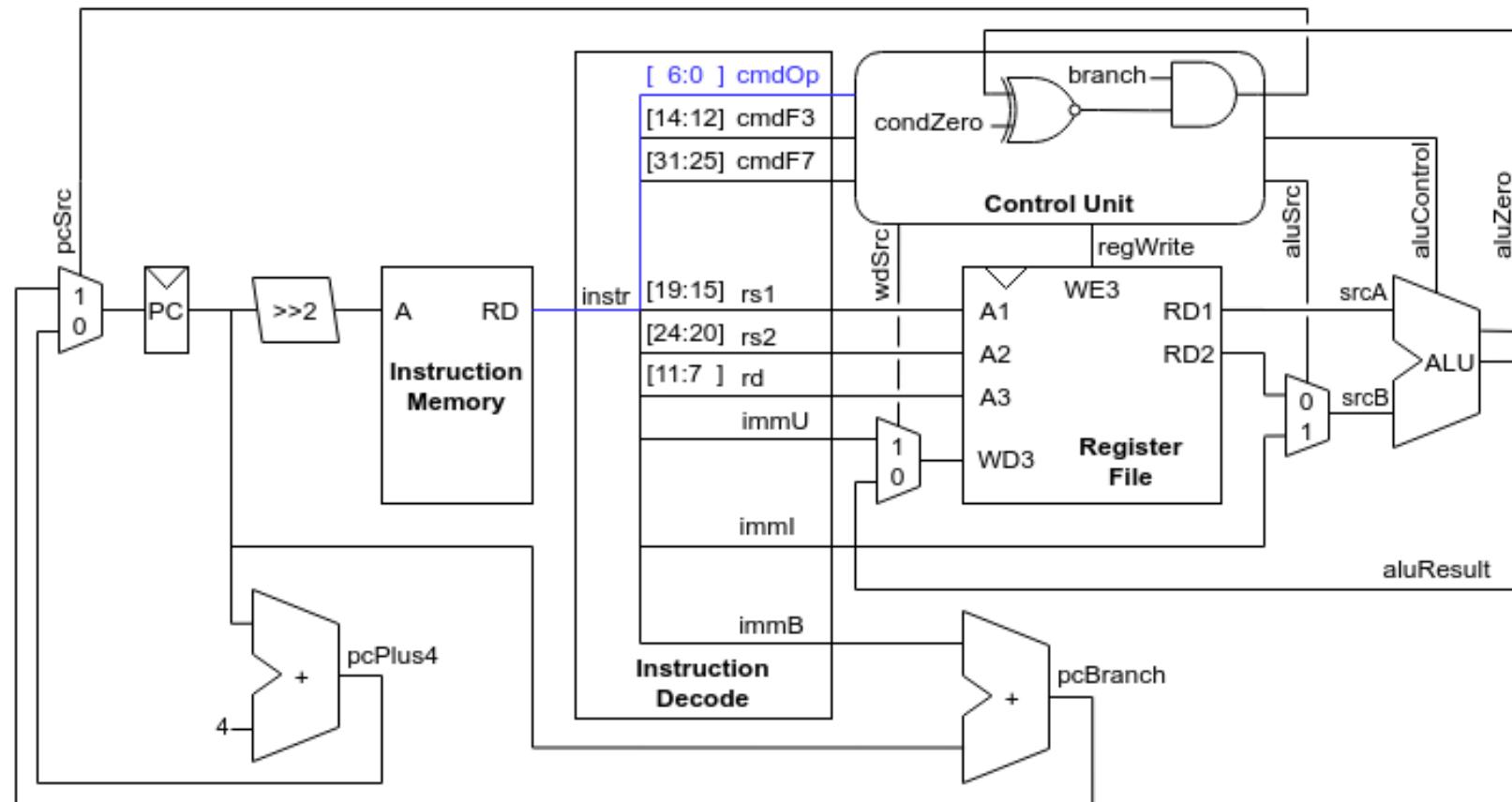
Сигналы управления 11

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0



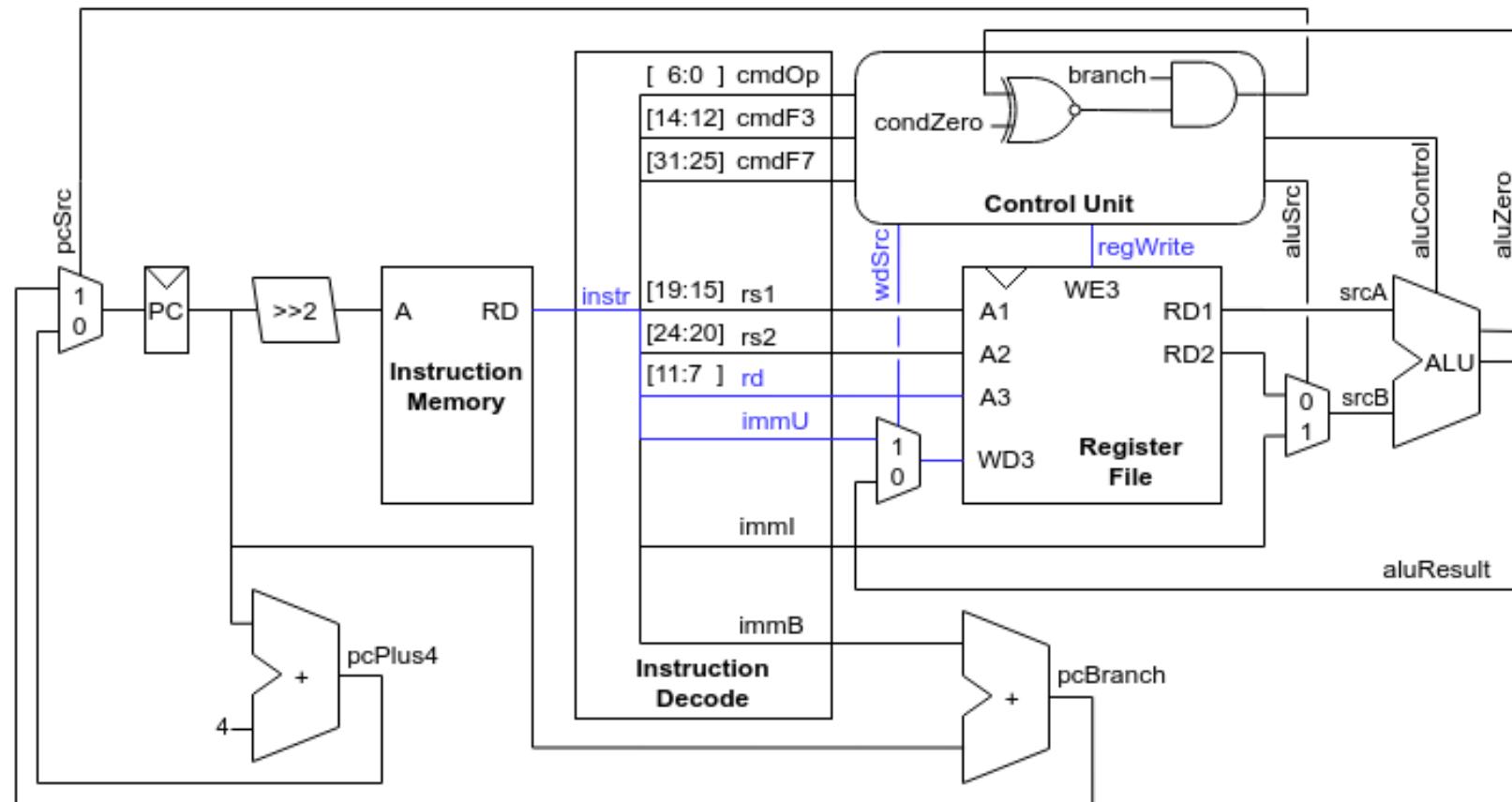
Сигналы управления 12

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????							



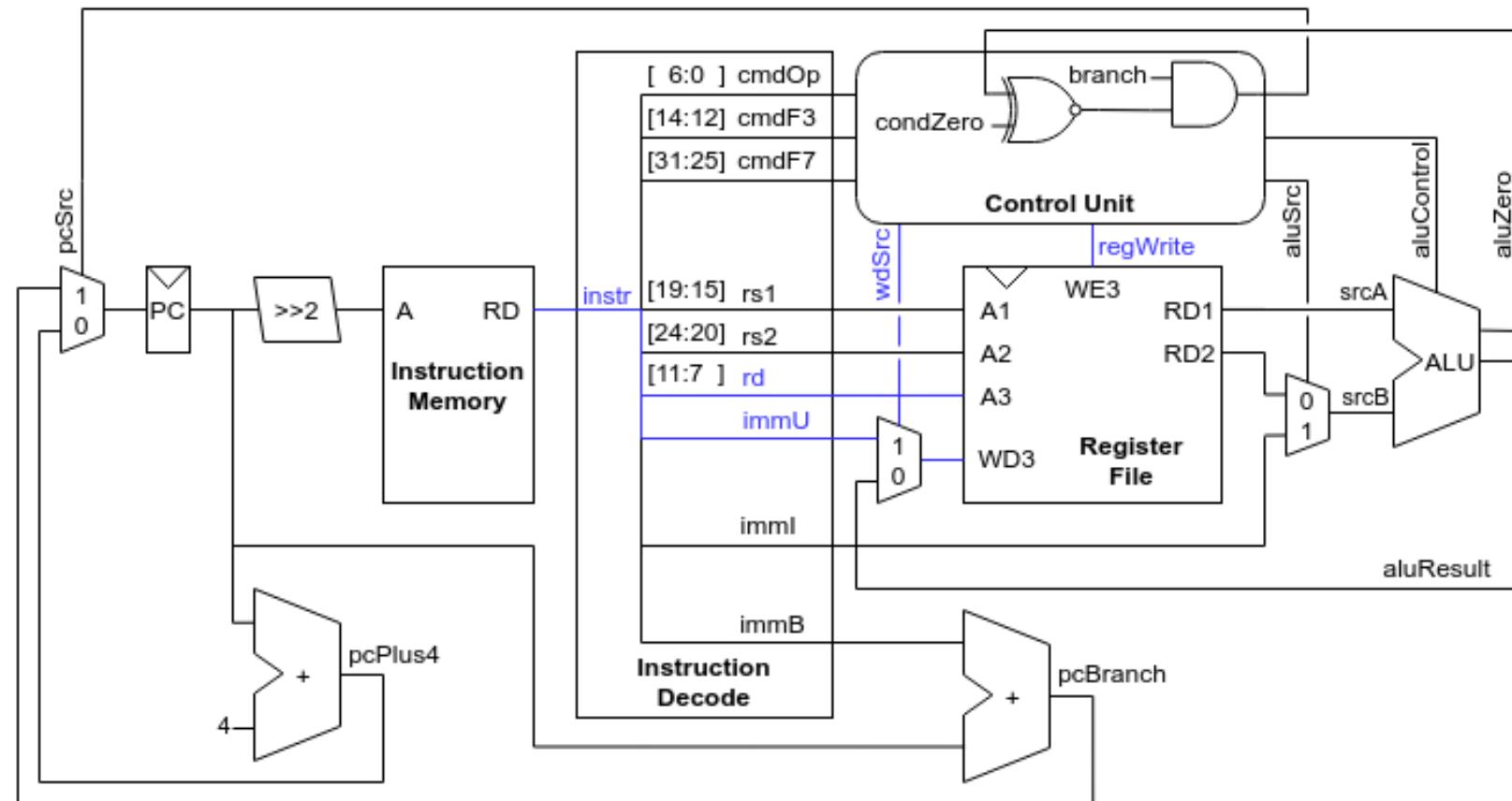
Сигналы управления 13

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????			1	1			



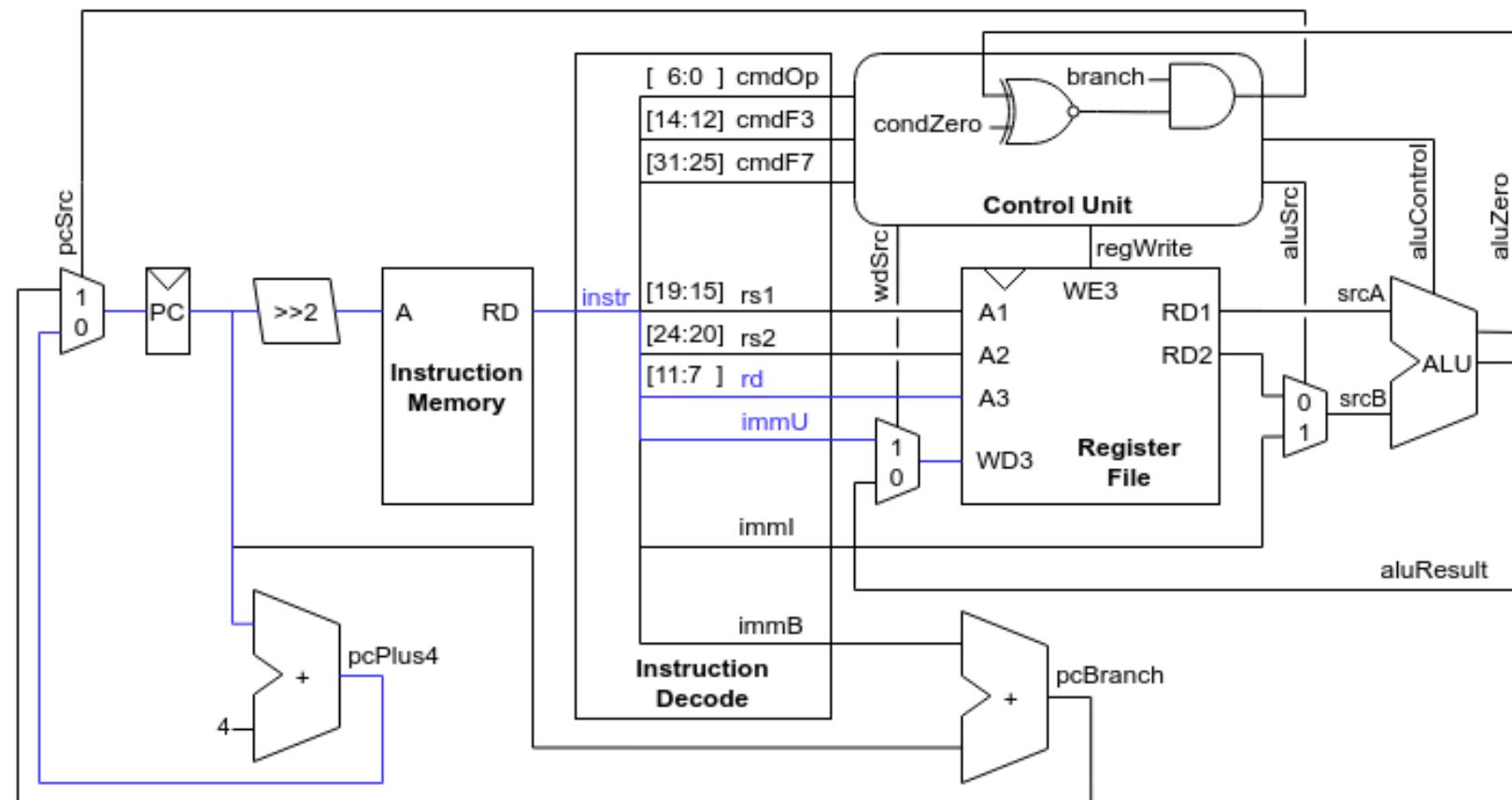
Сигналы управления 14

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0



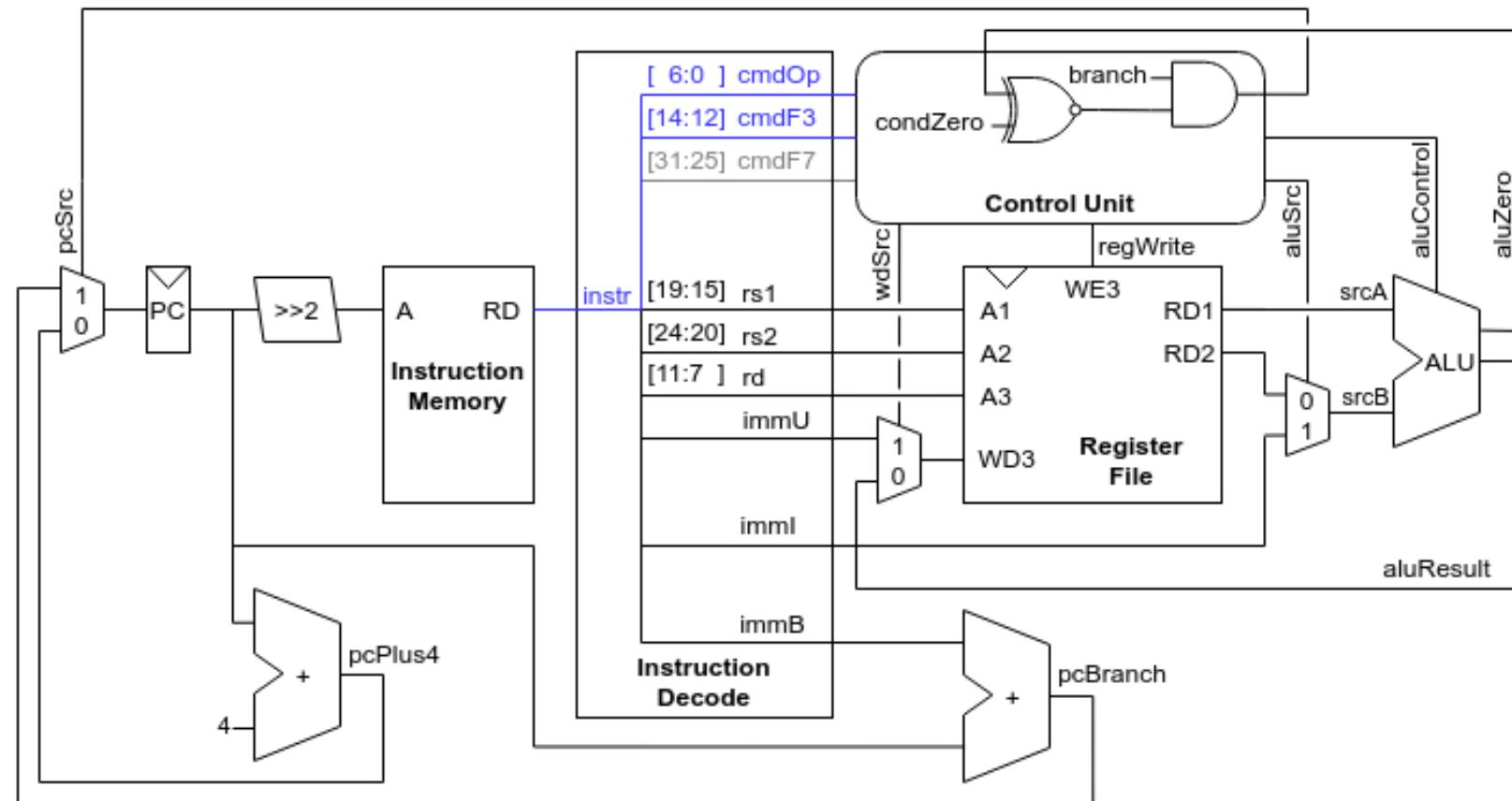
Сигналы управления 15

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0

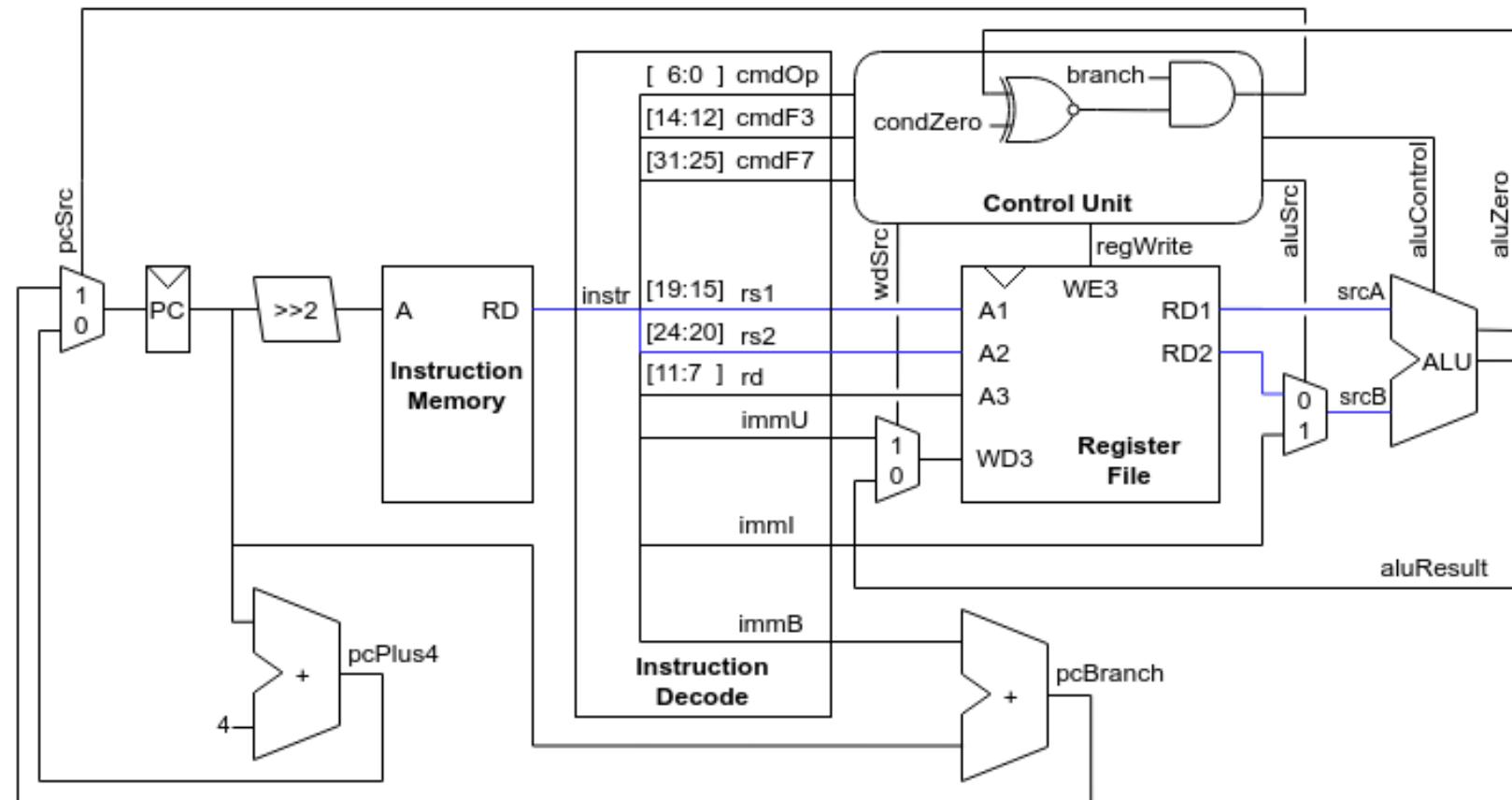


Сигналы управления 16

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????							

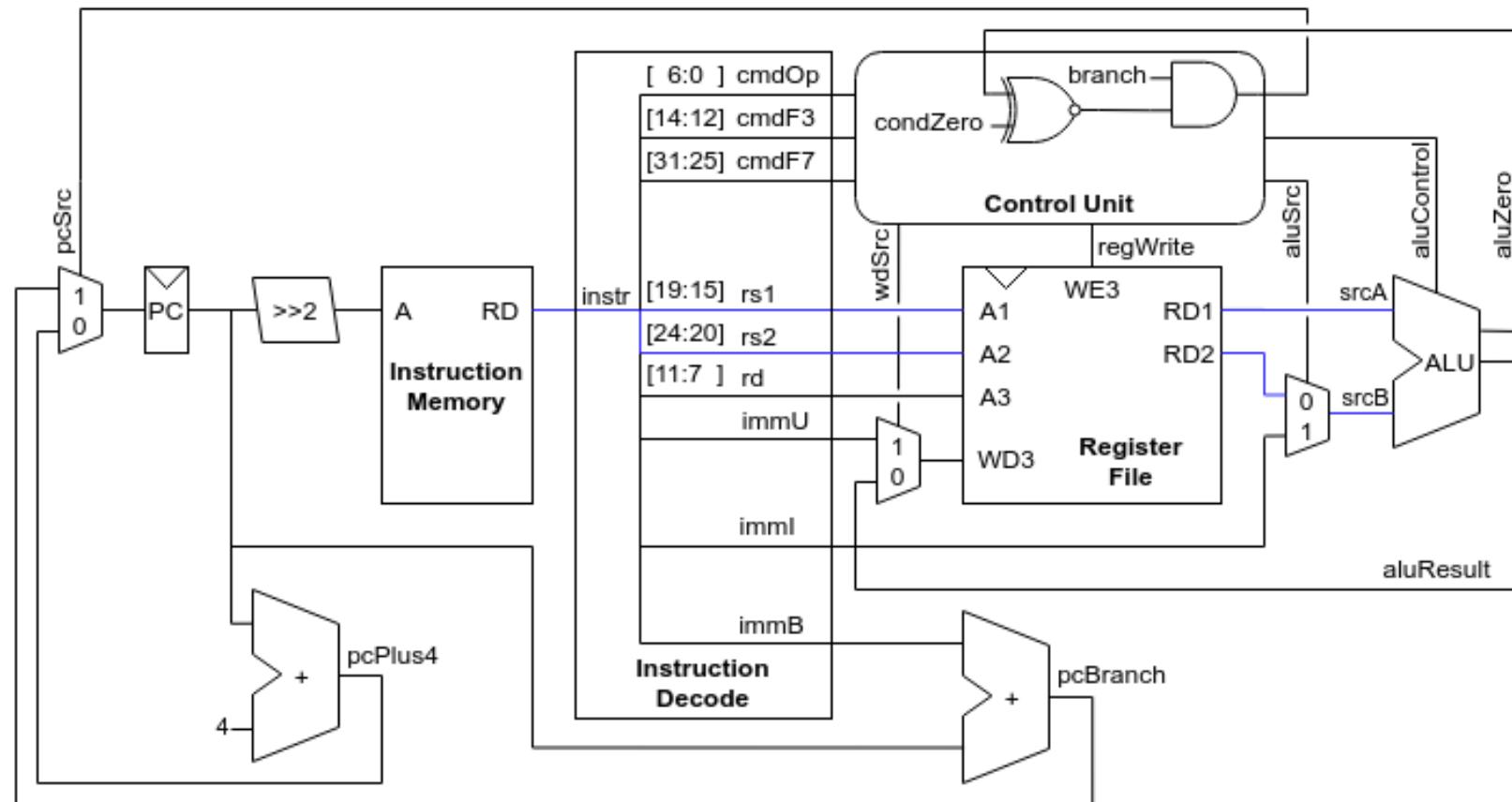


Сигналы управления 17



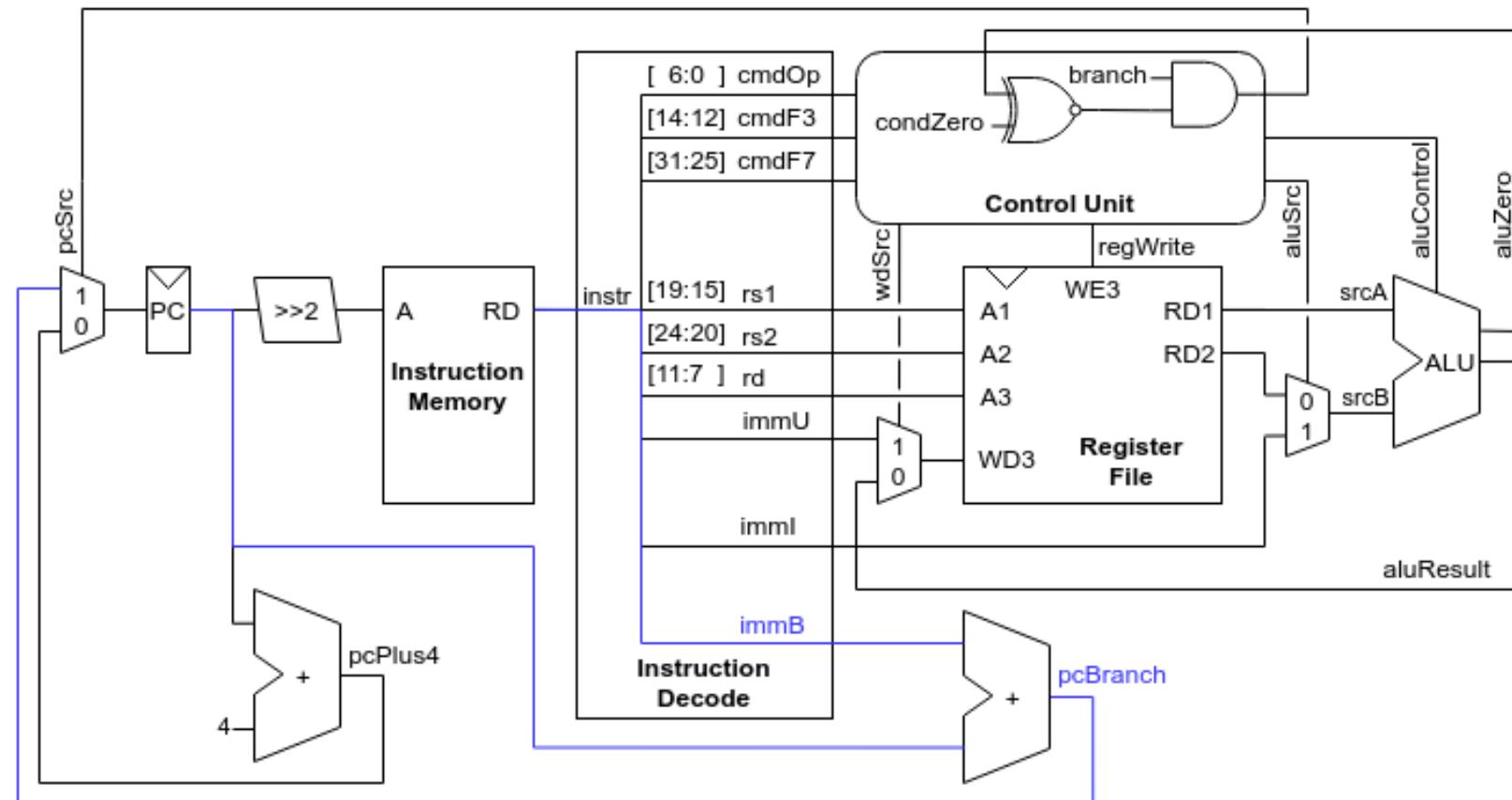
Сигналы управления 18

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0			



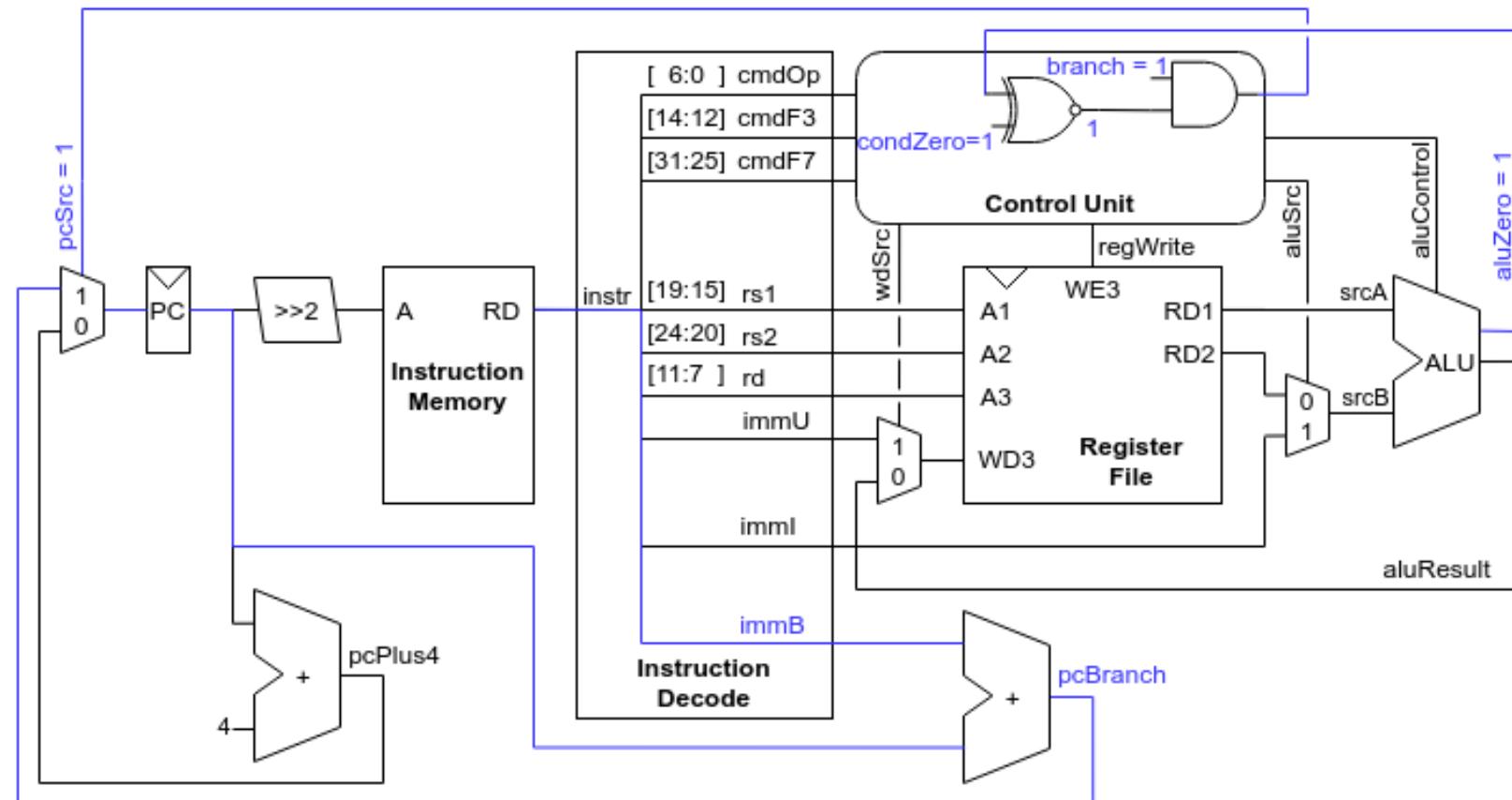
Сигналы управления 19

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0			



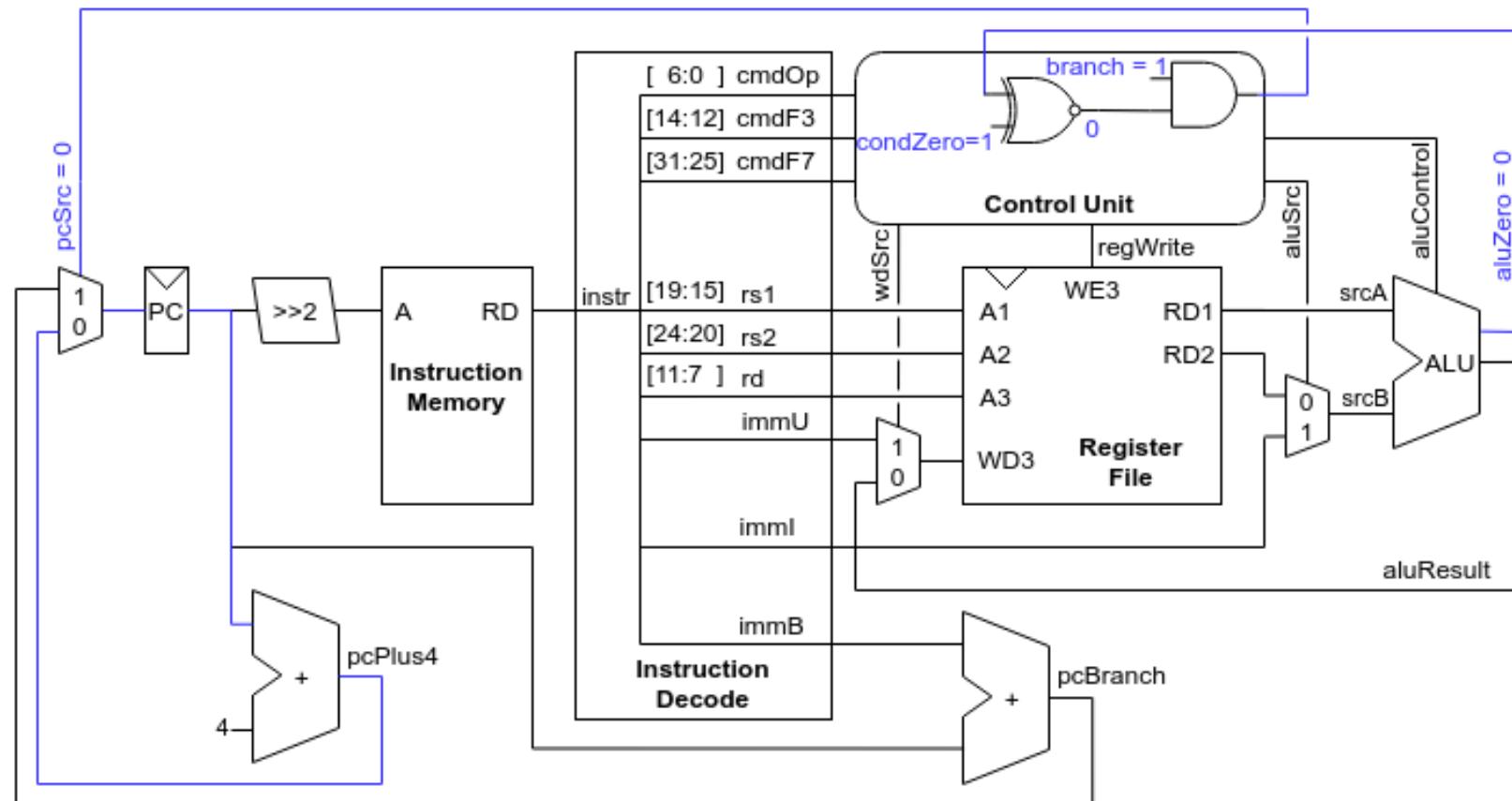
Сигналы управления 20

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1



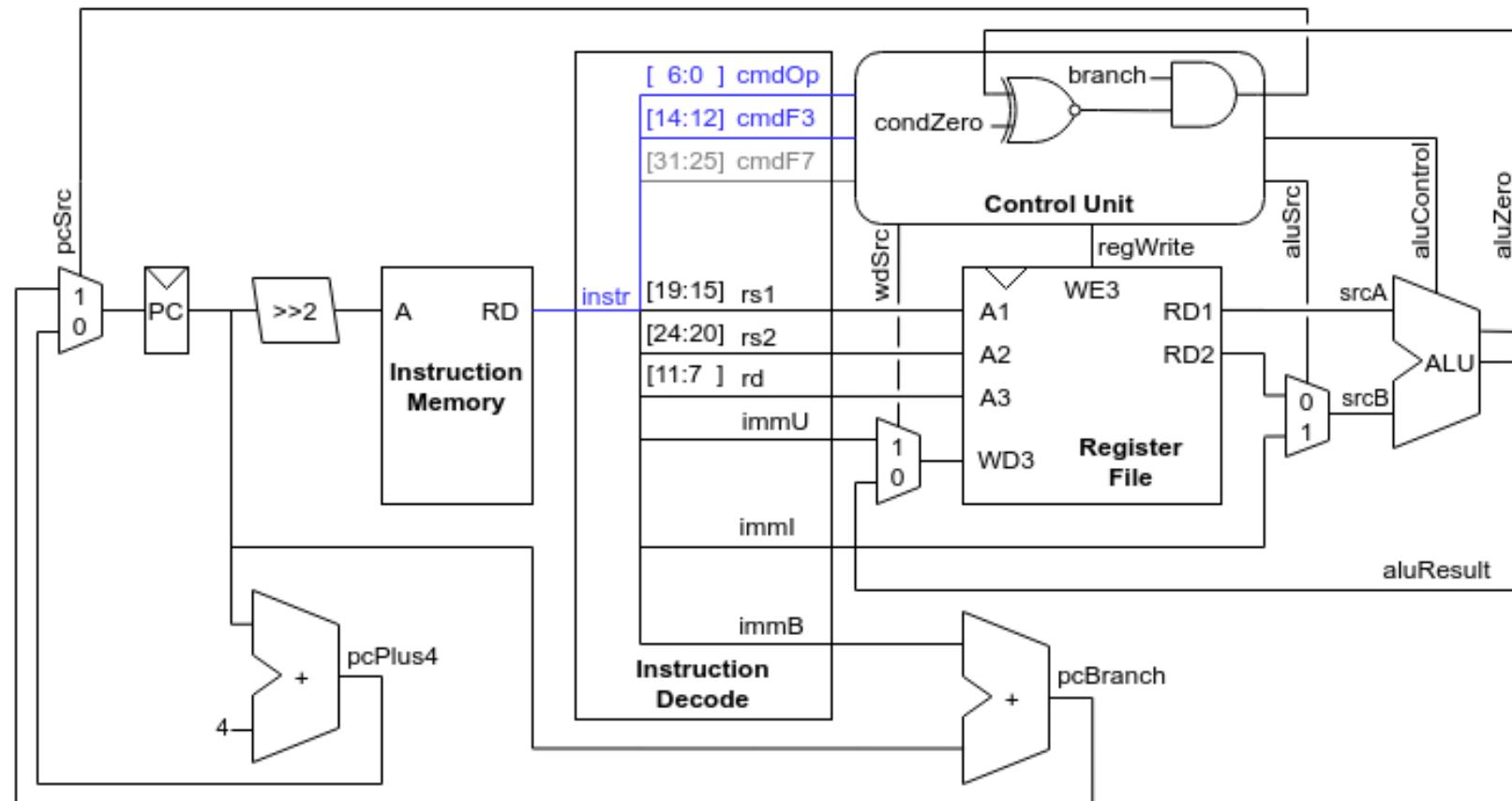
Сигналы управления 21

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1



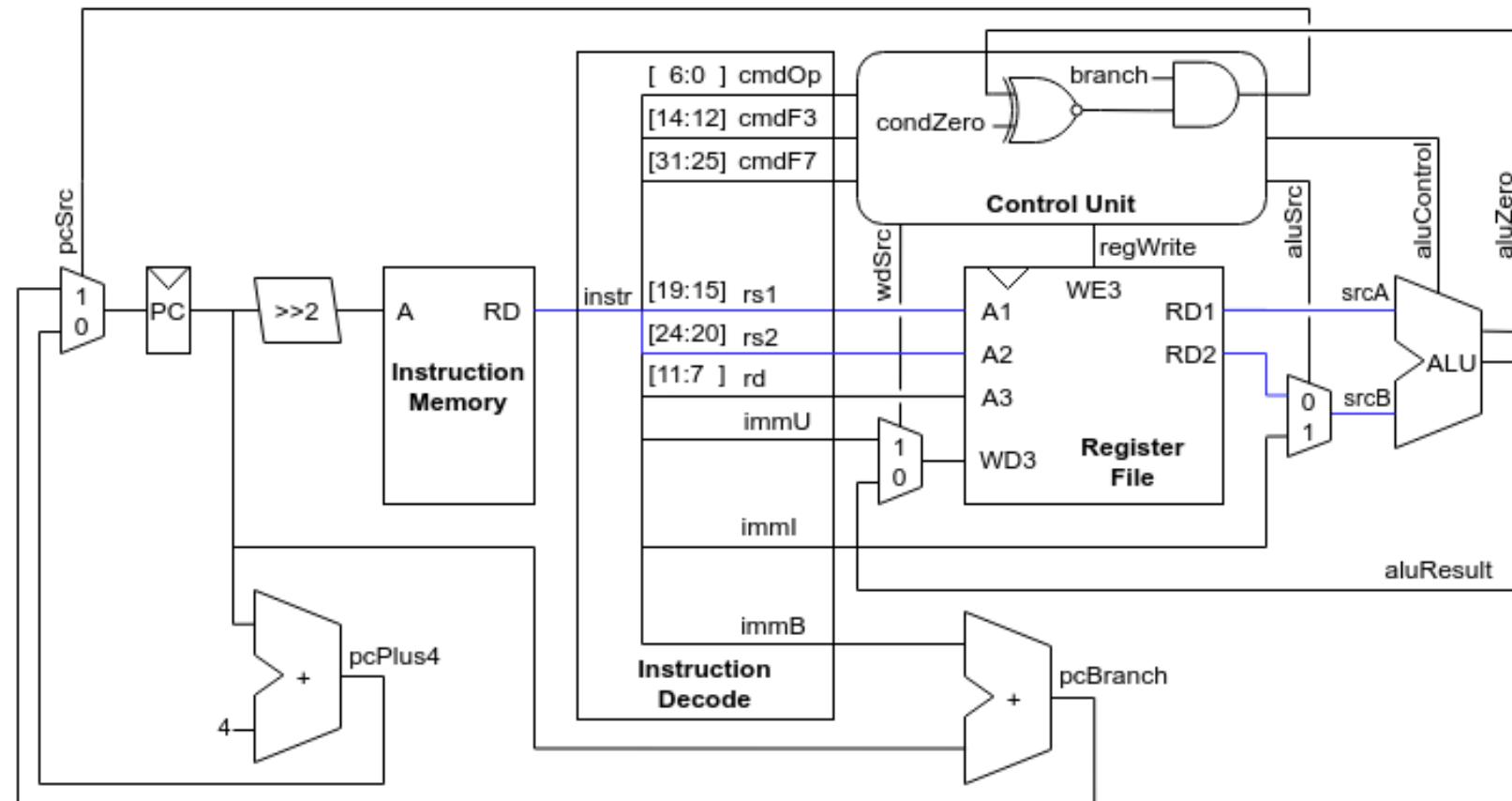
Сигналы управления 22

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1
BNE	1100011	001	???????							



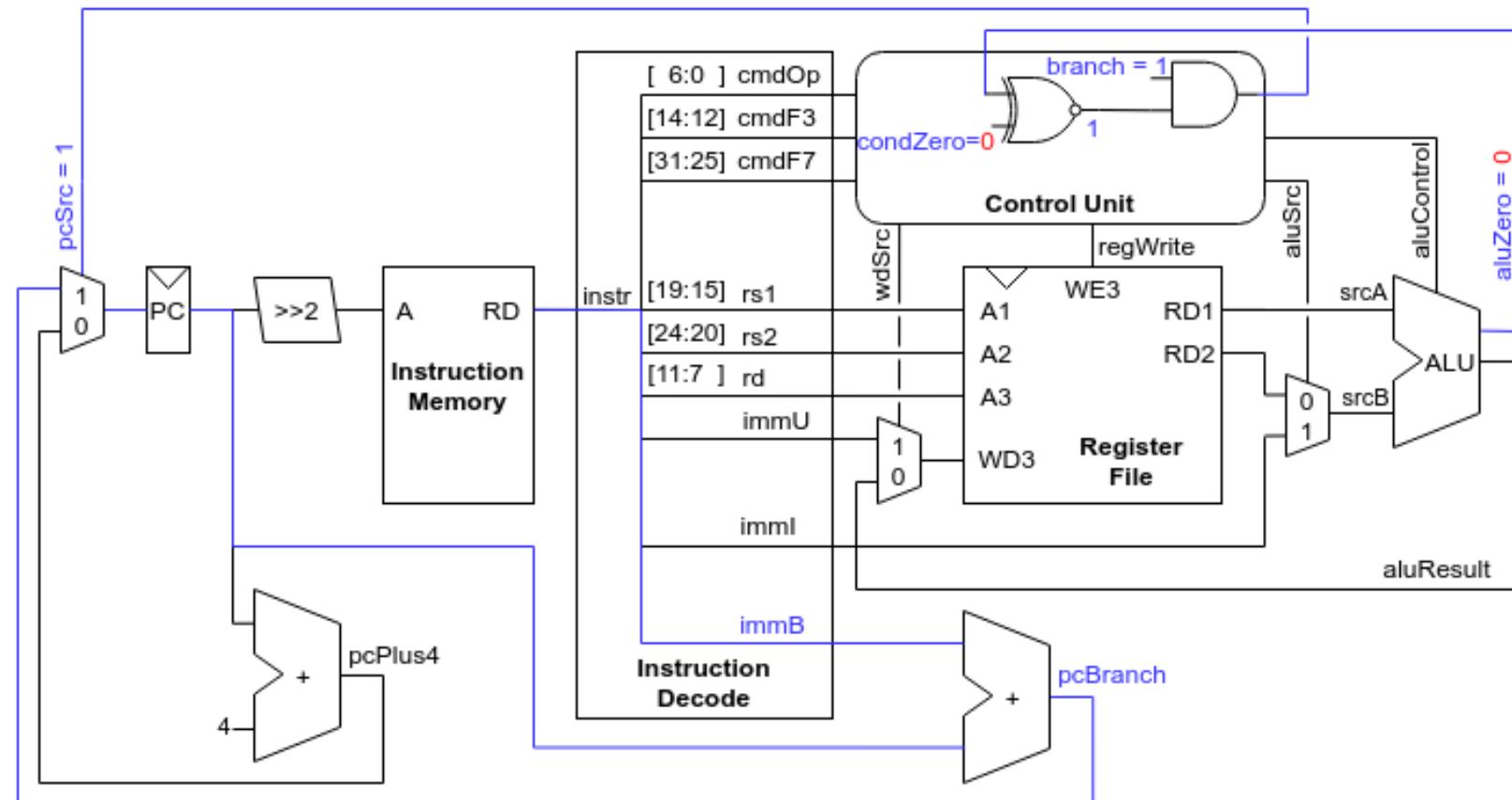
Сигналы управления 23

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1
BNE	1100011	001	???????	0	100	0	0			



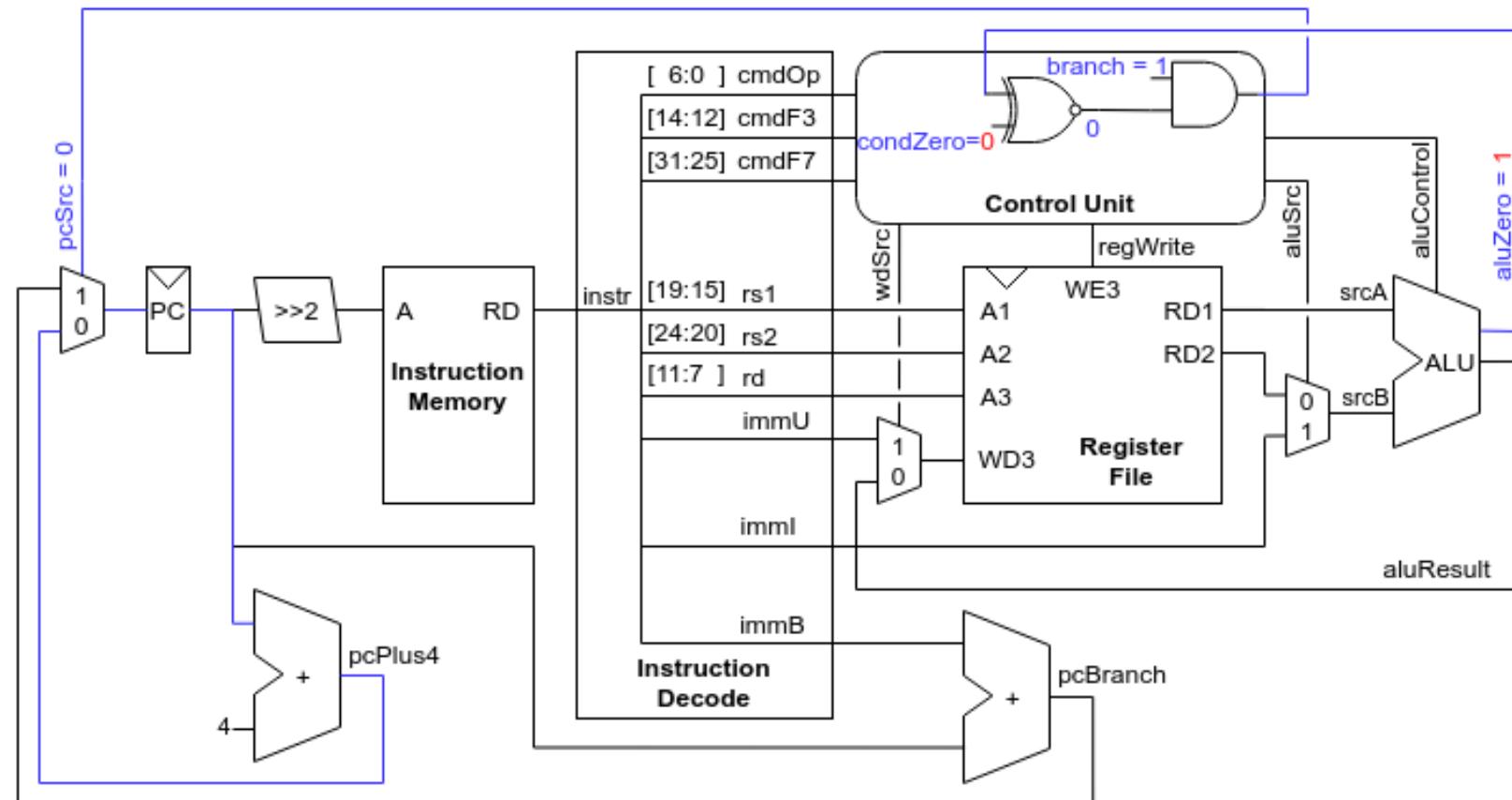
Сигналы управления 24

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1
BNE	1100011	001	???????	0	100	0	0	~aluZero	1	0



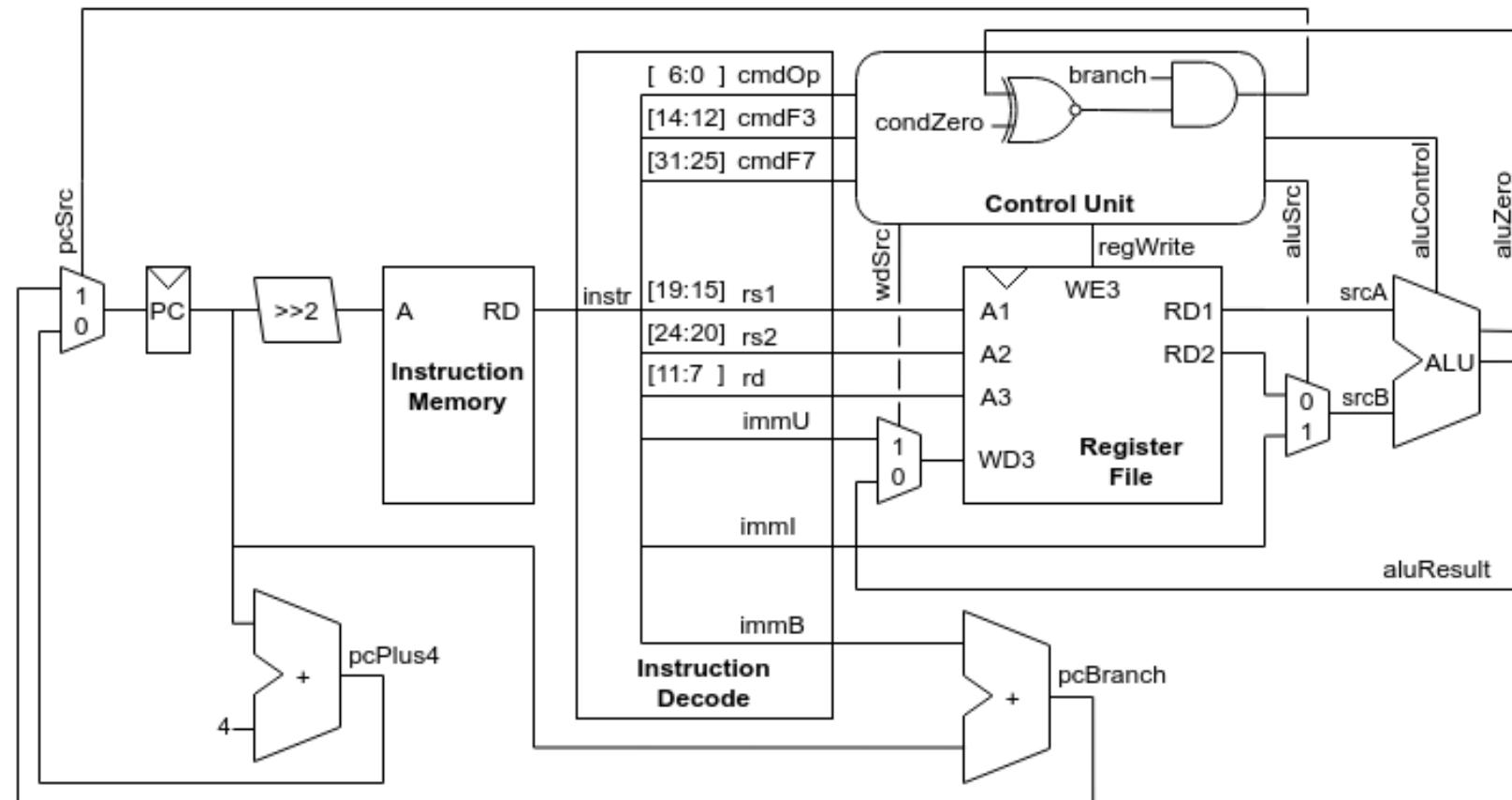
Сигналы управления 25

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1
BNE	1100011	001	???????	0	100	0	0	~aluZero	1	0



Сигналы управления 26

Instruction	cmdOp	cmdF3	cmdF7	aluSrc	aluControl	wdSrc	regWrite	pcSrc	branch	condZero
ADDI	0010011	000	???????	1	000	0	1	0	0	0
ADD	0110011	000	0000000	0	000	0	1	0	0	0
LUI	0110111	???	???????	0	000	1	1	0	0	0
BEQ	1100011	000	???????	0	100	0	0	aluZero	1	1
BNE	1100011	001	???????	0	100	0	0	~aluZero	1	0

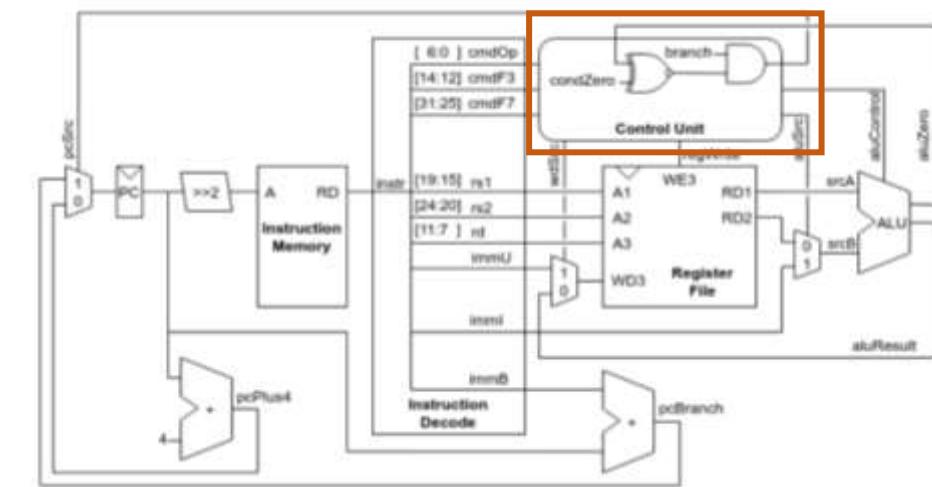


Реализация: коды инструкций

```
// sr_cpu.vh
// instruction opcode
`define RVOP_ADDI    7'b0010011
`define RVOP_BEQ     7'b1100011
...
// instruction funct3
`define RVF3_ADDI    3'b000
`define RVF3_BEQ     3'b000
`define RVF3_BNE     3'b001
`define RVF3_ADD     3'b000
...
`define RVF3_ANY     3'b???
// instruction funct7
`define RVF7_ADD     7'b0000000
...
`define RVF7_ANY     7'b???????
```

Реализация: устройство управления (начало)

```
// sr_cpu.v
module sr_control
(
    input      [ 6:0] cmdOp,
    input      [ 2:0] cmdF3,
    input      [ 6:0] cmdF7,
    input          aluZero,
    output          pcSrc,
    output reg     regWrite,
    output reg     aluSrc,
    output reg     wdSrc,
    output reg [2:0] aluControl
);
    reg          branch;
    reg          condZero;
    assign pcSrc = branch & (aluZero == condZero);
```



Реализация: устройство управления (продолжение)

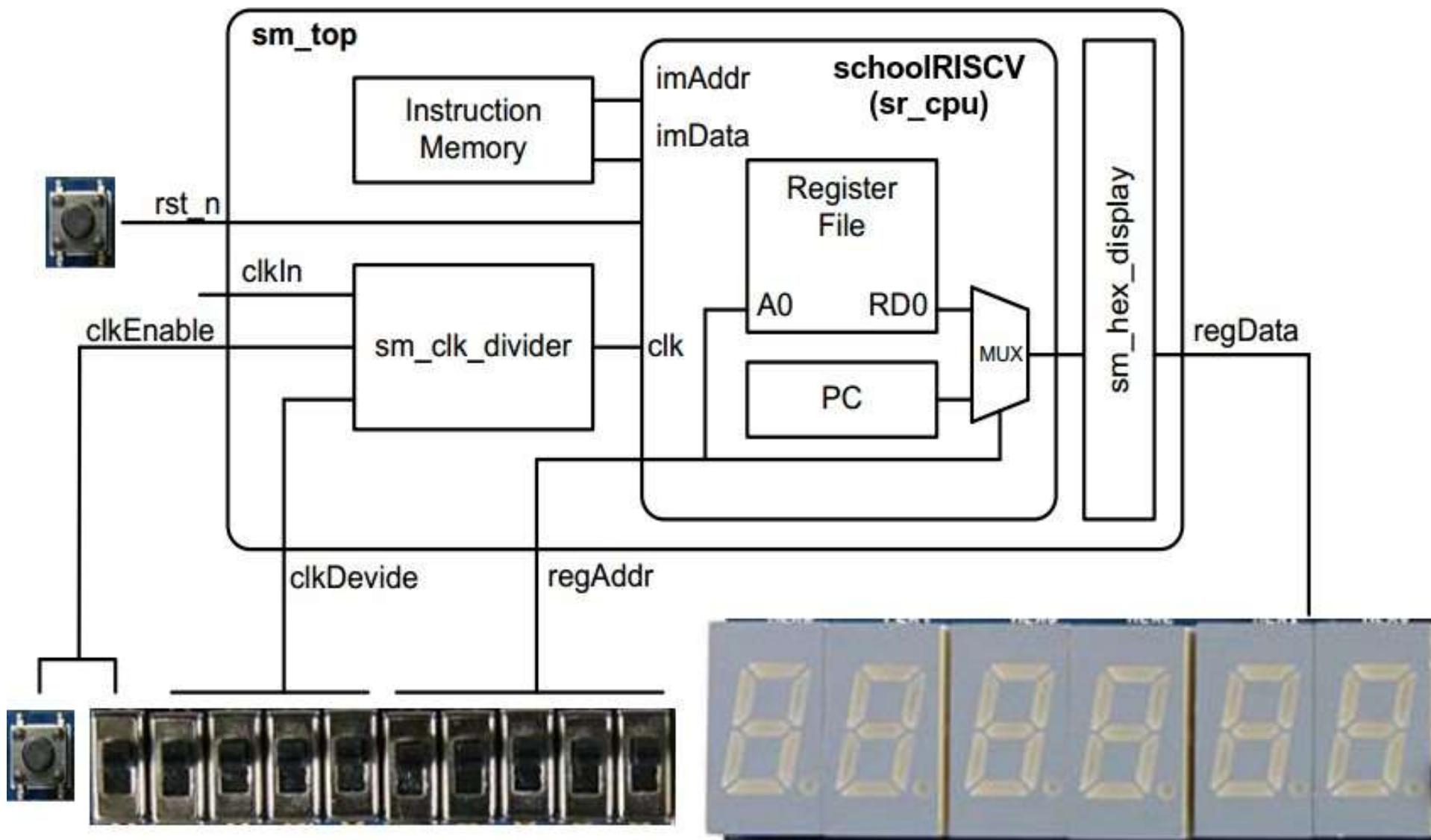
```
// sr_cpu.v
always @ (*) begin
    branch      = 1'b0;
    condZero   = 1'b0;
    regWrite   = 1'b0;
    aluSrc     = 1'b0;
    wdSrc      = 1'b0;
    aluControl = `ALU_ADD;

    casez( {cmdF7, cmdF3, cmdOp} )
        { `RVF7_ADD, `RVF3_ADD, `RVOP_ADD } : begin regWrite = 1'b1; aluControl = `ALU_ADD; end
        { `RVF7_OR,  `RVF3_OR,  `RVOP_OR  } : begin regWrite = 1'b1; aluControl = `ALU_OR;  end
        { `RVF7_SRL, `RVF3_SRL, `RVOP_SRL } : begin regWrite = 1'b1; aluControl = `ALU_SRL; end
        { `RVF7_SLTU, `RVF3_SLTU, `RVOP_SLTU } : begin regWrite = 1'b1; aluControl = `ALU_SLTU; end
        { `RVF7_SUB,  `RVF3_SUB,  `RVOP_SUB  } : begin regWrite = 1'b1; aluControl = `ALU_SUB; end

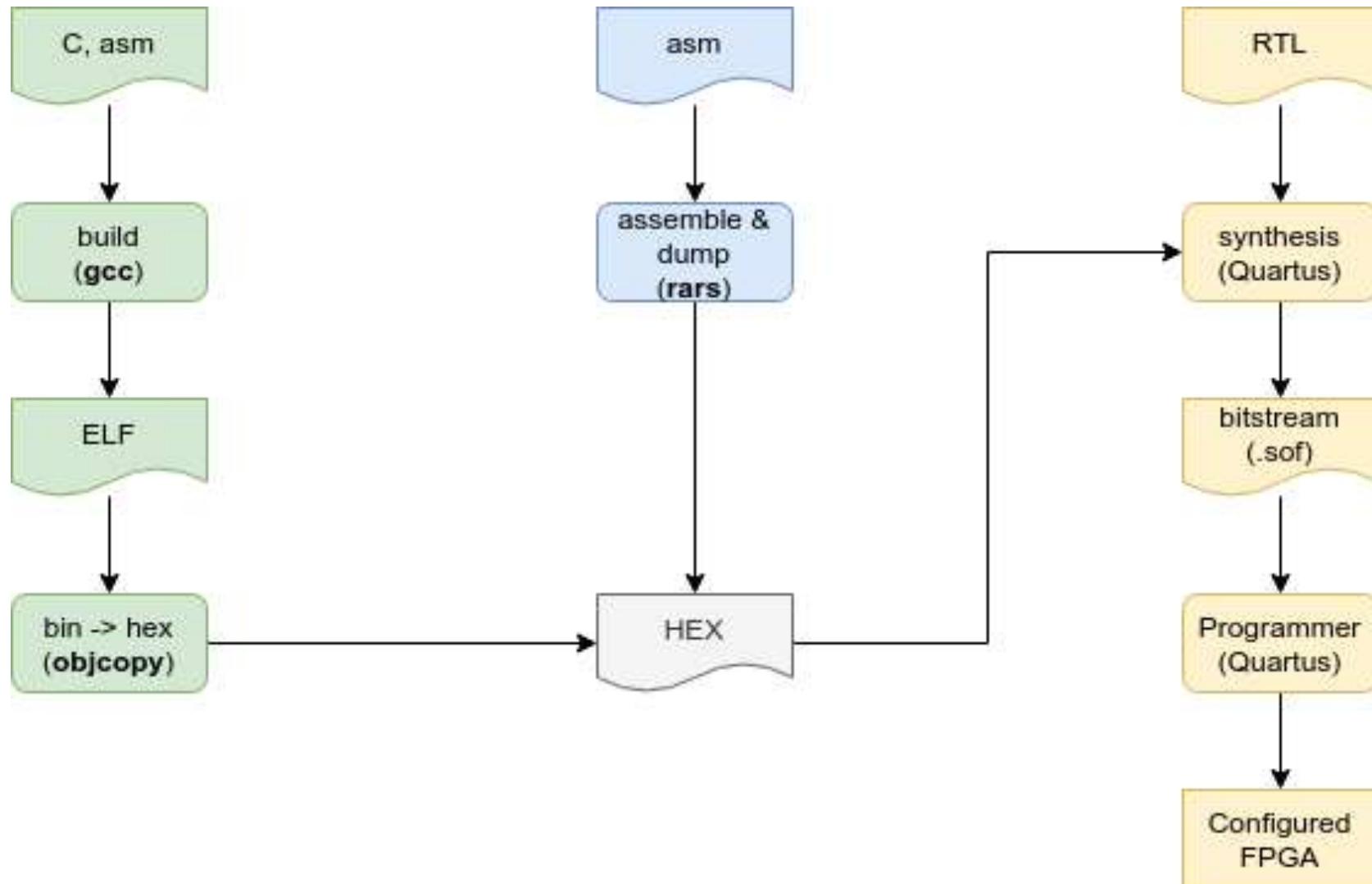
        { `RVF7_ANY,  `RVF3_ADDI, `RVOP_ADDI } : begin regWrite = 1'b1; aluSrc = 1'b1; aluControl = `ALU_ADD; end
        { `RVF7_ANY,  `RVF3_ANY,   `RVOP_LUI  } : begin regWrite = 1'b1; wdSrc  = 1'b1; end

        { `RVF7_ANY,  `RVF3_BEQ,  `RVOP_BEQ  } : begin branch = 1'b1; condZero = 1'b1; aluControl = `ALU_SUB; end
        { `RVF7_ANY,  `RVF3_BNE,  `RVOP_BNE  } : begin branch = 1'b1; aluControl = `ALU_SUB; end
    endcase
end
```

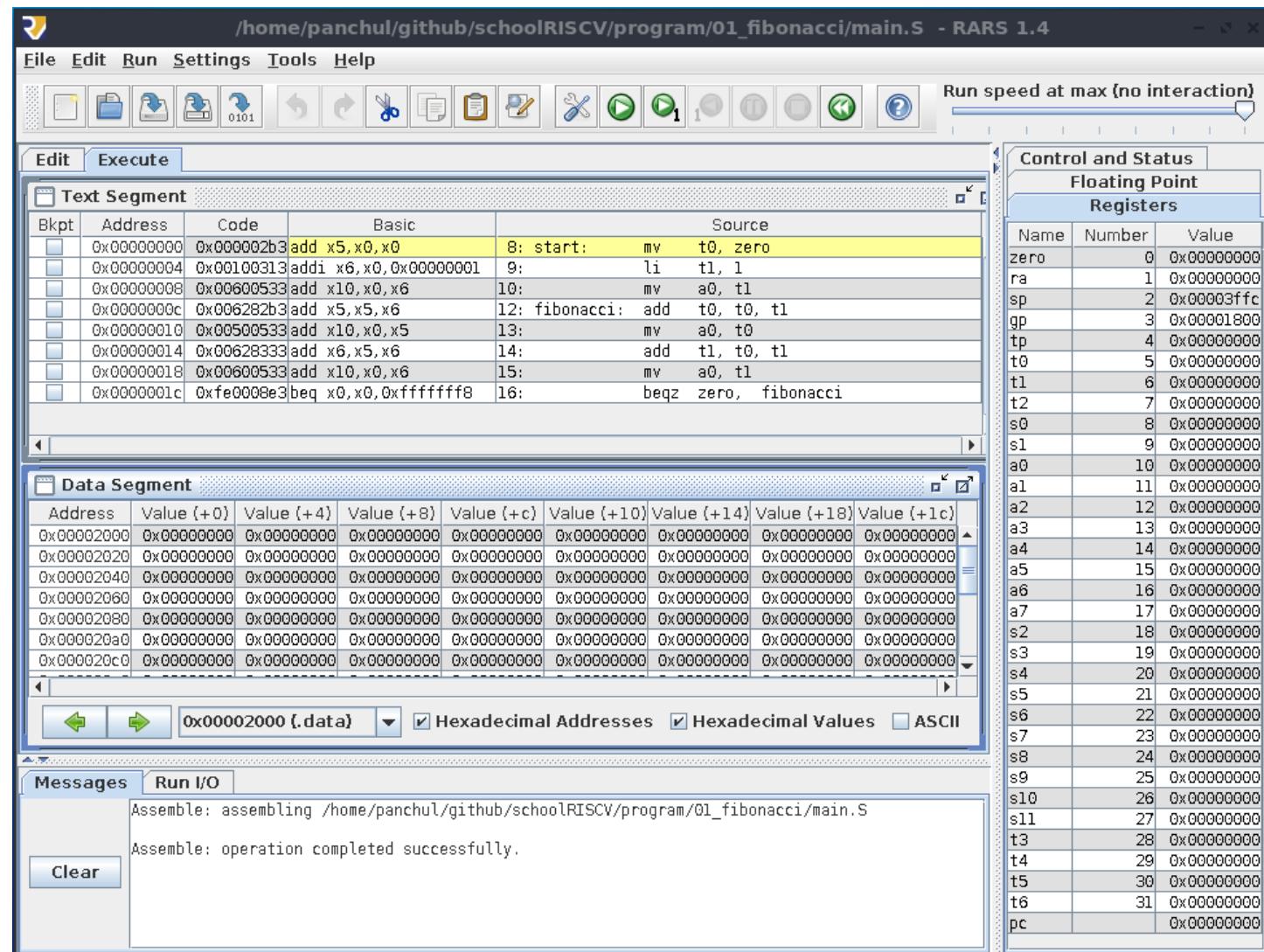
Структура проекта и подключение периферии ПЛИС.



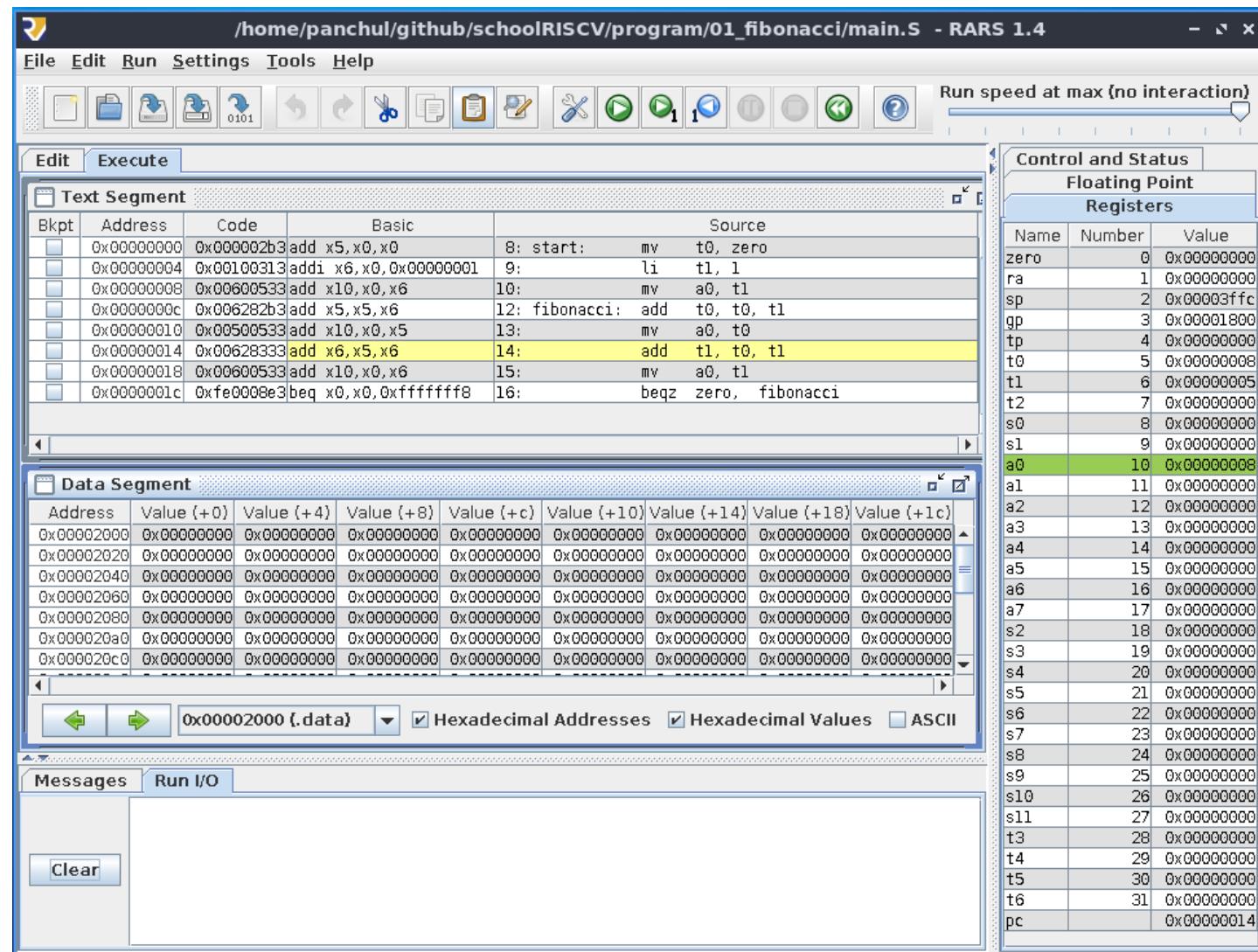
Программирование системы



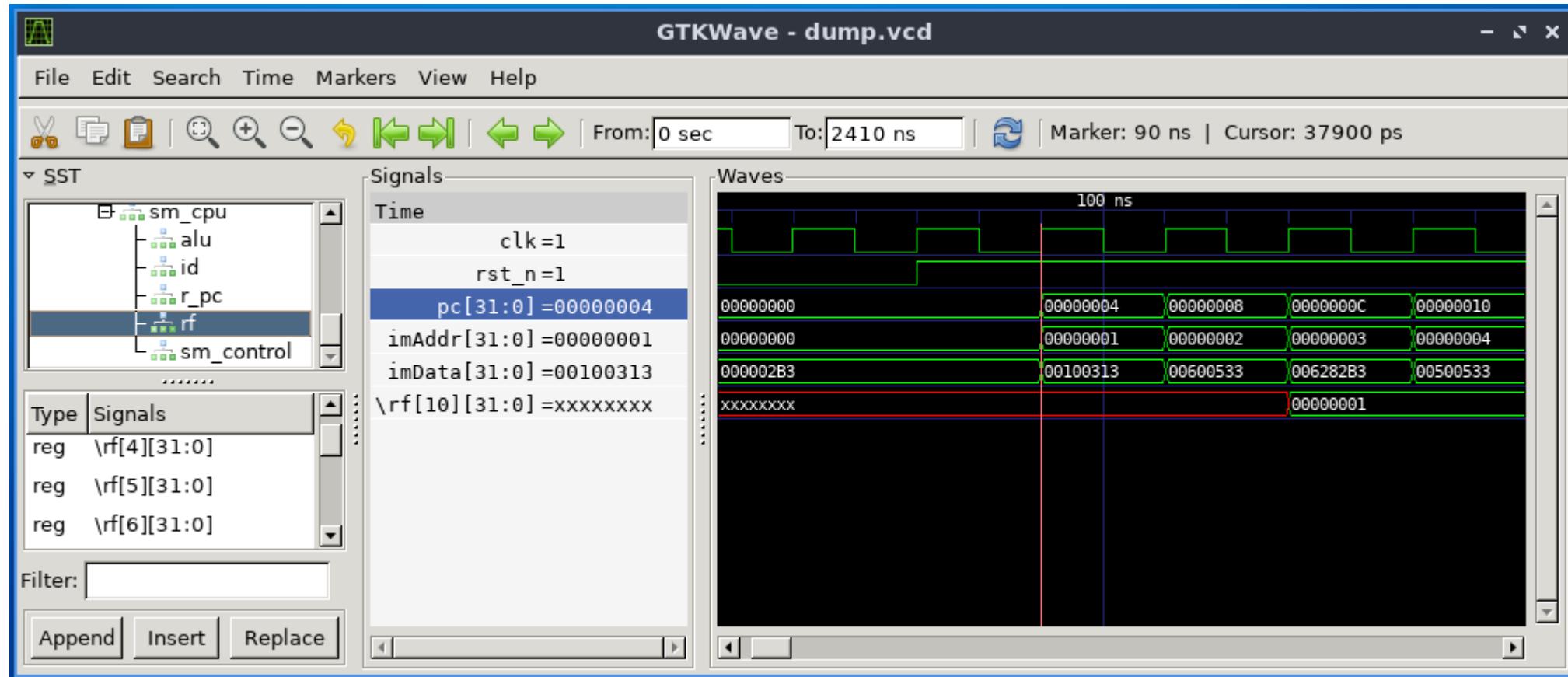
Запуск в симуляторе RARS



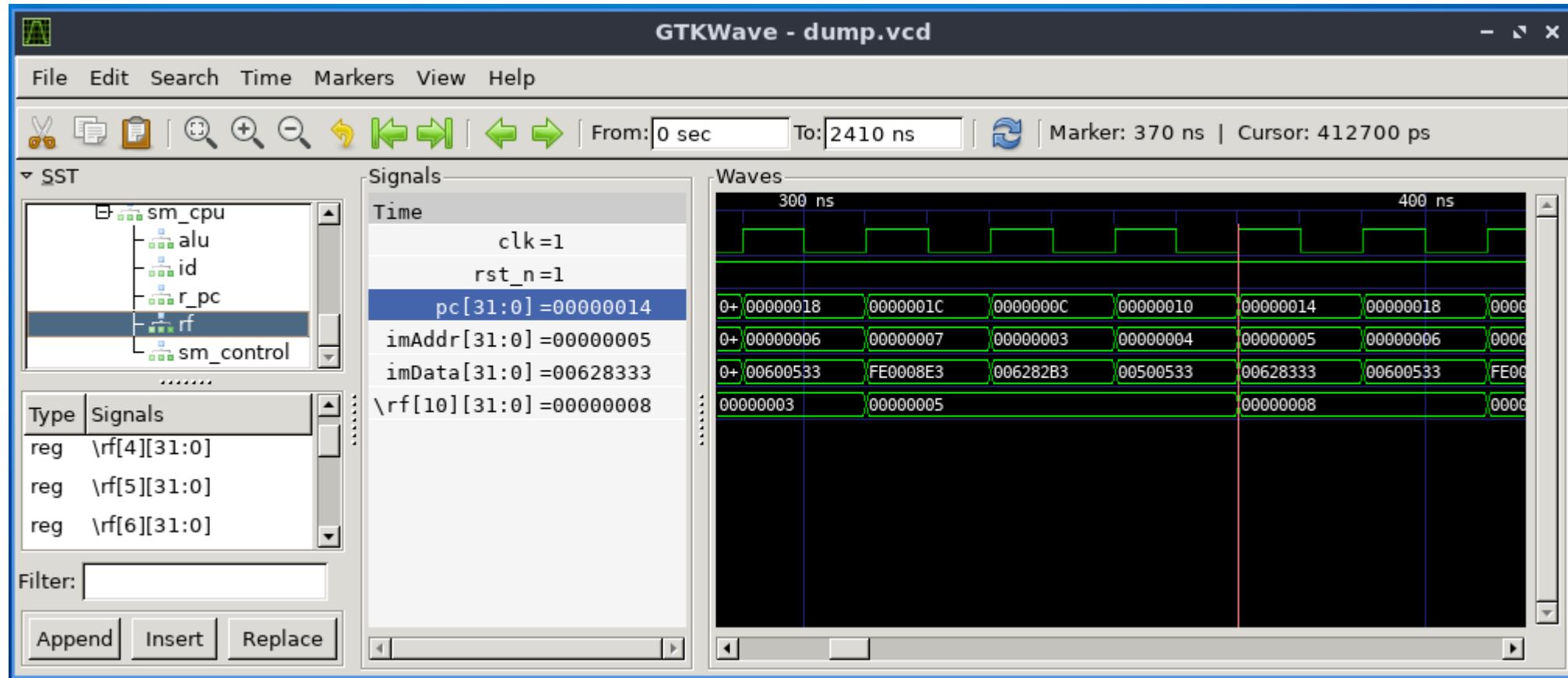
Запуск в симуляторе RARS



Запуск в симуляторе iverilog/gtkwave



Запуск в симуляторе iverilog/gtkwave

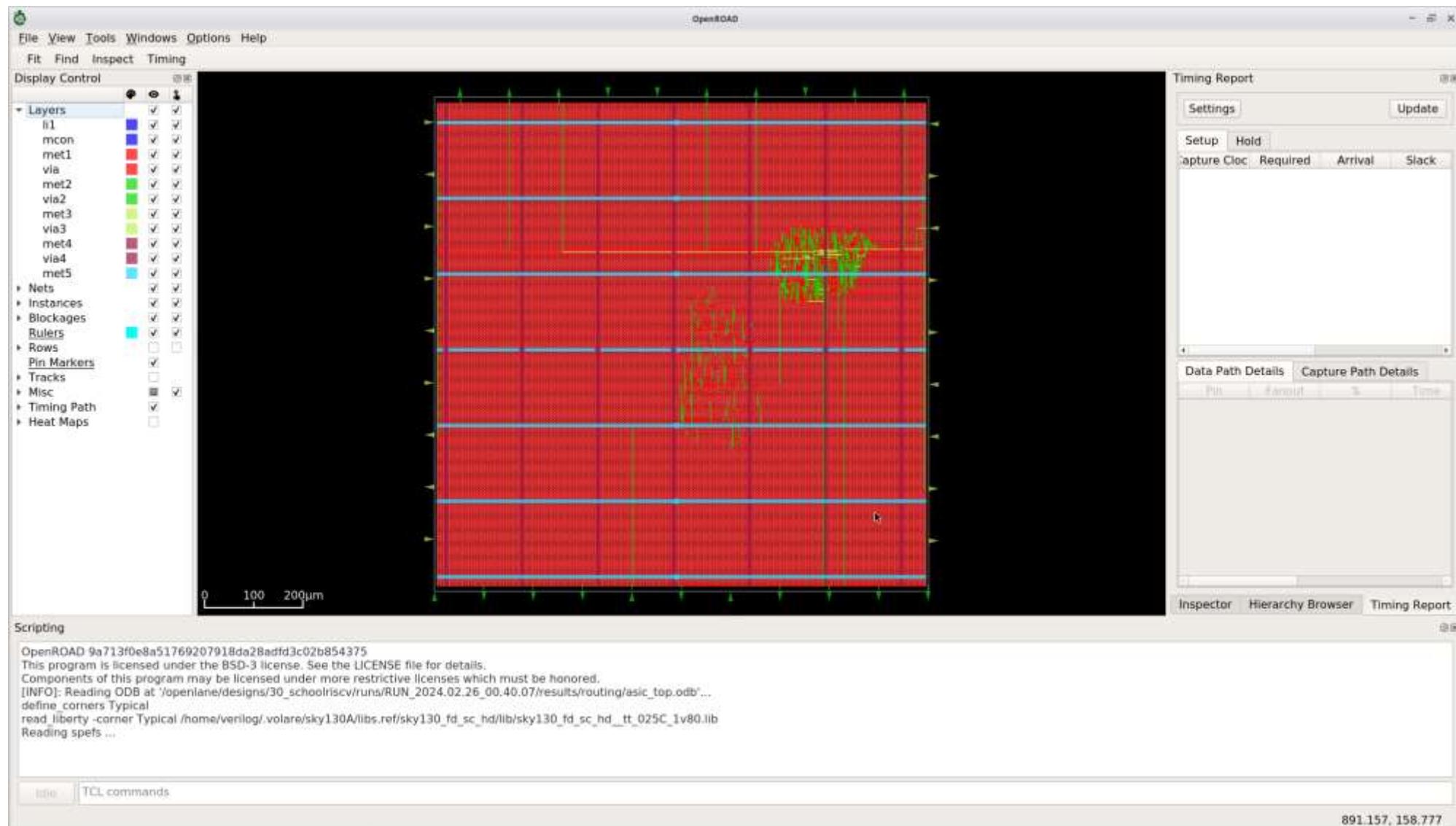


Использование assembler RISC-V и отладка кода с помощью ПЛИС

45 минут

- Маршрут проектирования микросхем. Повторение синтаксиса языка Verilog
- Описание комбинационной и последовательностной логики на языке Verilog
- Введение в архитектуру RISC-V.
- Практика. Использование assembler RISC-V(45 минут)
- Введение в микроархитектуру RISC-V.
- **Использование assembler RISC-V и отладка кода с помощью ПЛИС(45 минут).**

Синтез в ASIC(OpenLane)



Синтез в ASIC(OpenLane)

