1

Verilog HDL. АЛУ

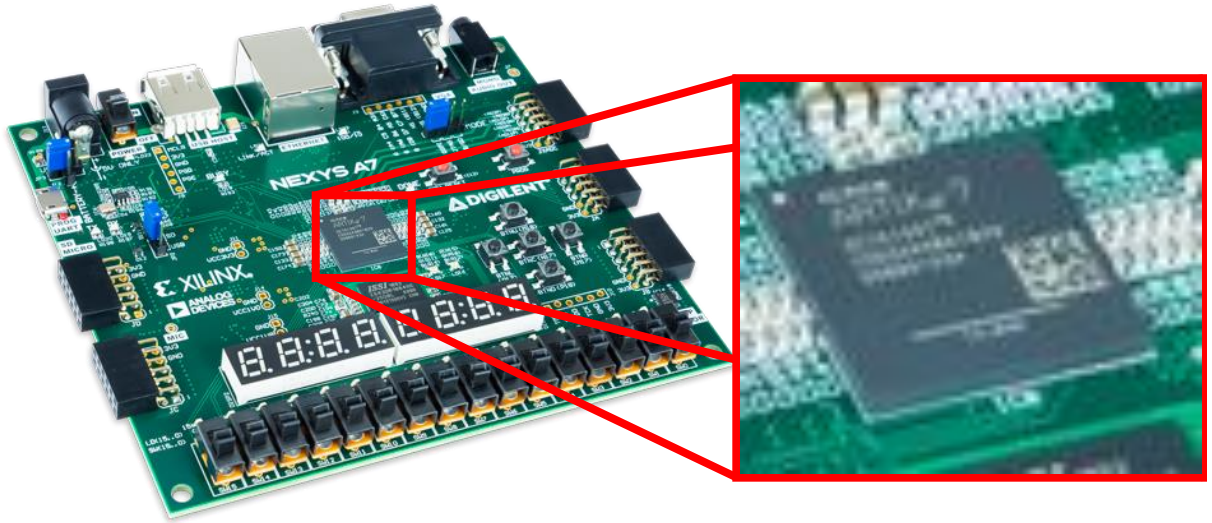
Архитектуры микропроцессорных систем и средств

План лабораторной работы

- 1 пара
  - О лабораторных работах (**T**)
  - Введение в FPGA и Verilog HDL (**T**)
  - Тренинг по Vivado и Verilog HDL (**TS**)
- 2 пара
  - Арифметико-логическое устройство (**T**)
  - Описание АЛУ на Verilog HDL (**S**)
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

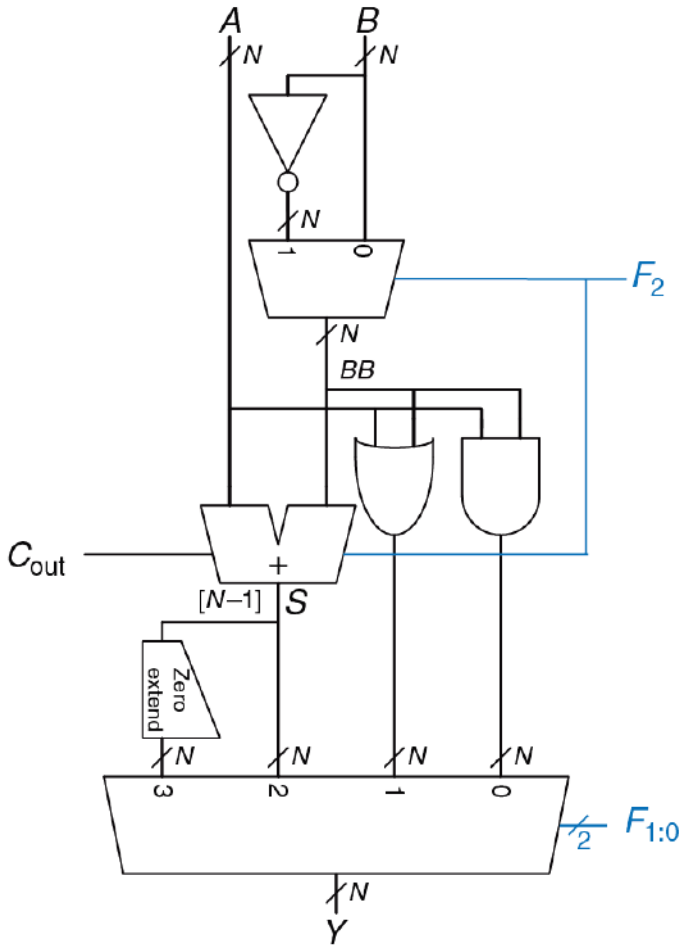
Цель лабораторных работ

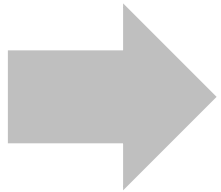
- Используя Verilog HDL реализовать на базе FPGA программируемый процессор с архитектурой RISC-V

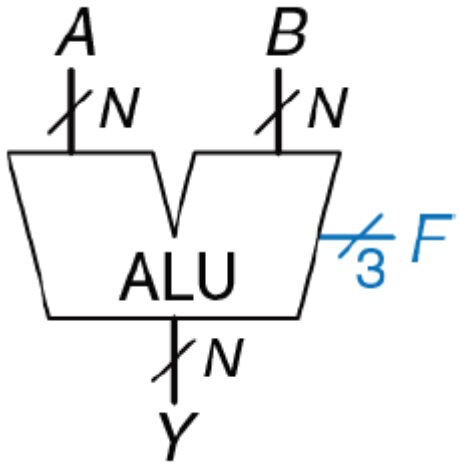



```
module fulladder (a, b, cin, s, cout);

input  a, b, cin;
output s, cout;

   wire p, g;

   assign p = a ^ b;
   assign g = a & b;
   assign s = p ^ cin;
   assign cout = g |(p & cin);

endmodule
```

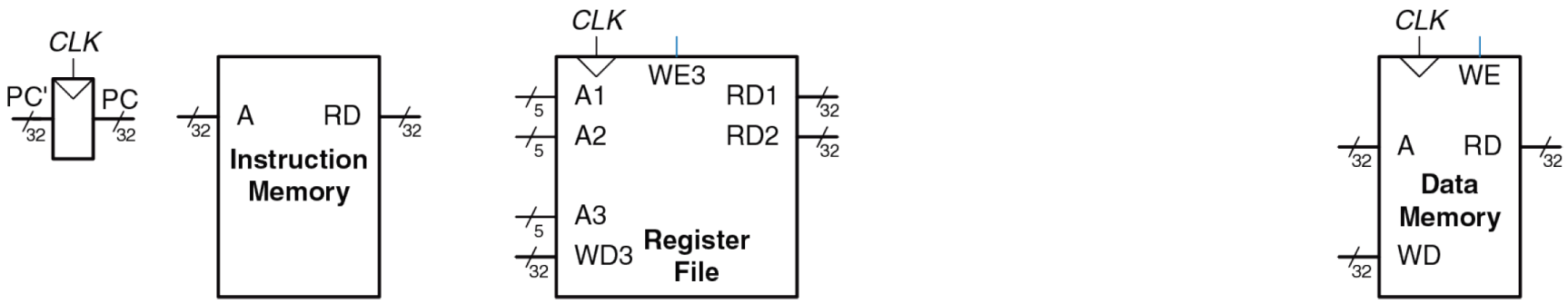
ЛР1. Арифметико-логическое устройство



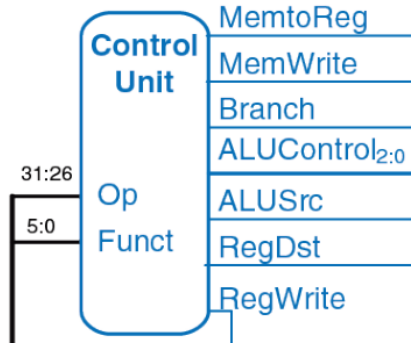




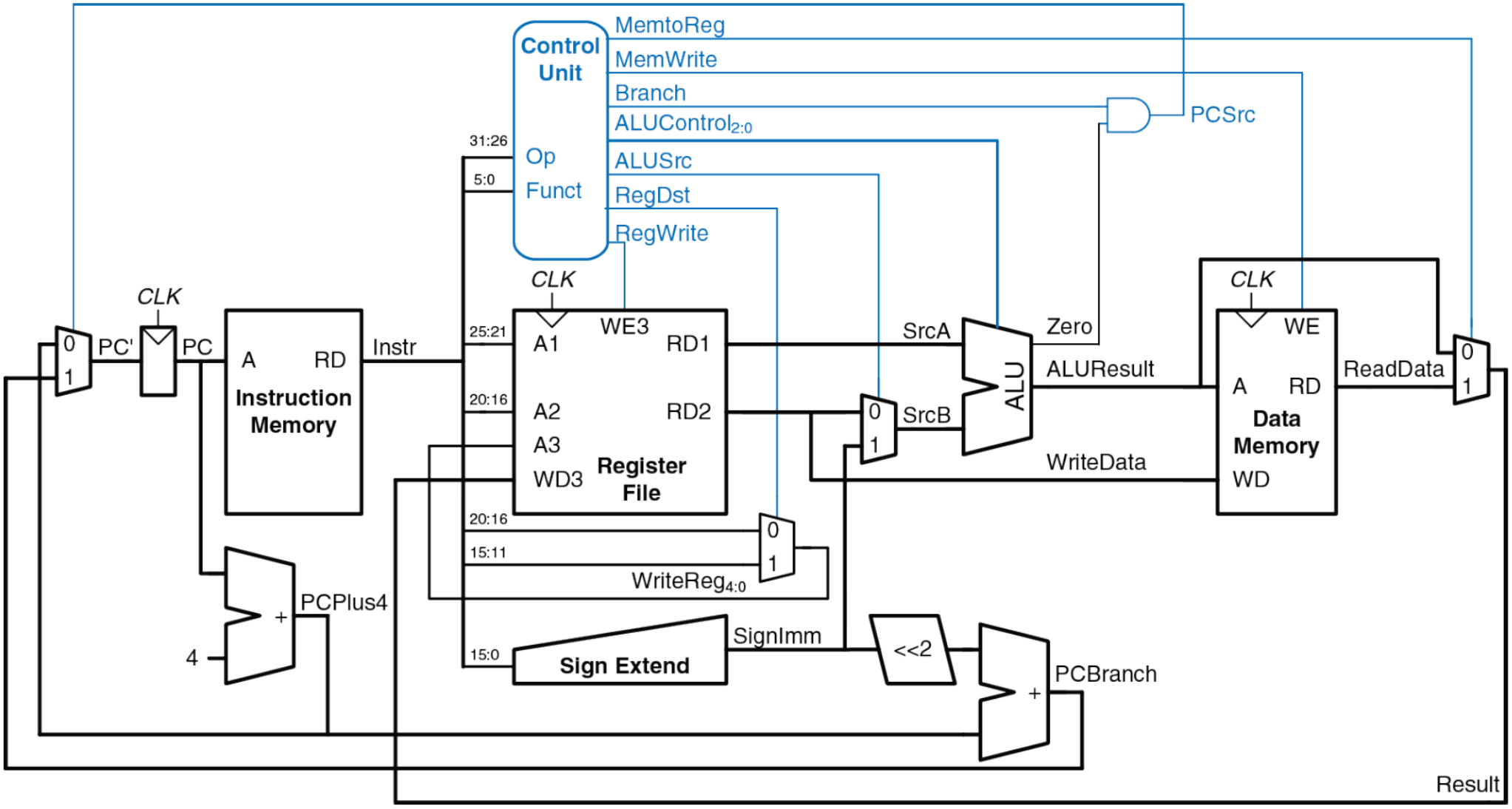
ЛР2. Регистровый файл. Память. Программируемое устройство



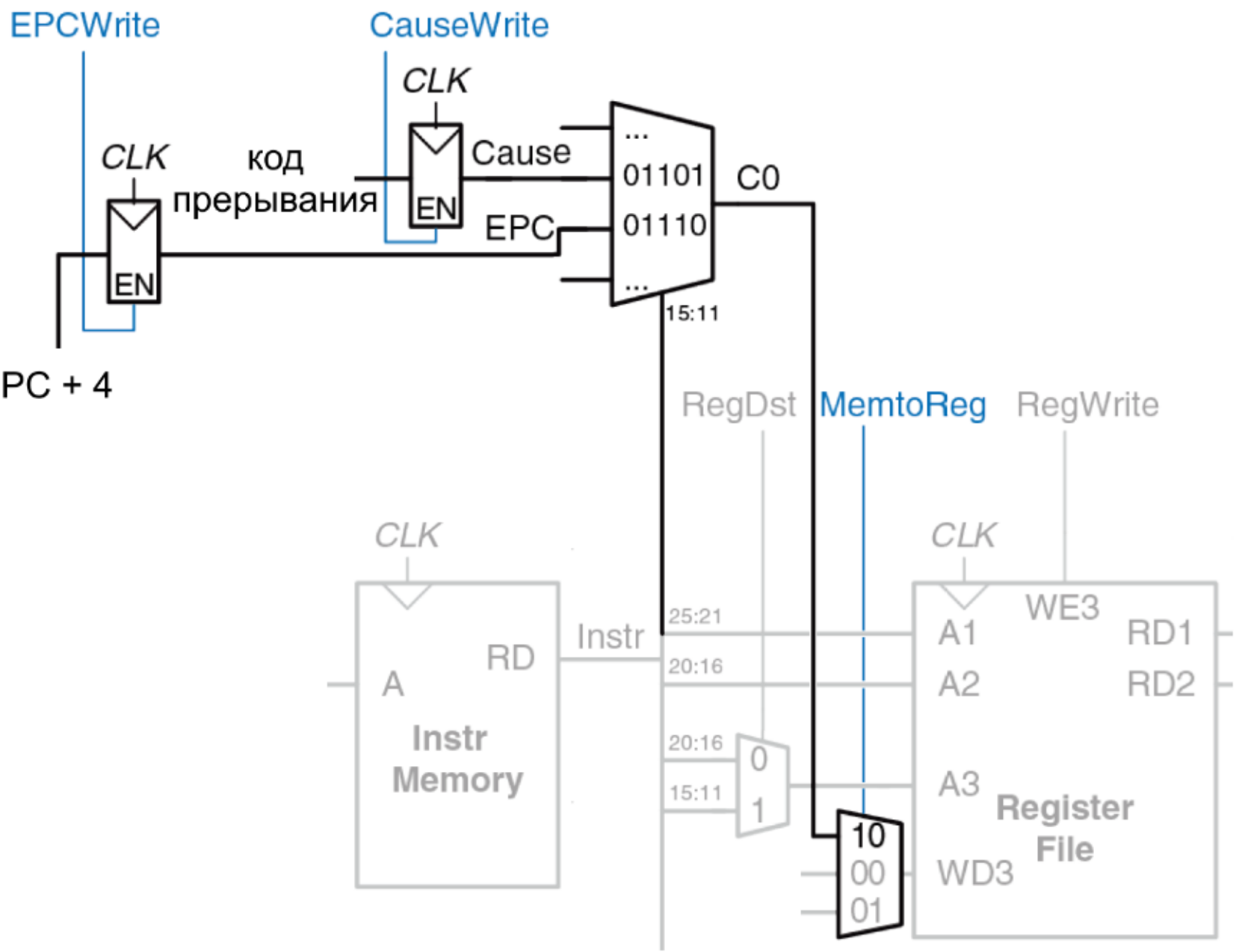
ЛР3. Устройство управления RISC-V



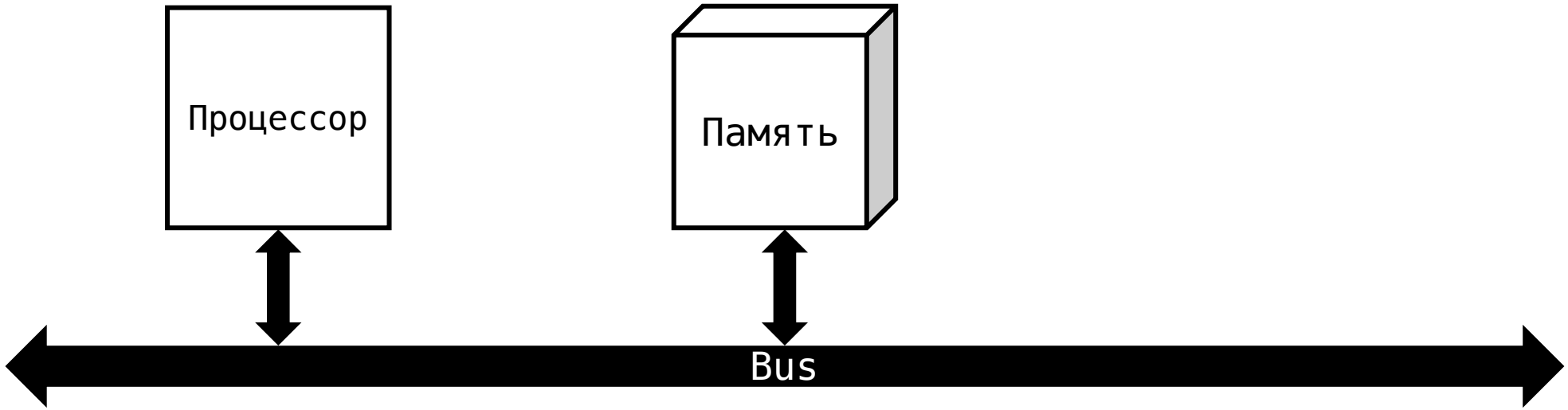
ЛР4. Тракт данных RISC-V



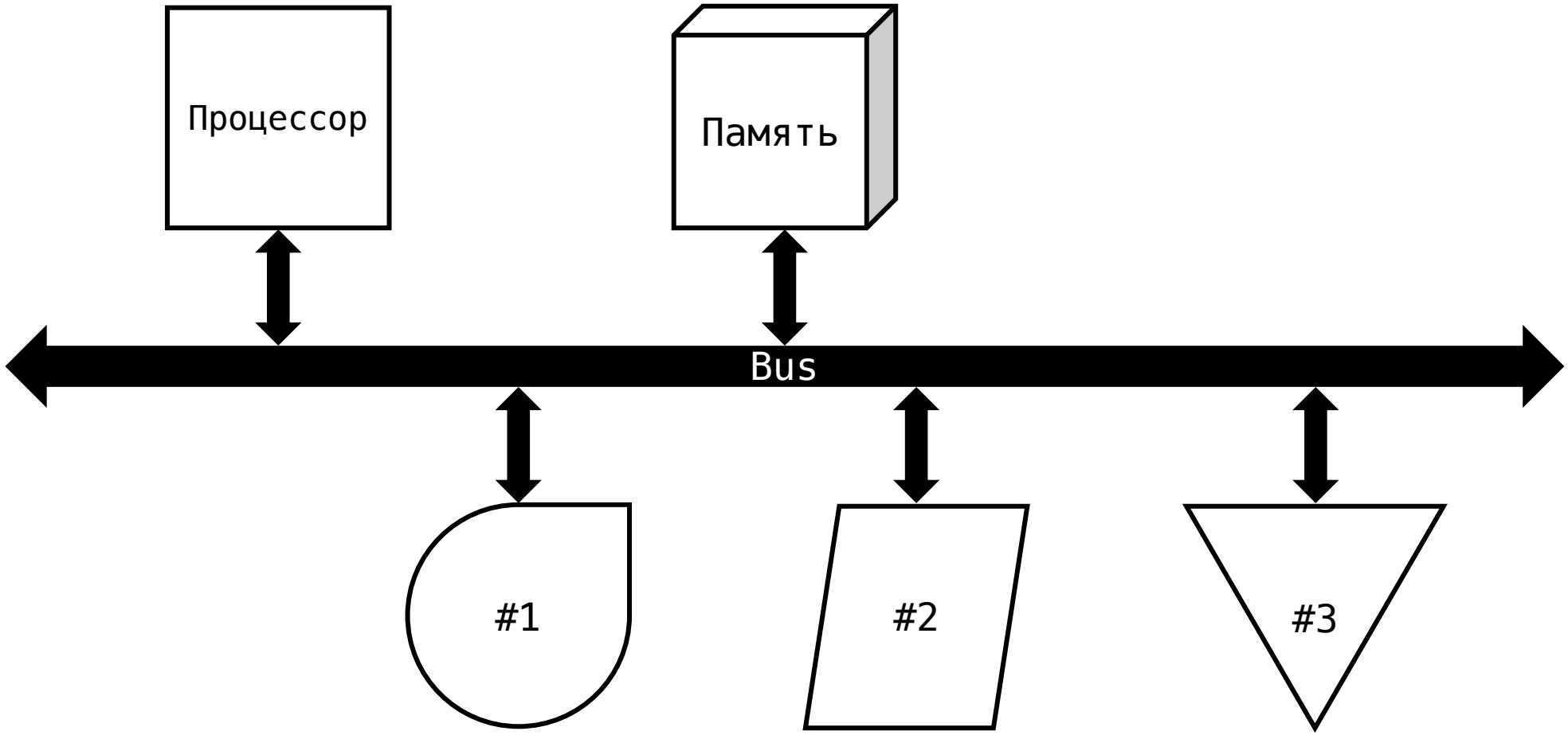
ния


EPCWrite
CauseWrite
CLK
CLK
код
прерывания
Cause
EPC
...
01101
01110
...
C0
EN
EN
15:11
PC + 4
RegDst
MemtoReg
RegWrite
CLK
CLK
Instr
RD
A
Instr
Memory
25:21
20:16
20:16
15:11
0
1
10
00
01
WE3
A1
A2
A3
WD3
RD1
RD2
Register
File

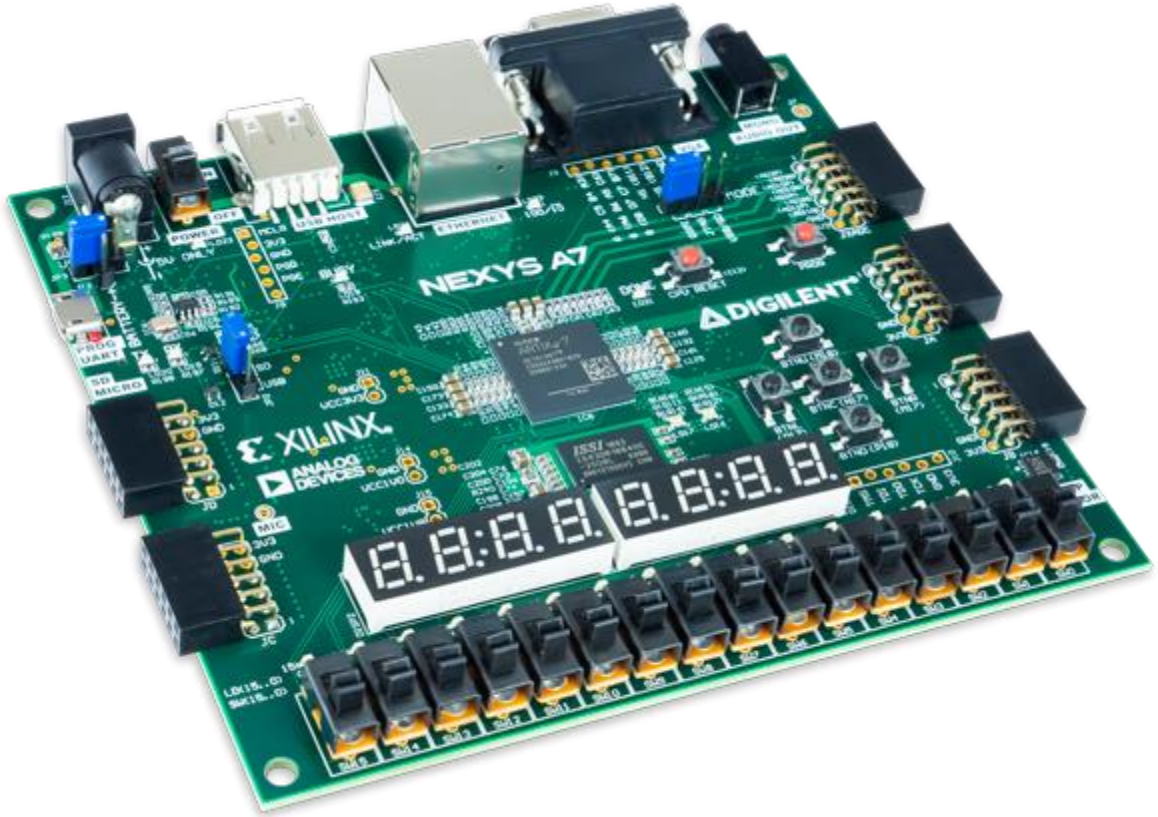
ЛР6. Память и шина



ЛР7. Ввод\вывод. Периферия

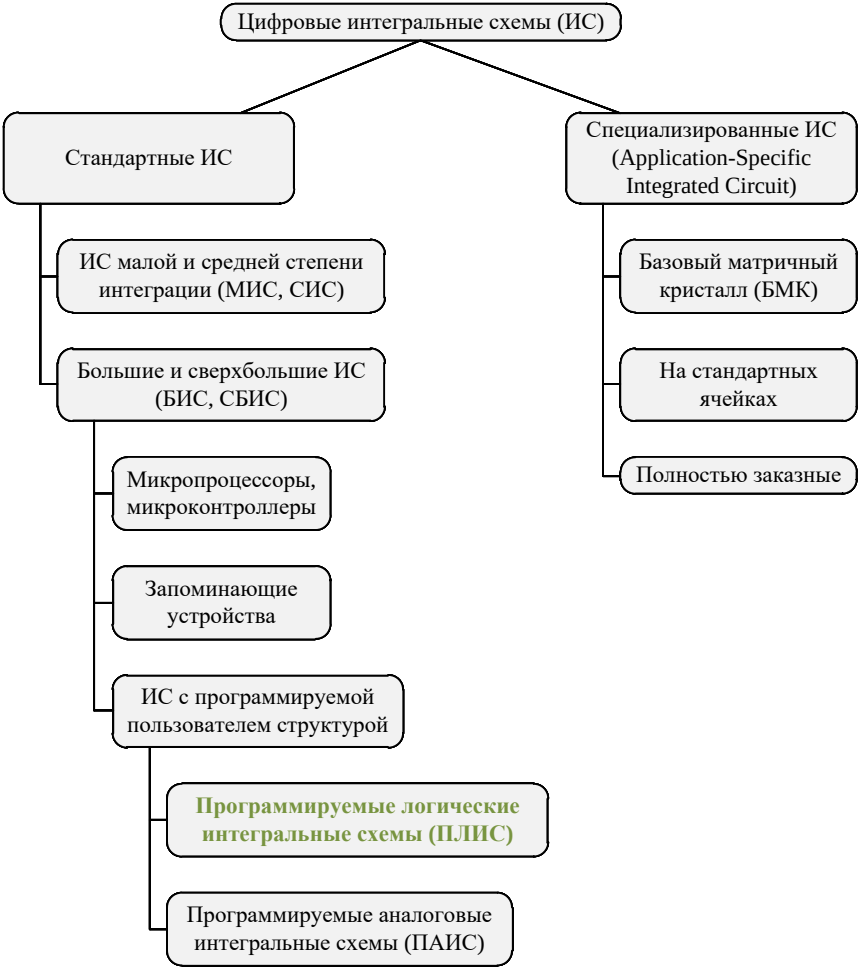


ЛР8. Программирование. Индивидуальное задание

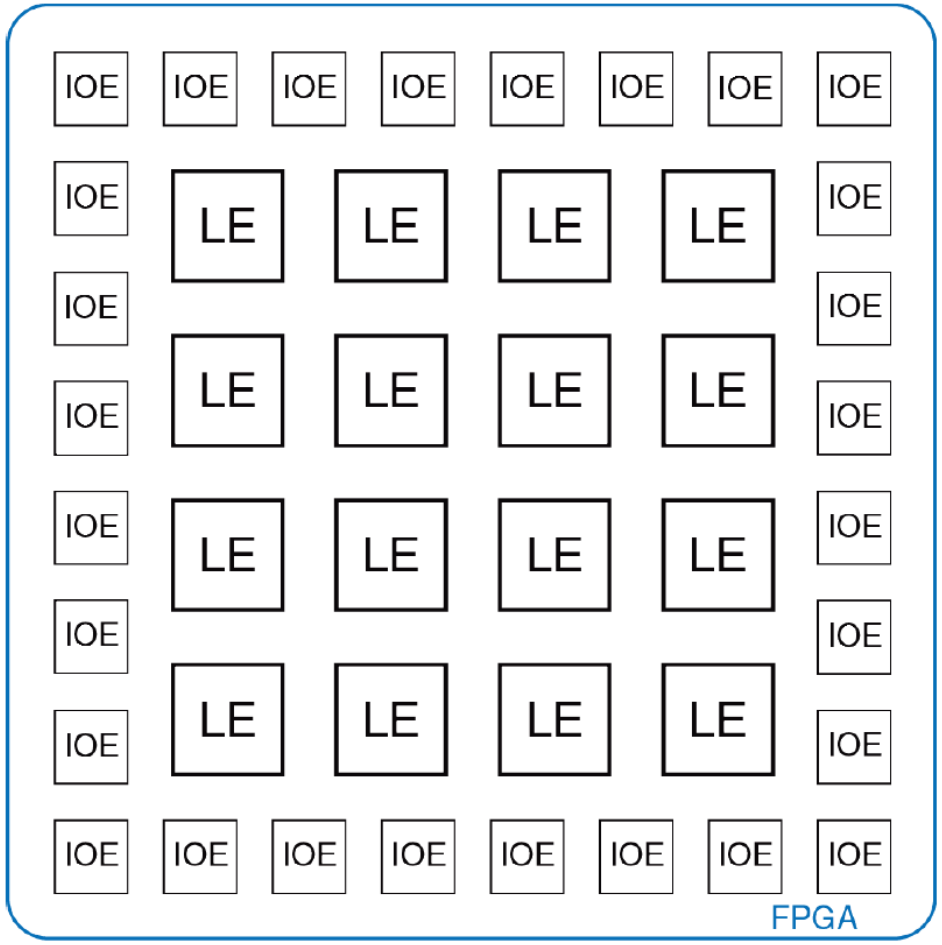


План лабораторных работ

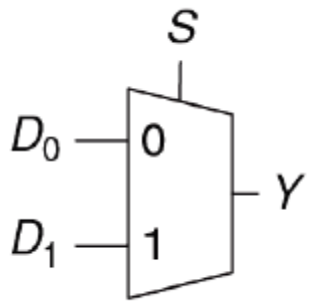
1. Арифметико-логическое устройство → **1**
2. Регистровый файл. Память. Программируемое устройство → **2**
3. Устройство управления RISC-V → **3**
4. Тракт данных RISC-V → **4**
5. Подсистема прерывания → **5**
6. Память и шина → **6**
7. Ввод/вывод. Периферия → **7**
8. Программирование (*C*). Индивидуальное задание → **8**

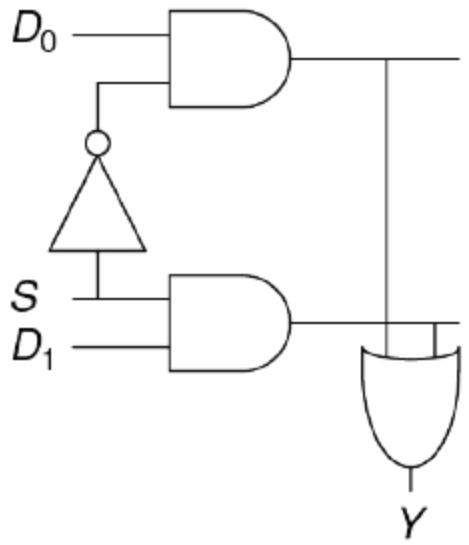

Цифровые интегральные схемы (ИС)
Стандартные ИС
Специализированные ИС (Application-Specific Integrated Circuit)
ИС малой и средней степени интеграции (МИС, СИС)
Большие и сверхбольшие ИС (БИС, СБИС)
Микропроцессоры, микроконтроллеры
Запоминающие устройства
ИС с программируемой пользователем структурой
Программируемые логические интегральные схемы (ПЛИС)
Программируемые аналоговые интегральные схемы (ПАИС)
Базовый матричный кристалл (БМК)
На стандартных ячейках
Полностью заказные

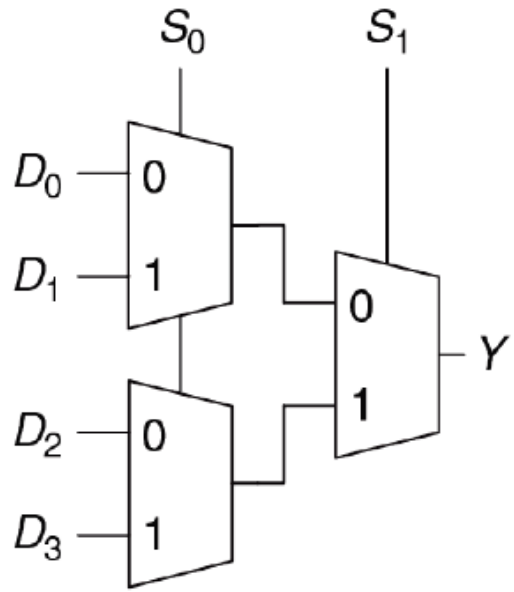
FPGA



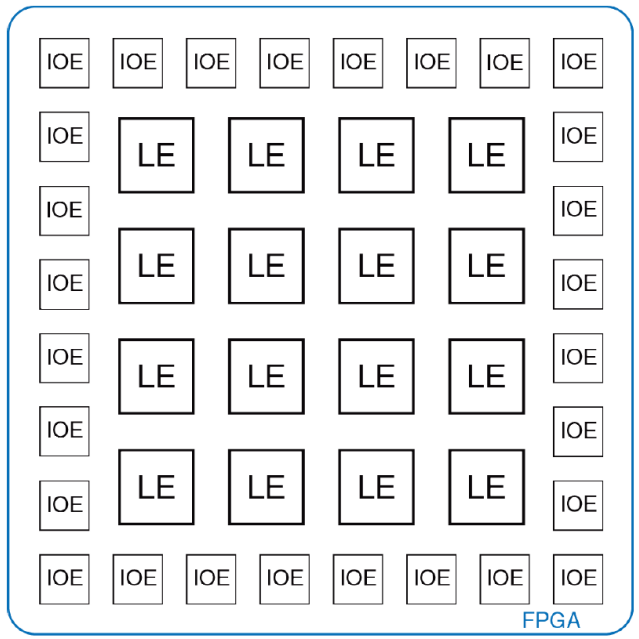
Мультиплексор

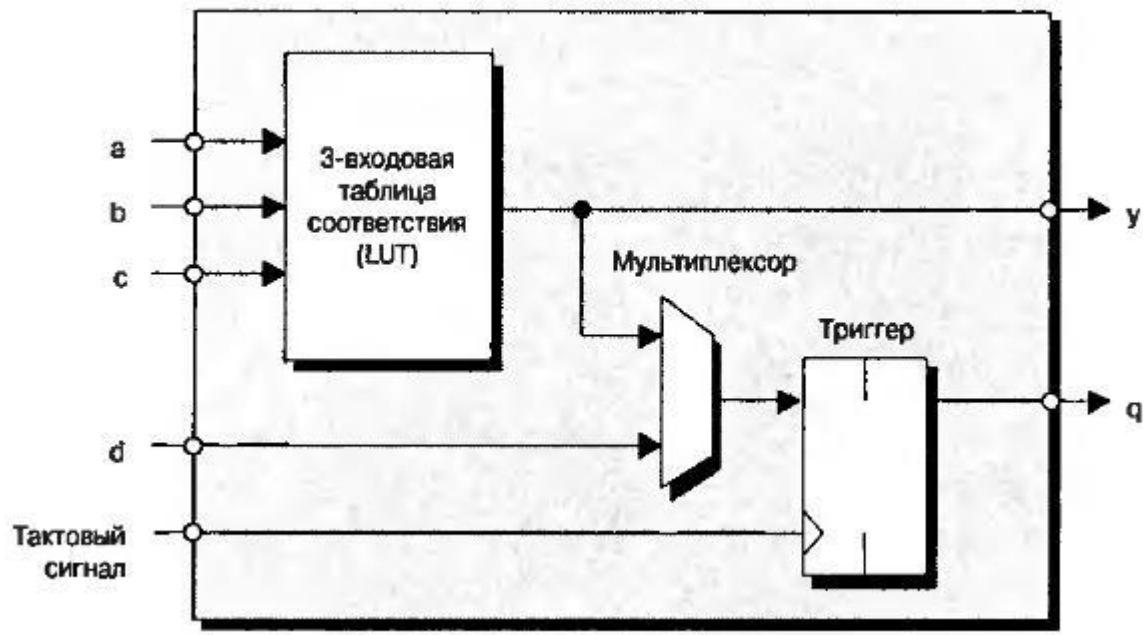




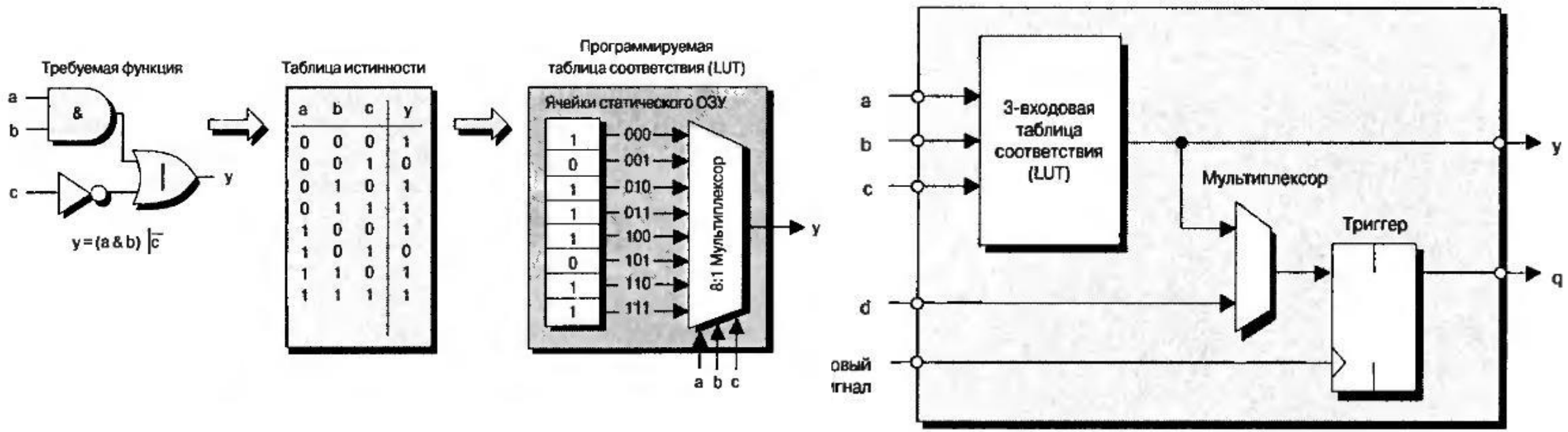


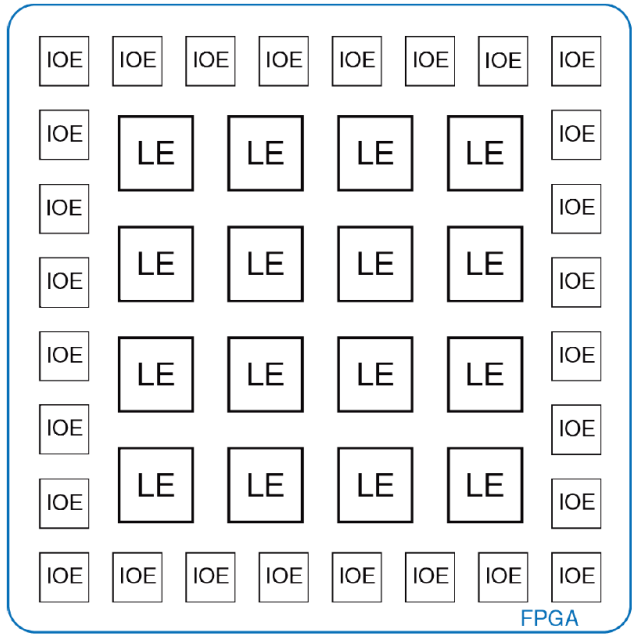
LE

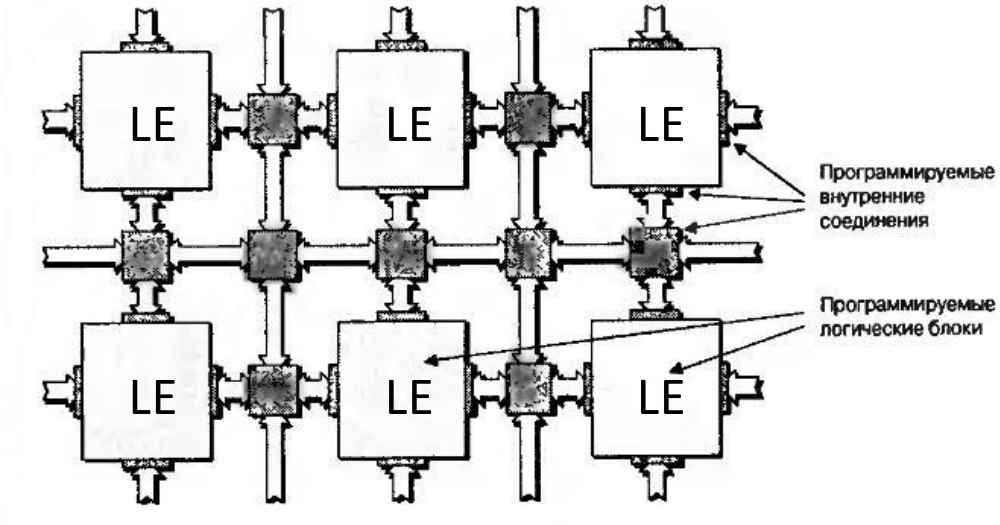

IOE
LE
FPGA

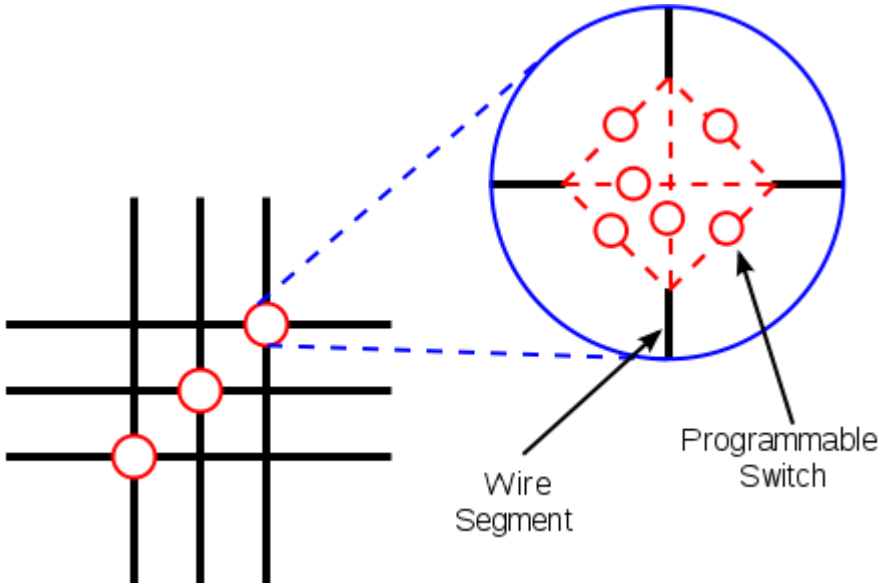

a
b
c
d
Тактовый сигнал
3-входовая таблица соответствия (LUT)
Мультиплексор
Триггер
y
q

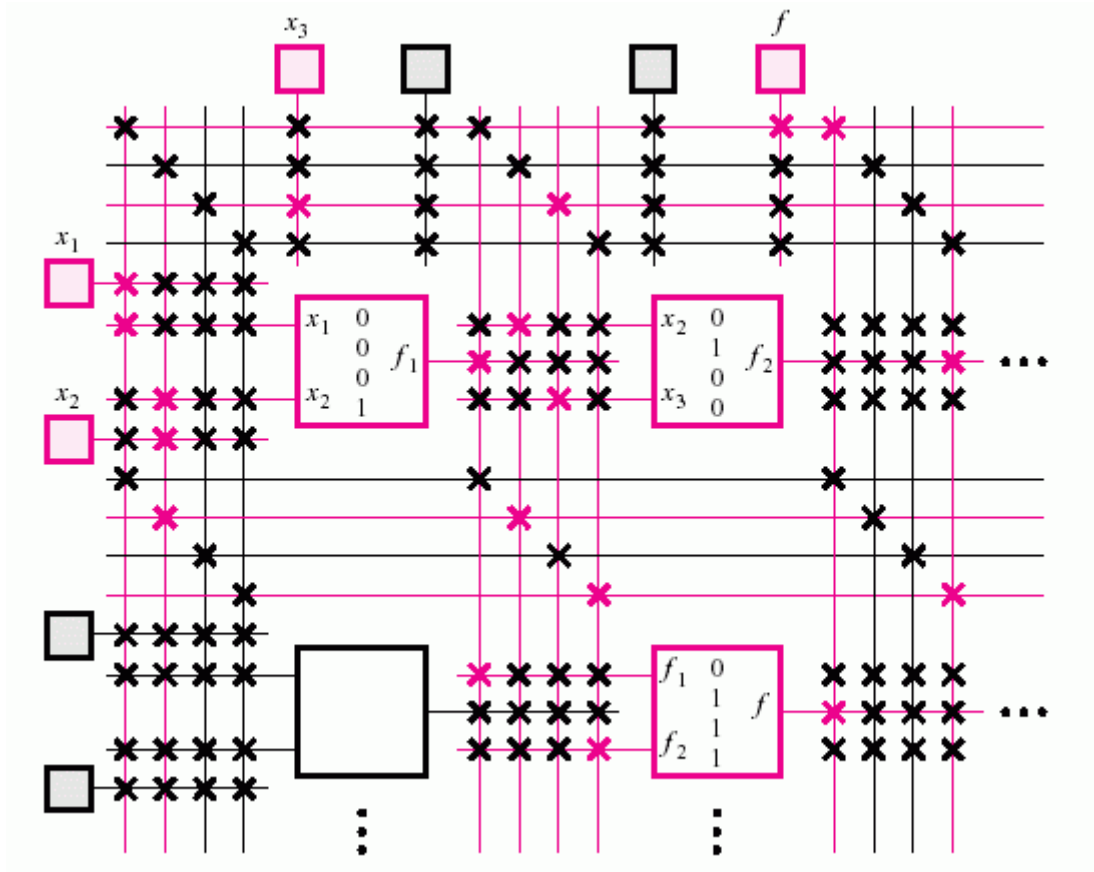
LE

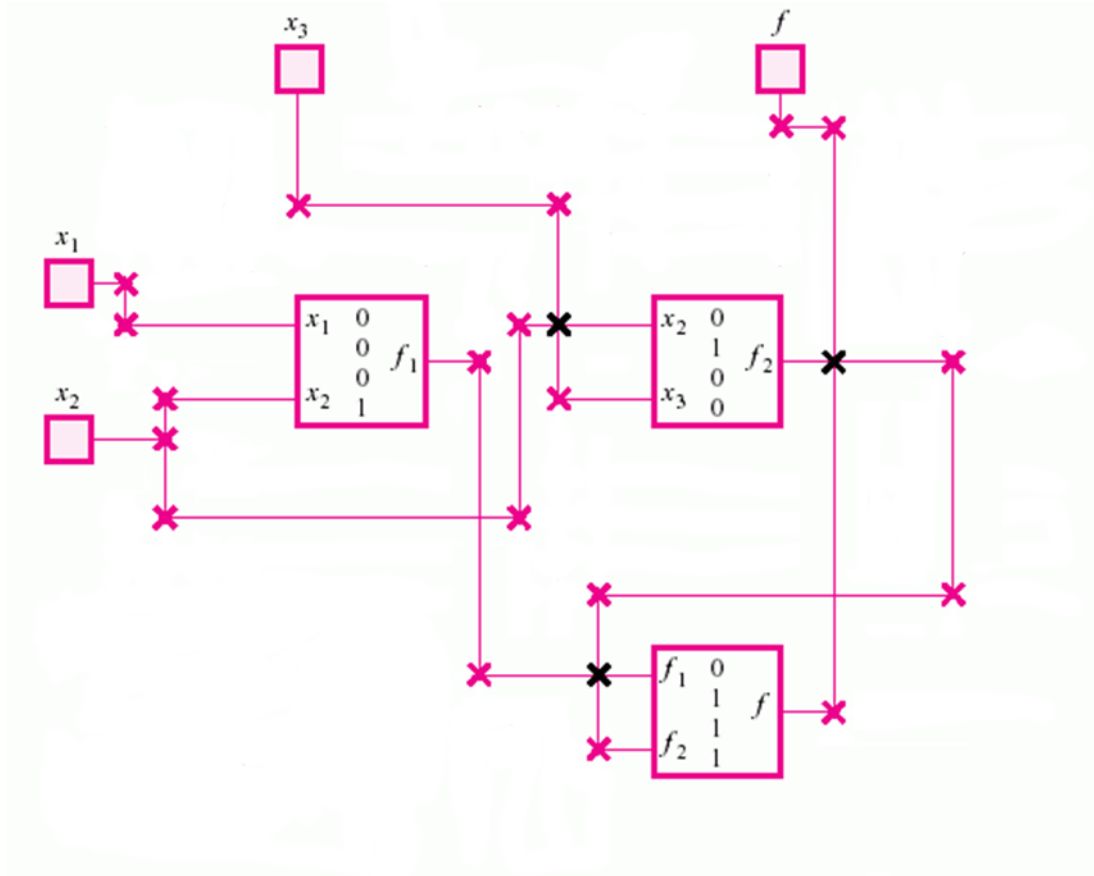

Требуемая функция
a
b
c
&
y
y = (a & b) |c|
Таблица истинности
a b c y
0 0 0 1
0 0 1 0
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 0
1 1 0 1
1 1 1 1
Программируемая таблица соответствия (LUT)
Ячейки статического ОЗУ
1
0
1
1
1
0
1
1
000
001
010
011
100
101
110
111
8:1 Мультиплексор
y
a b c
a
b
c
d
овый
гнал
3-входовая таблица соответствия (LUT)
Мультиплексор
Триггер
y
q


IOE
LE
FPGA

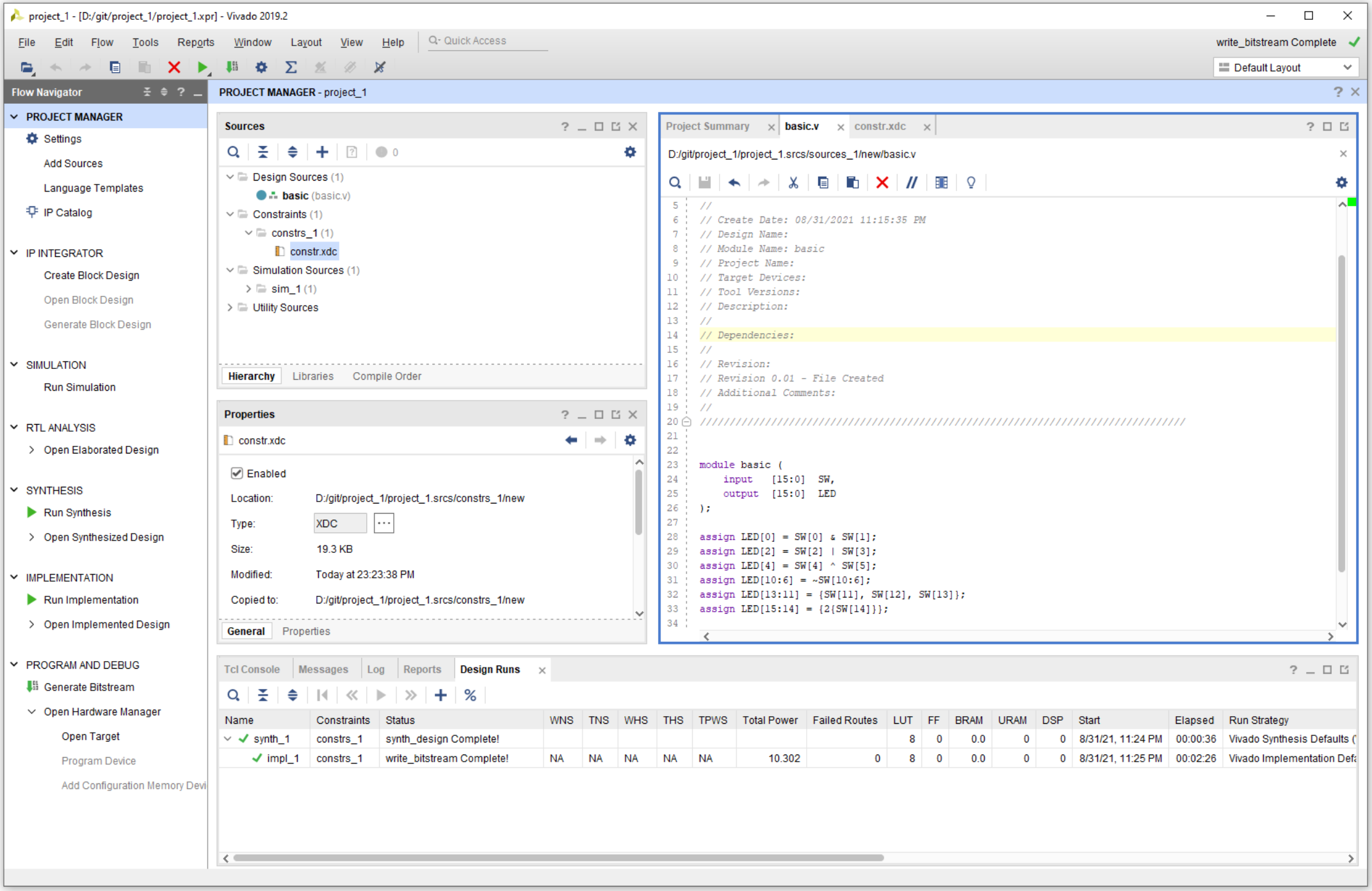

LE
Программируемые внутренние соединения
Программируемые логические блоки


Wire Segment
Programmable Switch

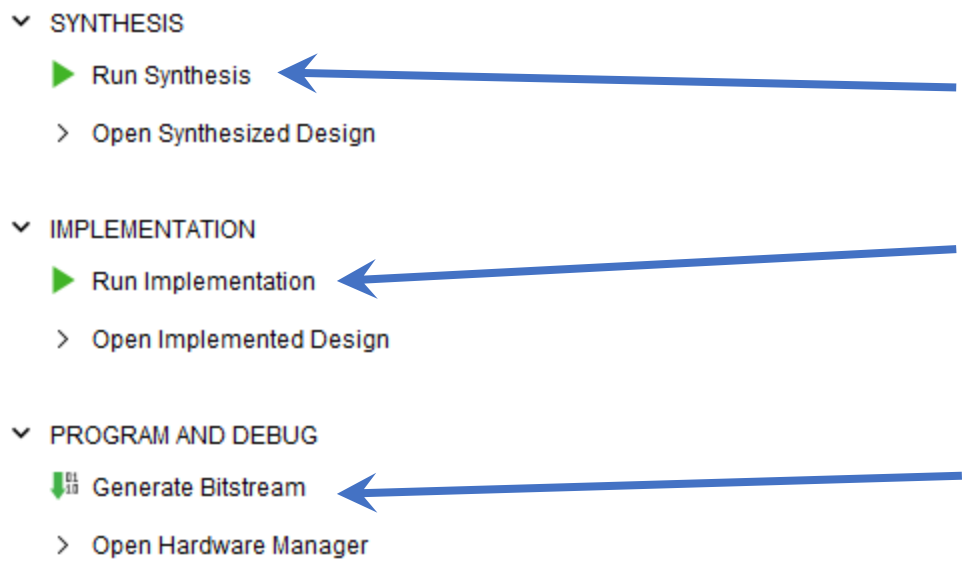

x_3
f
x_1
x_2
x_1 0 0 0 x_2 1 f_1
x_2 0 1 0 x_3 0 f_2
f_1 0 1 1 f_2 1 f


x_3
f
x_1
x_2
x_1 0 0 0 x_2 1 f_1
x_2 0 1 0 x_3 0 f_2
f_1 0 1 1 f_2 1 f


VIVADO
HLx Editions
2019.2
XILINX
Copyright 1986-2019 Xilinx, Inc.
All Rights Reserved.


project_1 - [D:/git/project_1/project_1.xpr] - Vivado 2019.2
File Edit Flow Tools Reports Window Layout View Help
Quick Access
write_bitstream Complete
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
Open Target
Program Device
Add Configuration Memory Devi
PROJECT MANAGER - project_1
Sources
Design Sources (1)
basic (basic.v)
Constraints (1)
constrs_1 (1)
constr.xdc
Simulation Sources (1)
sim_1 (1)
Utility Sources
Hierarchy Libraries Compile Order
Properties
constr.xdc
Enabled
Location: D:/git/project_1/project_1.srcs/constrs_1/new
Type: XDC
Size: 19.3 KB
Modified: Today at 23:23:38 PM
Copied to: D:/git/project_1/project_1.srcs/constrs_1/new
General Properties
Project Summary
basic.v
constr.xdc
D:/git/project_1/project_1.srcs/sources_1/new/basic.v
5 //
6 // Create Date: 08/31/2021 11:15:35 PM
7 // Design Name:
8 // Module Name: basic
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //
21
22
23 module basic (
24 input [15:0] SW,
25 output [15:0] LED
26);
27
28 assign LED[0] = SW[0] & SW[1];
29 assign LED[2] = SW[2] | SW[3];
30 assign LED[4] = SW[4] ^ SW[5];
31 assign LED[10:6] = ~SW[10:6];
32 assign LED[13:11] = {SW[11], SW[12], SW[13]};
33 assign LED[15:14] = {2{SW[14]}};
34
Tcl Console Messages Log Reports Design Runs
Name Constraints Status WNS TNS WHS THS TPWS Total Power Failed Routes LUT FF BRAM URAM DSP Start Elapsed Run Strategy
synth_1 constrs_1 synth_design Complete! 8 0 0.0 0 0 8/31/21, 11:24 PM 00:00:36 Vivado Synthesis Defaults (
impl_1 constrs_1 write_bitstream Complete! NA NA NA NA NA 10.302 0 8 0 0.0 0 0 8/31/21, 11:25 PM 00:02:26 Vivado Implementation Defa

Процесс компиляции



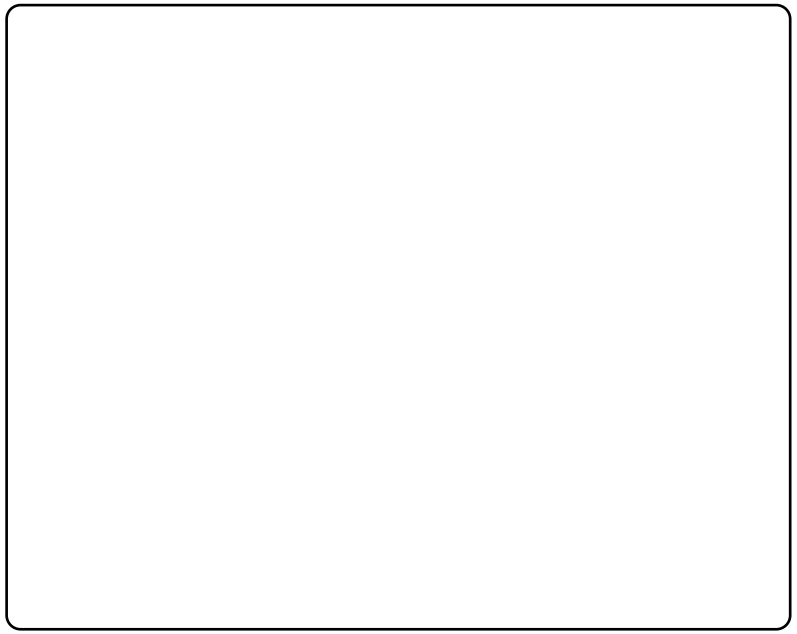
Синтез и верификация Verilog кода

Имплементация (размещение на кристалле)

Bitstream (генерация прошивки)

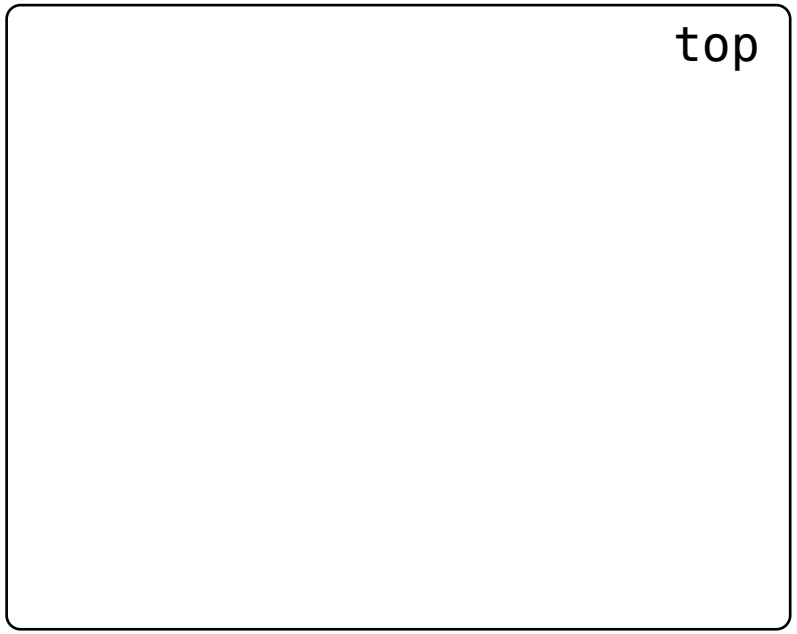
Verilog HDL

```
module
```

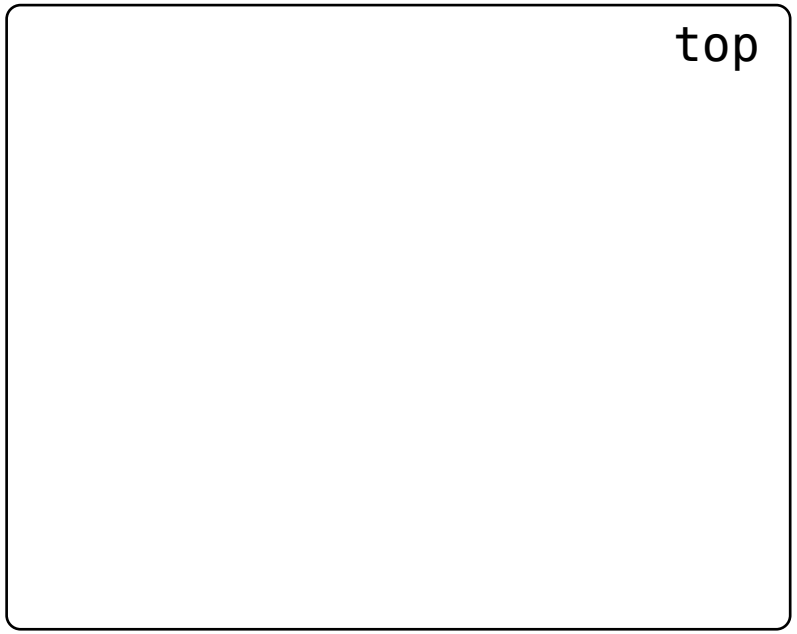


```
endmodule
```

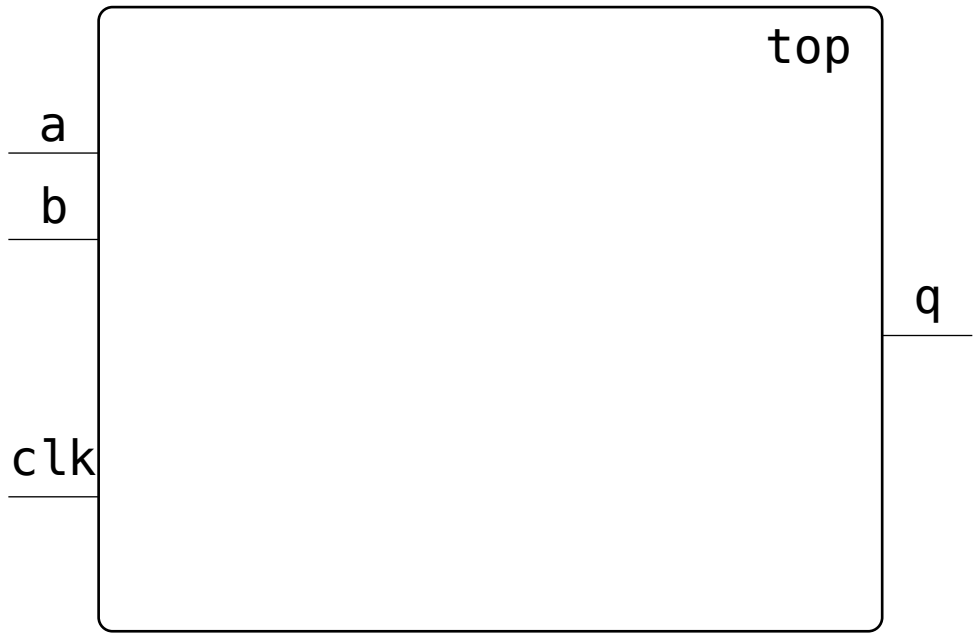
```
module top

endmodule
```



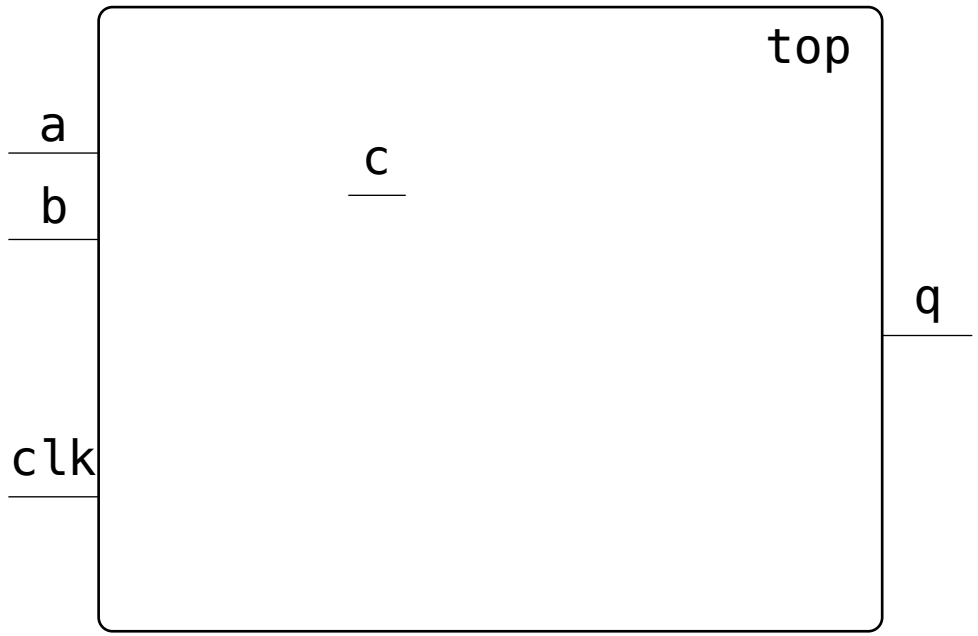
```
module top ();

endmodule
```



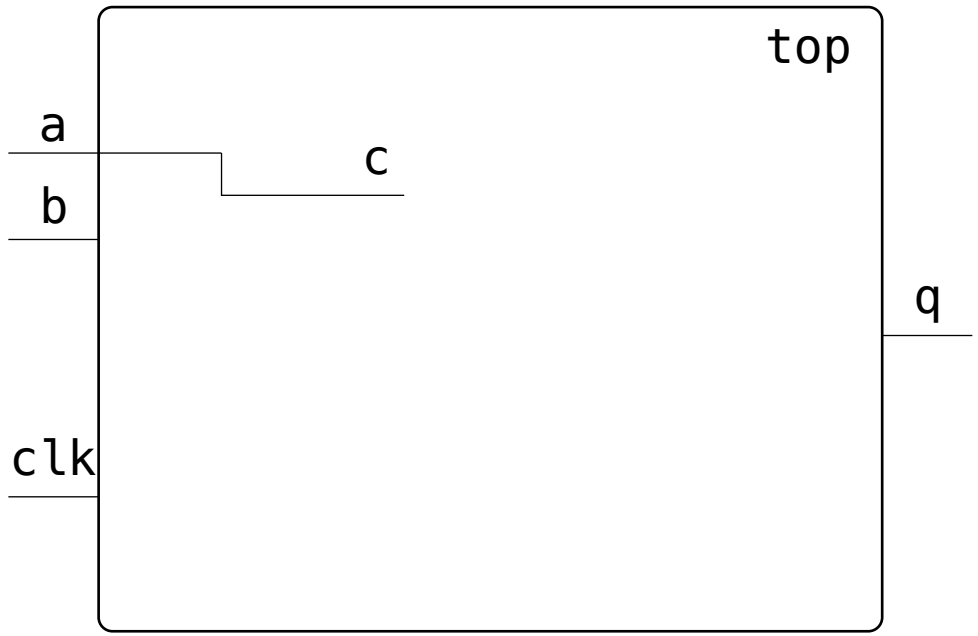
```
module top (
        input           a,
        input           b,
        input           clk,
        output          q
);


endmodule
```



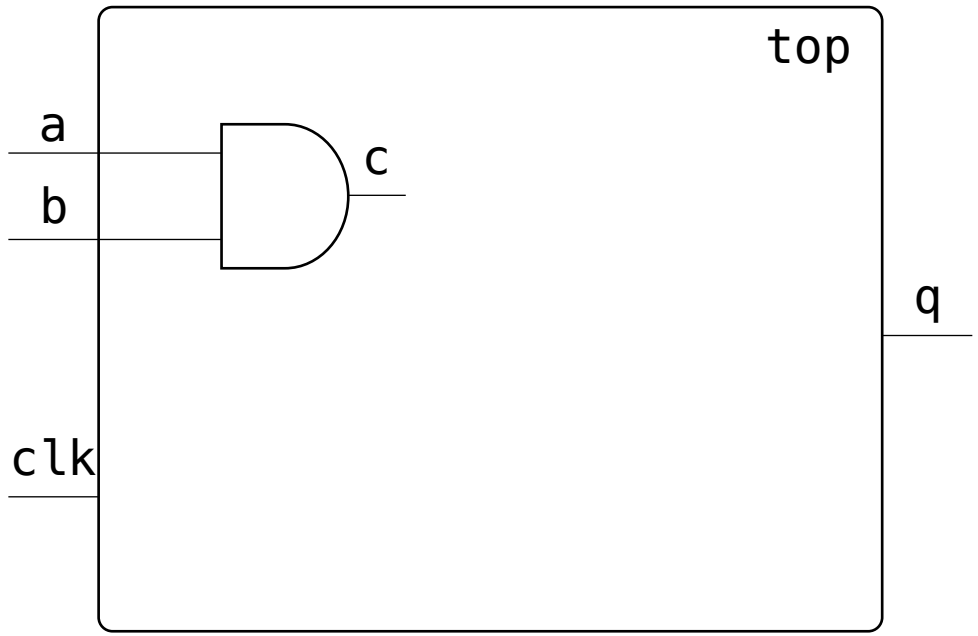
```
module top (
        input           a,
        input           b,
        input           clk,
        output          q
);

wire c;



endmodule
```



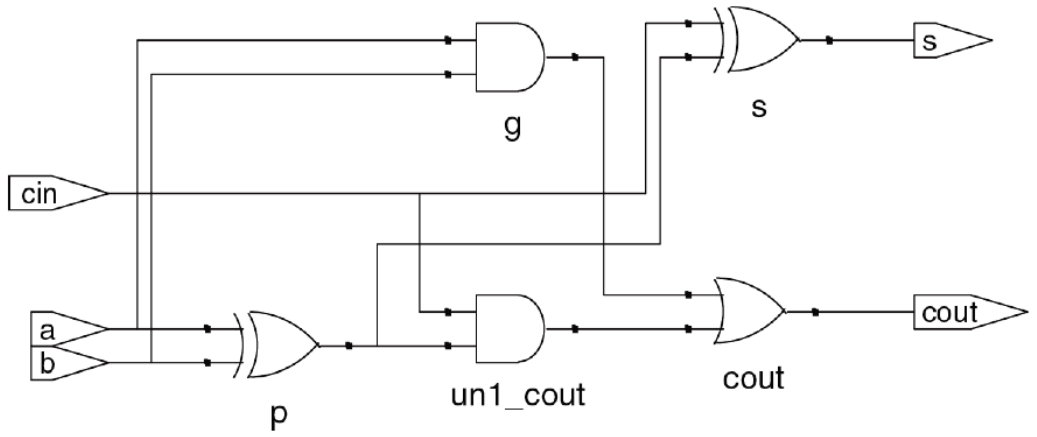
```
module top (
        input           a,
        input           b,
        input           clk,
        output          q
);

wire c;

assign c = a;



endmodule
```



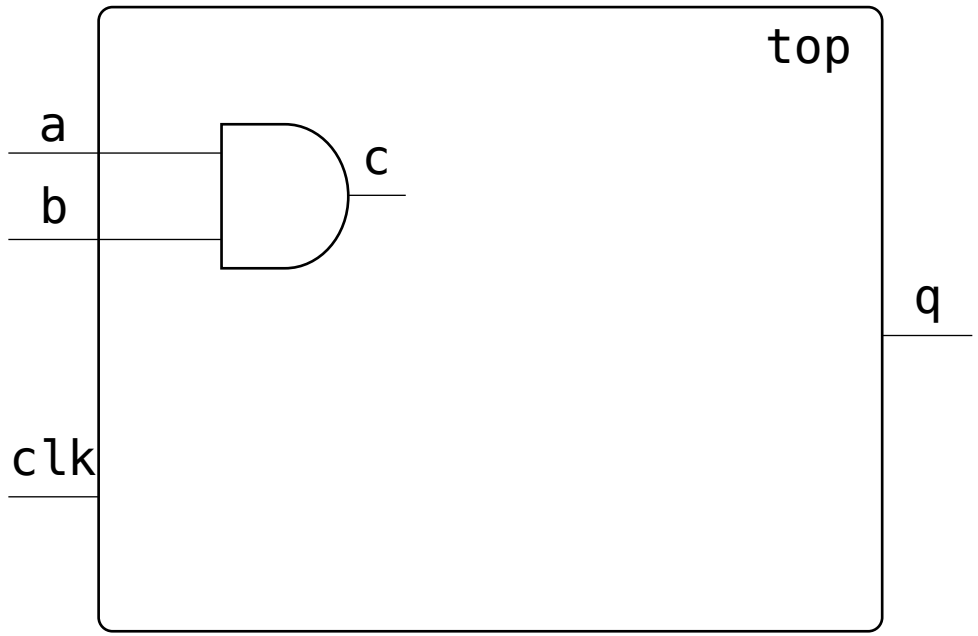
```
module top (
        input           a,
        input           b,
        input           clk,
        output          q
);

wire c;

assign c = a & b;



endmodule
```



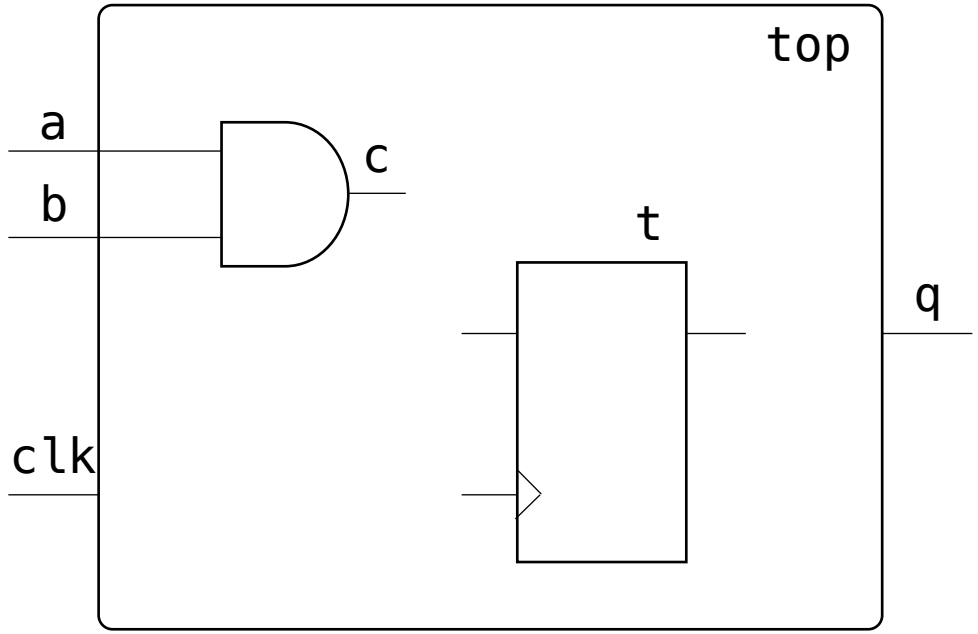
```
module fulladder (a, b, cin, s, cout);

       input  a, b, cin;
       output s, cout;

wire p, g;

assign p = a ^ b;
assign g = a & b;
assign s = p ^ cin;
assign cout = g |(p & cin);

endmodule
```



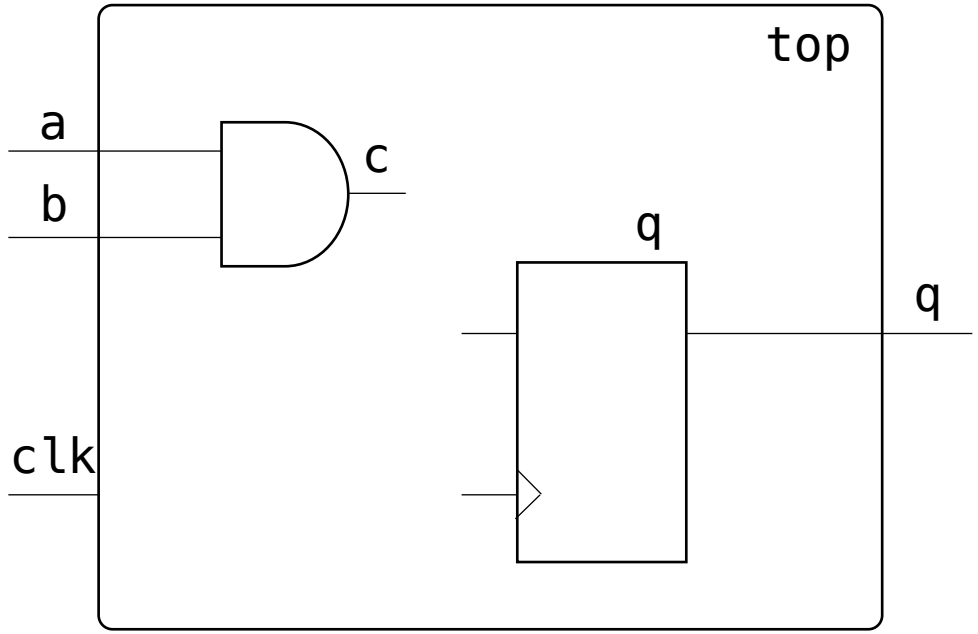
```
module top (
        input           a,
        input           b,
        input           clk,
        output          q
);

wire c;

assign c = a & b;



endmodule
```



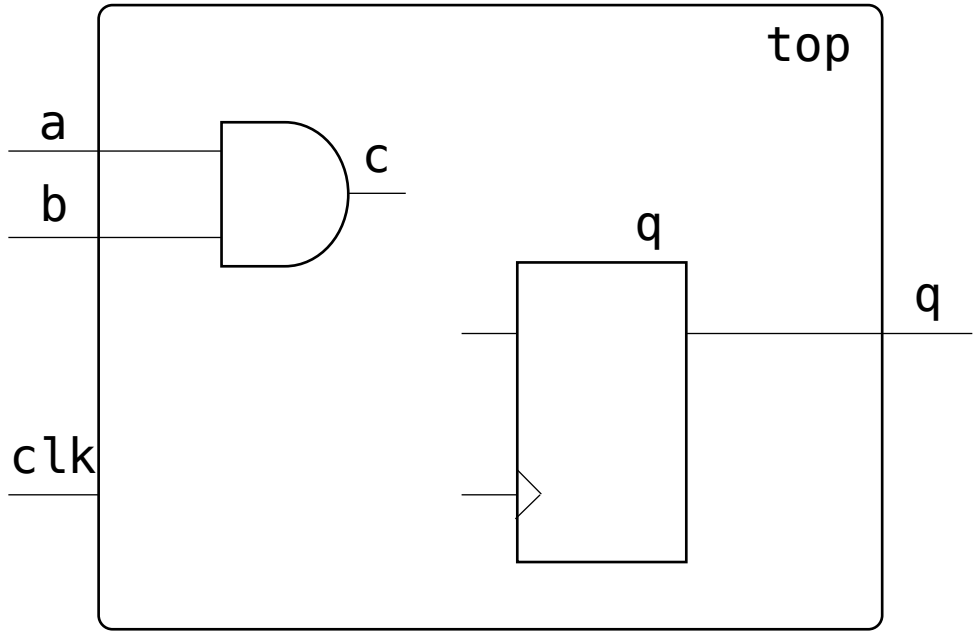
```
module top (
       input         a,
       input         b,
       input         clk,
       output        q
);

wire c;
reg  t;
assign c = a & b;



endmodule
```



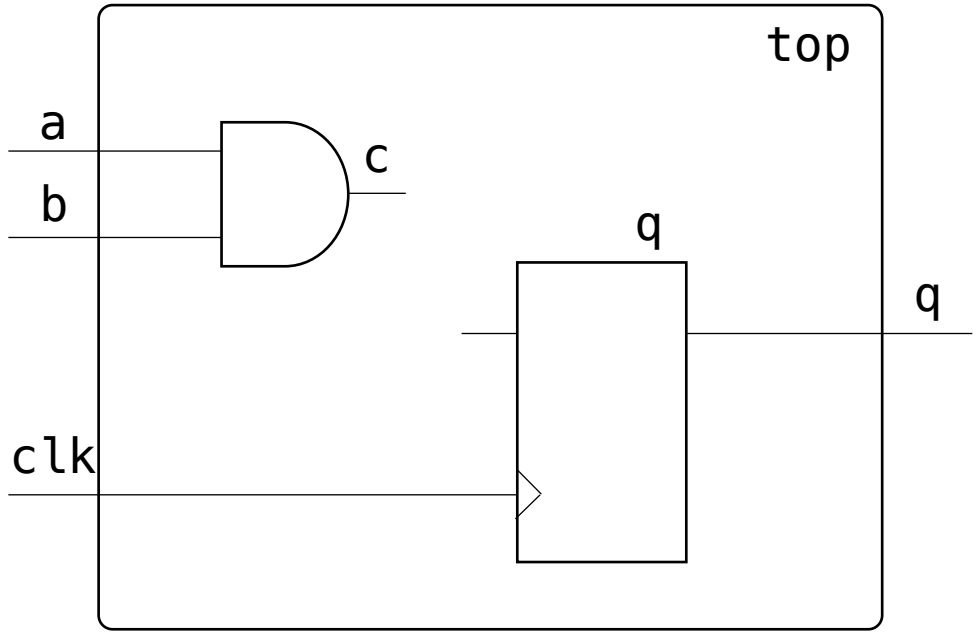
```
module top (
        input           a,
        input           b,
        input           clk,
        output  reg     q
);

wire c;

assign c = a & b;



endmodule
```



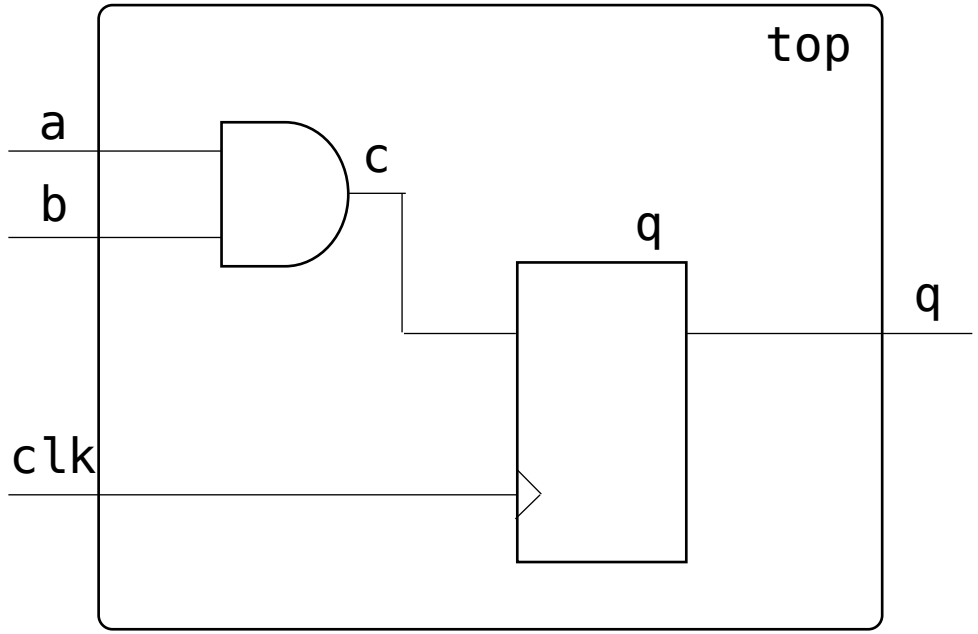
```
module top (
       input         a,
       input         b,
       input         clk,
       output  reg   q
);

wire c;

assign c = a & b;

always @ ()


endmodule
```



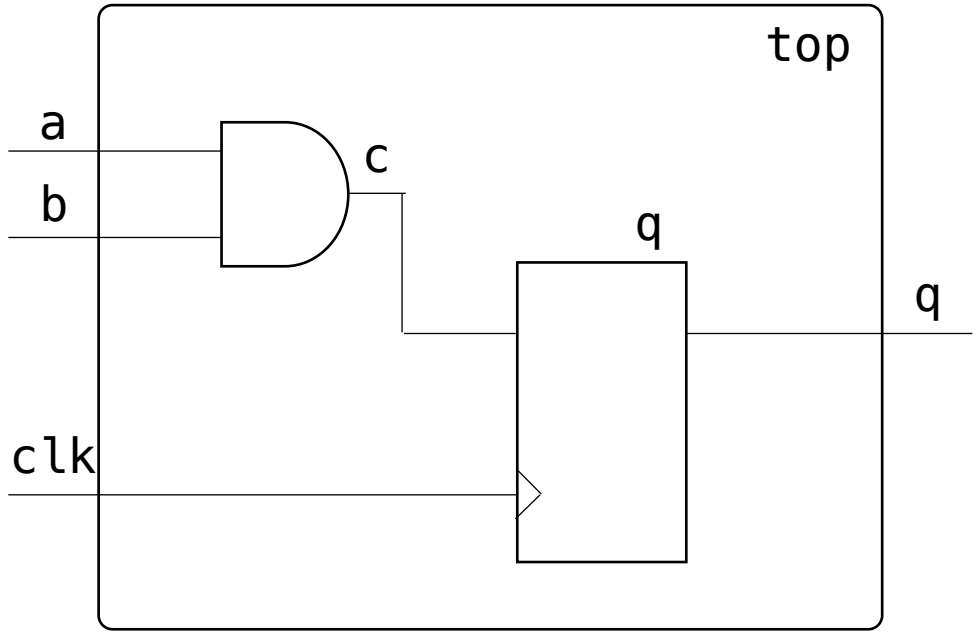
```
module top (
        input           a,
        input           b,
        input           clk,
        output  reg     q
);

wire c;

assign c = a & b;

always @ (posedge clk)


endmodule
```



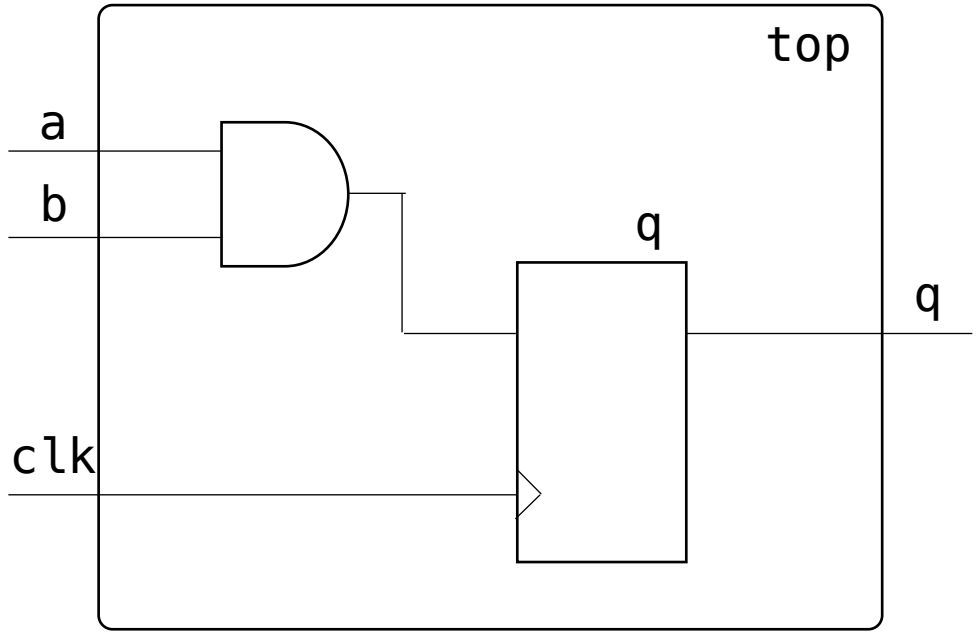
```
module top (
        input           a,
        input           b,
        input           clk,
        output  reg     q
);

wire c;

assign c = a & b;

always @ (posedge clk)
        q <= c;

endmodule
```



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

wire c;

always @ (posedge clk)
        q <= c;

assign c = a & b;

endmodule
```

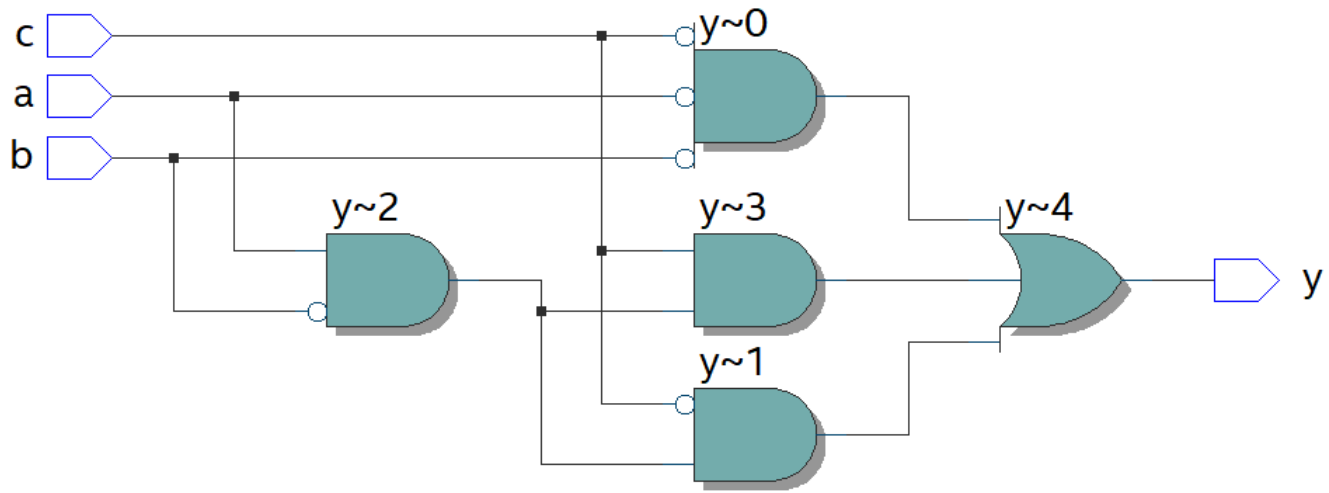


```
module top (
        input           a,
        input           b,
        input           clk,
        output  reg     q
);


always @ (posedge clk)
        q <= a & b;


endmodule
```



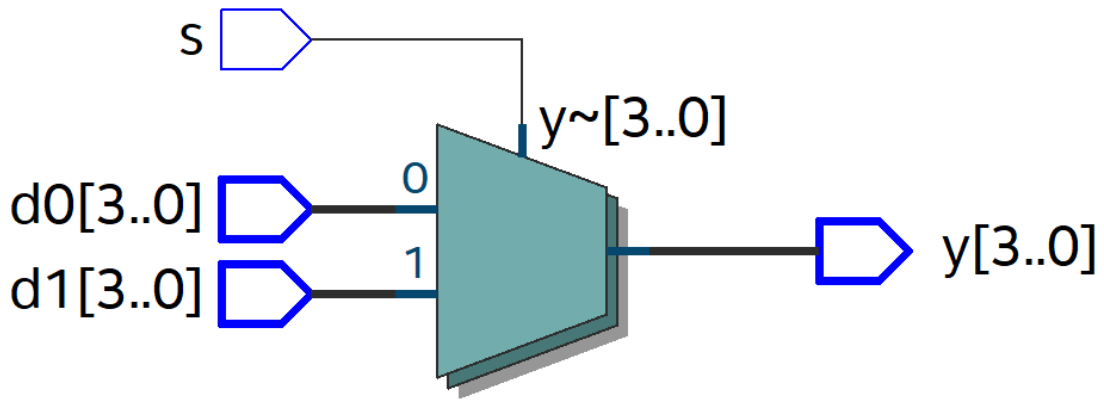
Комбинационная логика

```
module dut (
        input  a, b, c,
        output y
);

assign y = a & ~b & ~c | a & ~b & c | a & ~b & c;

endmodule
```



Тернарный оператор

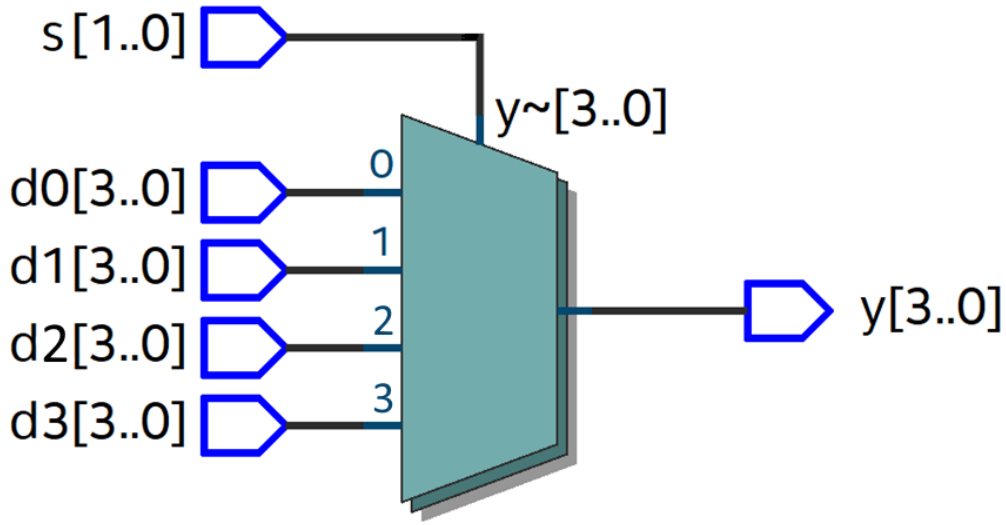
```
module ternarny (
        input   [3:0]   d0, d1,
        input           s,
        output  [3:0]   y
);

assign y = s ? d1 : d0;

endmodule
```



`? :` данный оператор называется тернарным, т.к. в нем используется **3** входных значения: `s`, `d1`, `d0`.

CASE

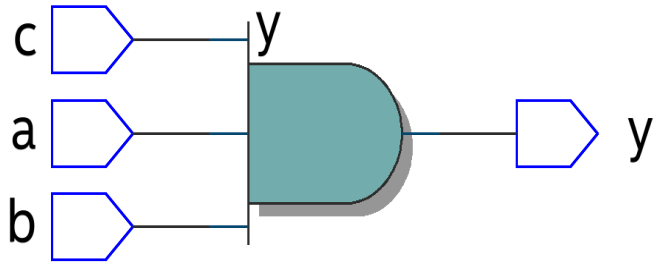
```
module mux (
        input       [3:0]   d0, d1, d2, d3,
        input       [1:0]   s,
        output reg  [3:0]   y
);

always @ (*) begin
        case (s)
                2'b00:  y = d0;
                2'b01:  y = d1;
                2'b10:  y = d2;
                2'b11:  y = d3;
        endcase
end

endmodule
```

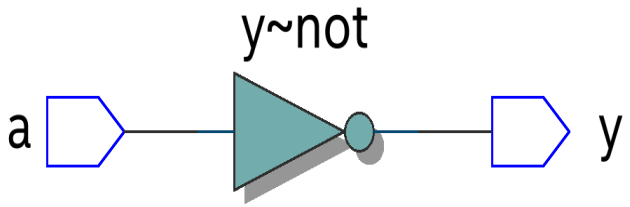


Иерархия модулей в Verilog

```
module and_3 (
       input  a, b, c,
       output y
);

assign y = a & b & c;

endmodule
```

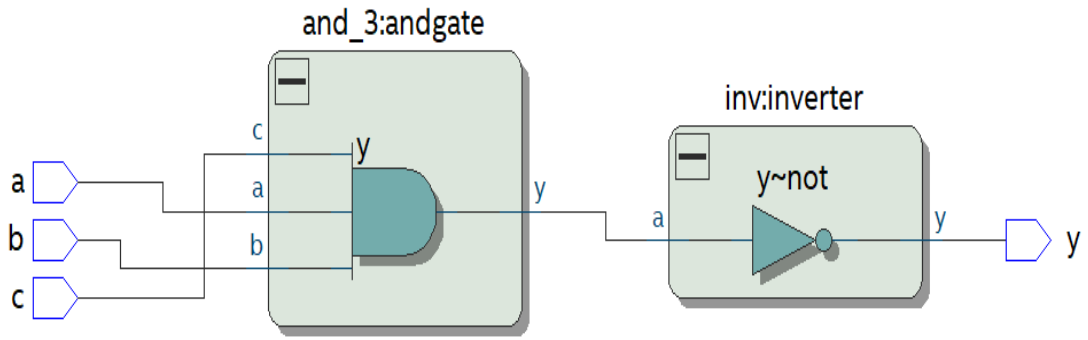
```
module inv (
       input  a
       output y
);

assign y = ~a;

endmodule
```





Иерархия модулей в Verilog

```
module dut (
        input  a, b, c,
        output y
);

wire n1;

and_3 andgate (
        .a(a),
        .b(b),
        .c(c),
        .y(n1)
);

inv inverter (
        .a(n1),
        .y(y)
);

endmodule
```



- Имя подключаемого модуля (**and_3**, **inv**)
- Название примитива. Например, нам может понадобиться **3** копии модуля **and_3**. Тогда мы сможем подключить 3 экземпляра модуля **and_3**, используя различные наименования для прототипов (andgate_1, andgate_2 …)
- Символ точка, перед наименованием порта отсылает к реальному порту подключаемого модуля (у модуля **inverter**, порты именуются **a, y**). В скобках обозначается куда будут подключаться сигналы в *top*-модуле

План лабораторной работы

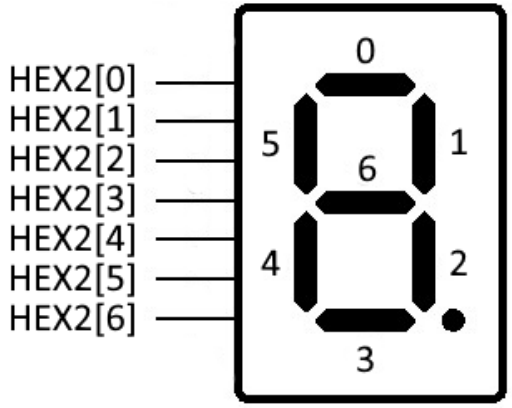
- 1 пара
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - Тренинг по Vivado и Verilog HDL (**TS**)
- 2 пара
  - Арифметико-логическое устройство (**T**)
  - Описание АЛУ на Verilog HDL (**S**)
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

План лабораторной работы

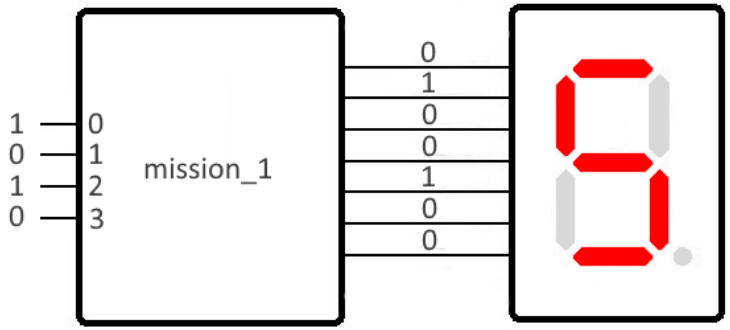
- 1 пара
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - ~~Тренинг по Vivado и Verilog HDL (**TS**)~~
  - Задание на отладочном стен (**S**)
- 2 пара
  - Арифметико-логическое устройство (**T**)
  - Описание АЛУ на Verilog HDL (**S**)
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

Задание на отладочном стенде

```
module mission_1 (
        input    [9:0] SW,
        output   [6:0] HEX2
);

// реализовать модуль управления
// семисегментными индикаторами


endmodule
```



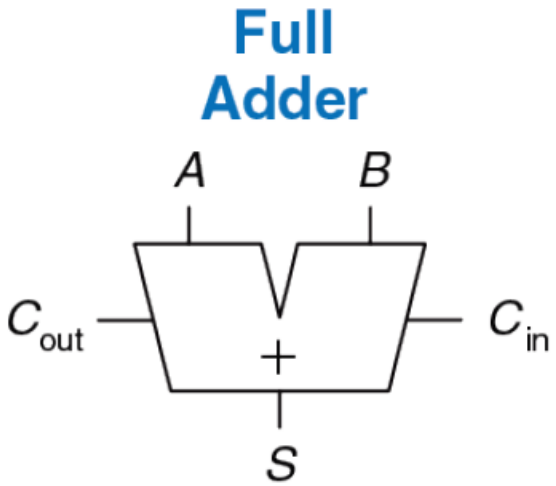
SW[3:0] = 4'b0101 ⟶ HEX2 = 7'b0010010



План лабораторной работы

- ~~1 пара~~
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - ~~Тренинг по Vivado и Verilog HDL (**TS**)~~
  - ~~Задание на отладочном стен (**S**)~~
- 2 пара
  - Арифметико-логическое устройство (**T**)
  - Описание АЛУ на Verilog HDL (**S**)
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

Сумматор

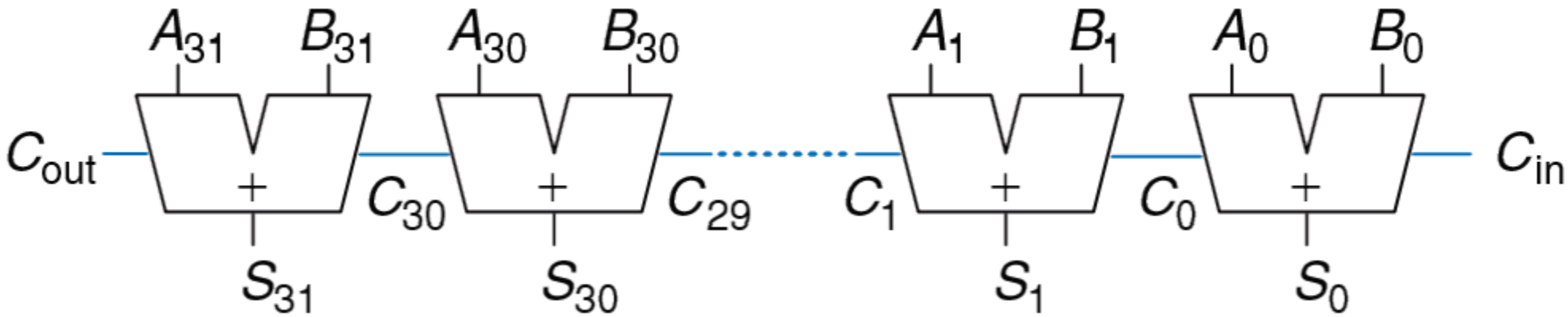


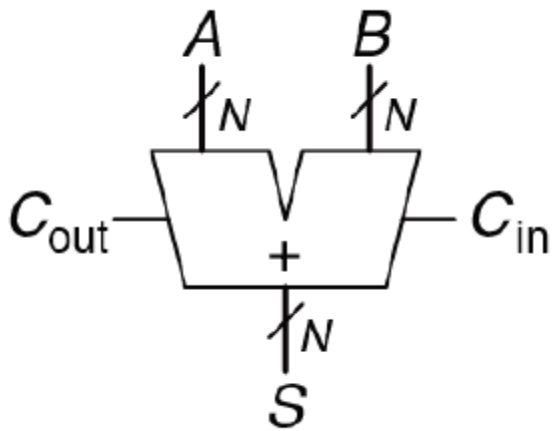
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Сумматор

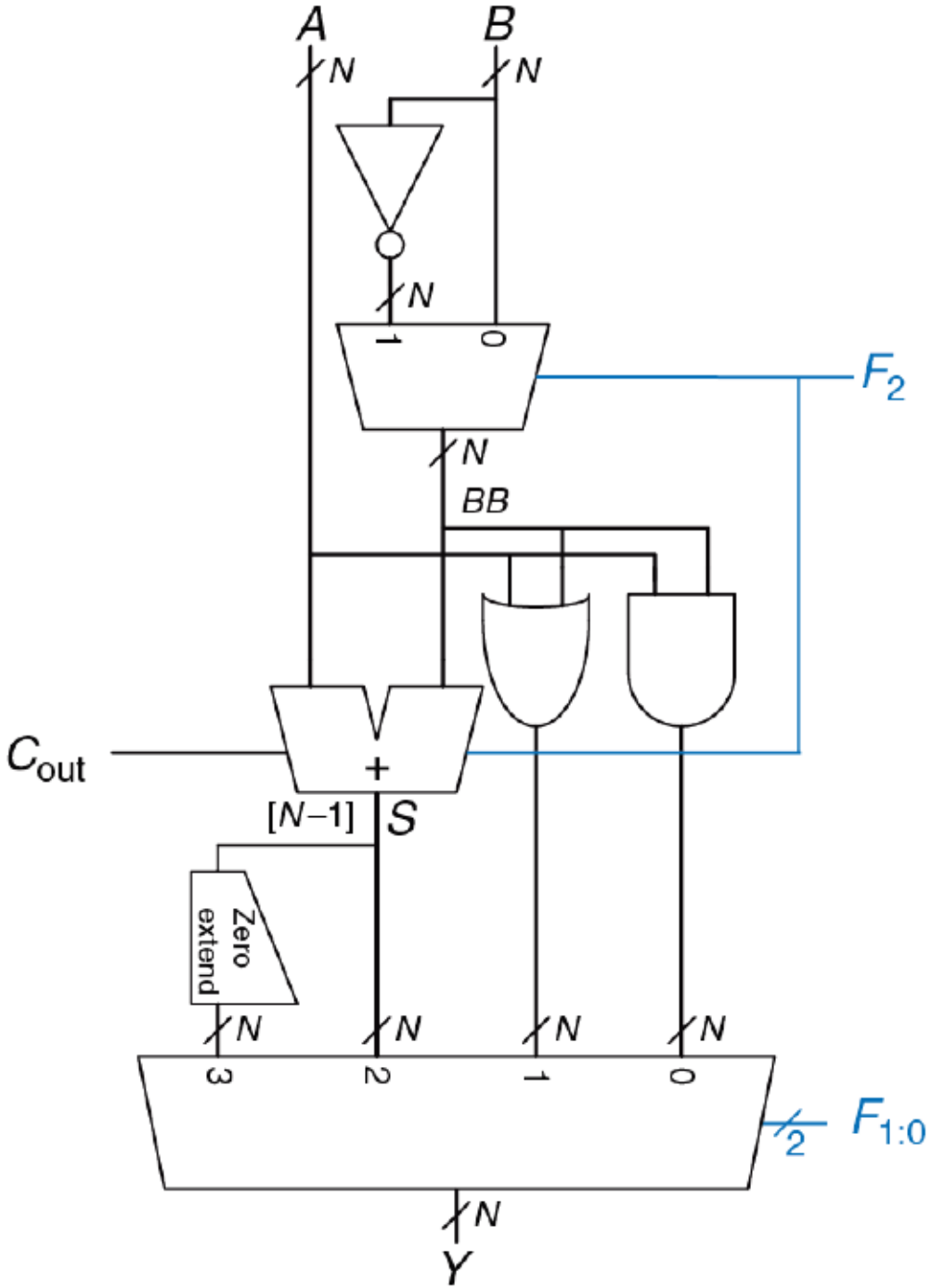

A_{31}
B_{31}
A_{30}
B_{30}
A_1
B_1
A_0
B_0
C_{out}
C_{30}
C_{29}
C_1
C_0
C_{in}
S_{31}
S_{30}
S_1
S_0


A
B
N
N
C_{out}
C_{in}
N
S

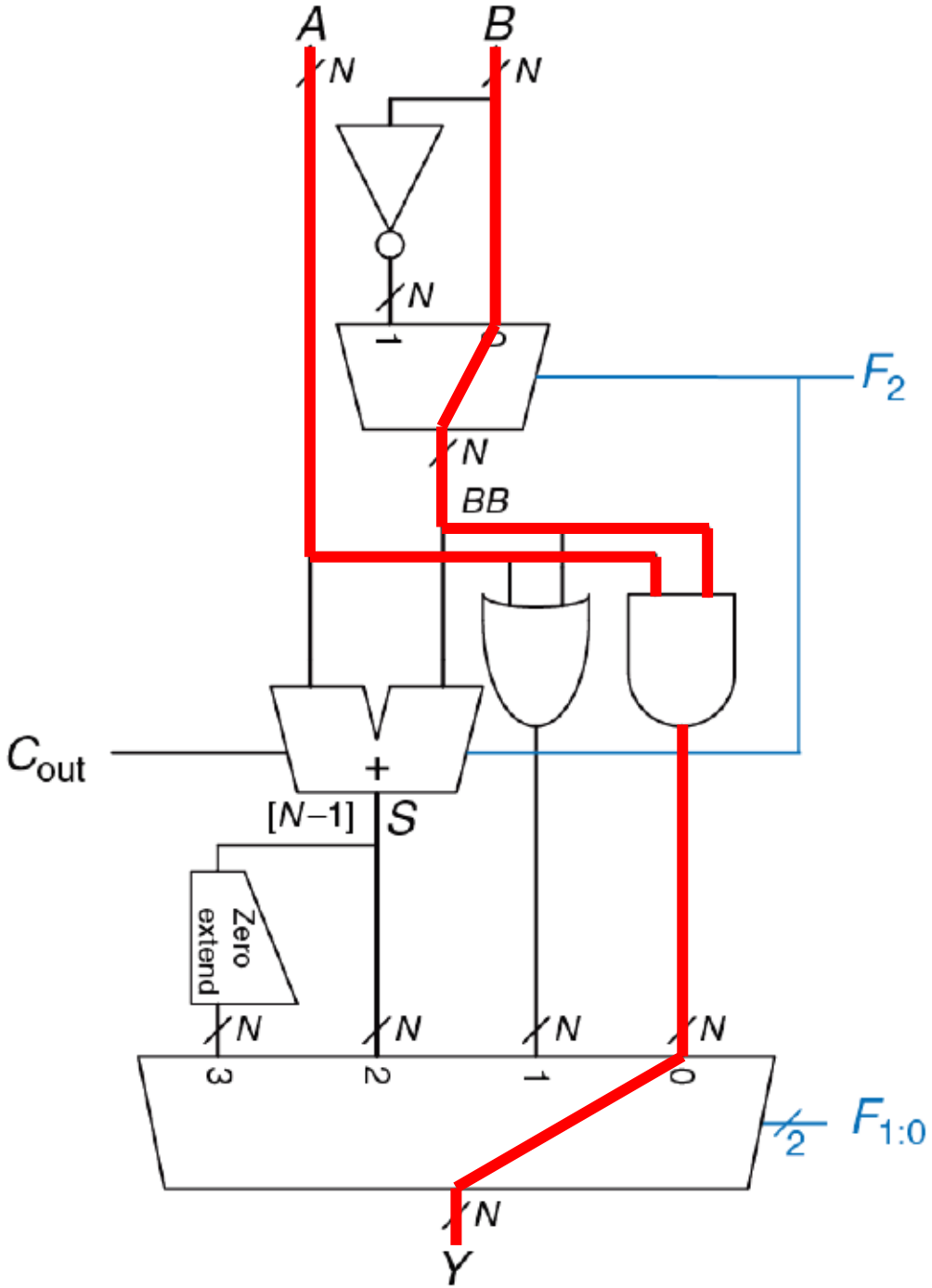
Арифметико-логическое устройство

- АЛУ – блок процессора, выполняющий арифметические и поразрядно логические операции
  - Арифметические операции имеют перенос
  - Логические операции без переноса
- АЛУ – комбинационная схема
- На вход АЛУ поступают информационные сигналы (данные, над которыми происходит операция) и управляющие сигналы (определяют, какая операция будет произведена над данными), на выходе – результат операции
- АЛУ формирует флаги результата

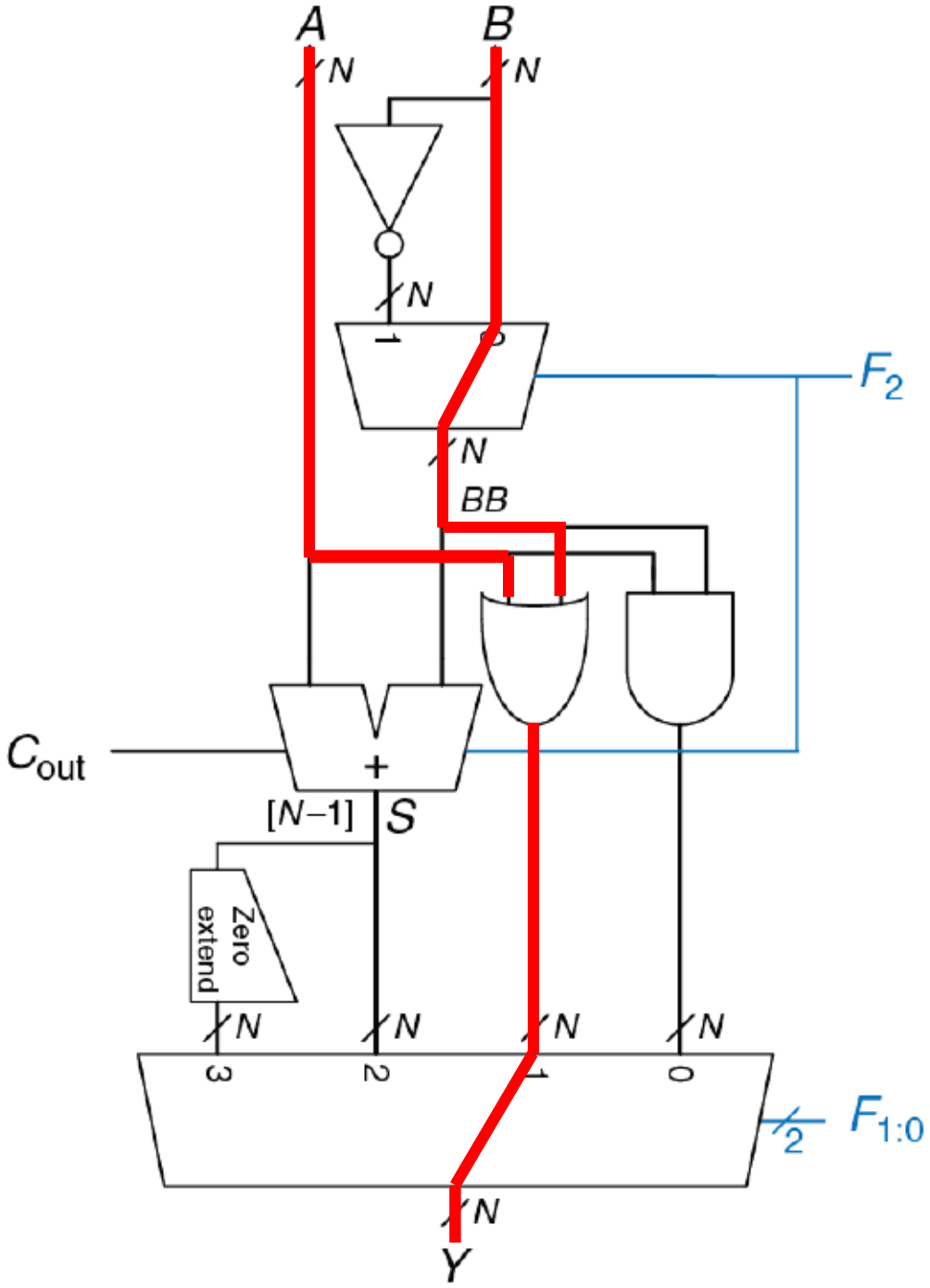
Пример АЛУ



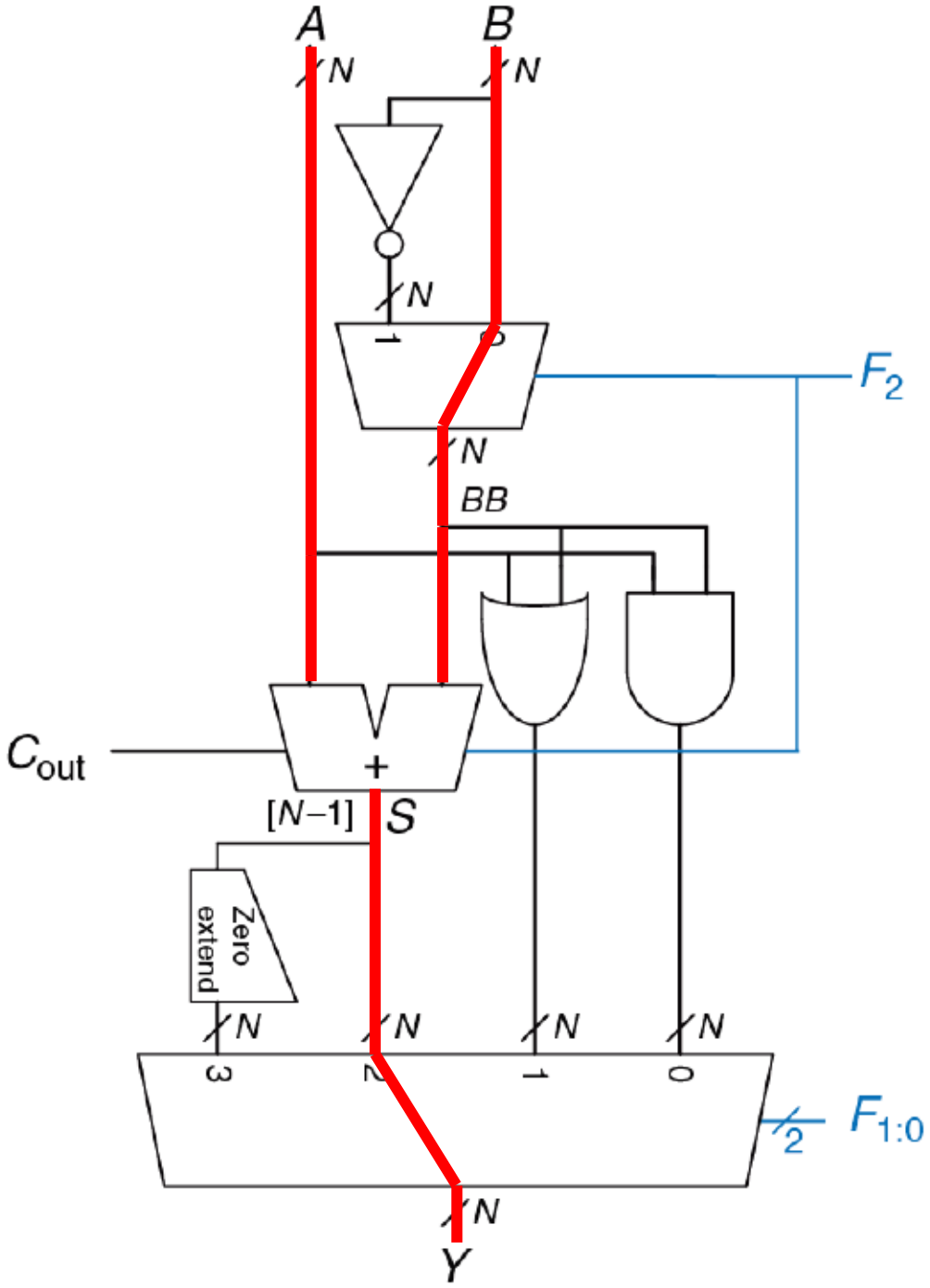
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



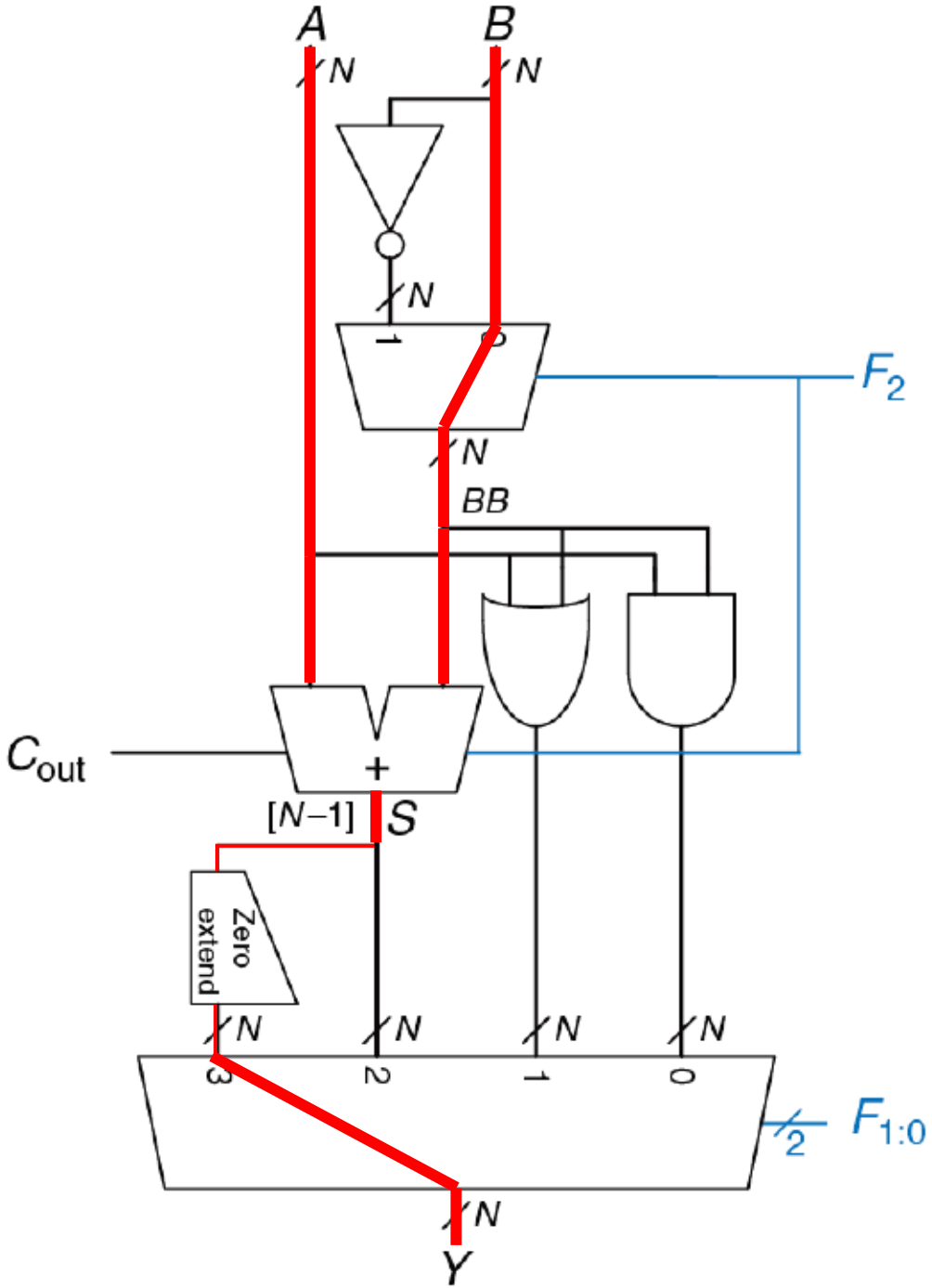
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



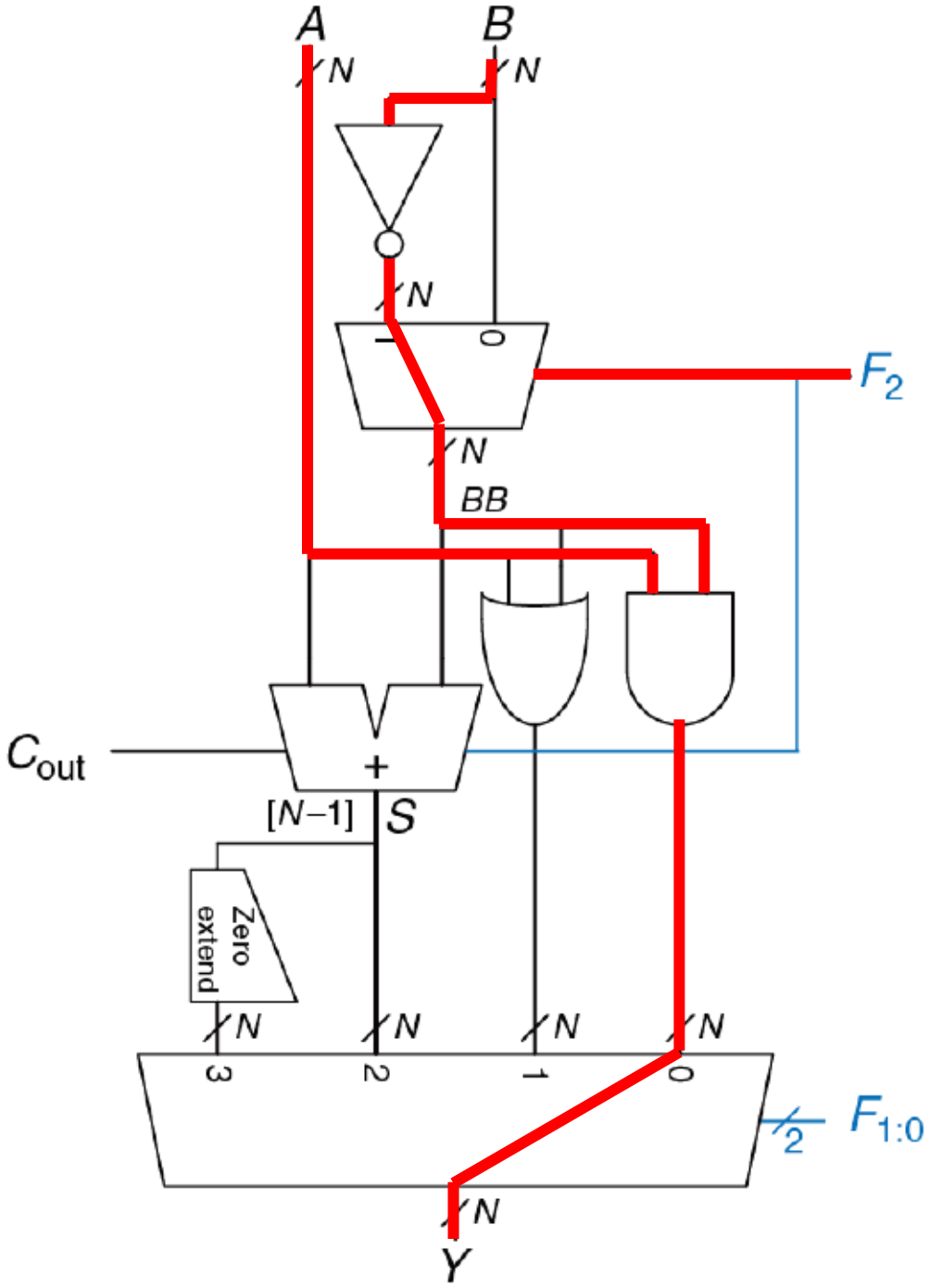
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



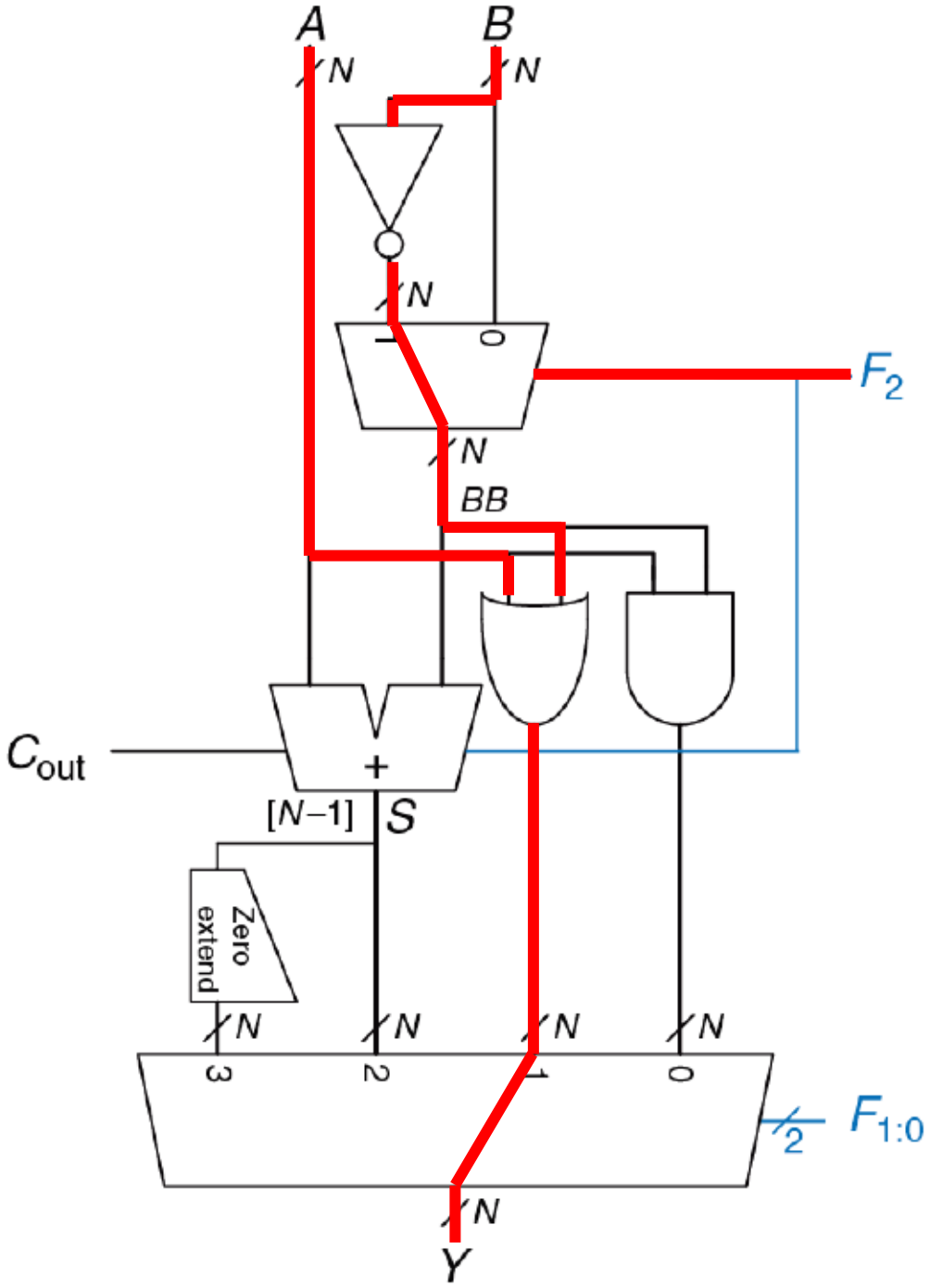
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



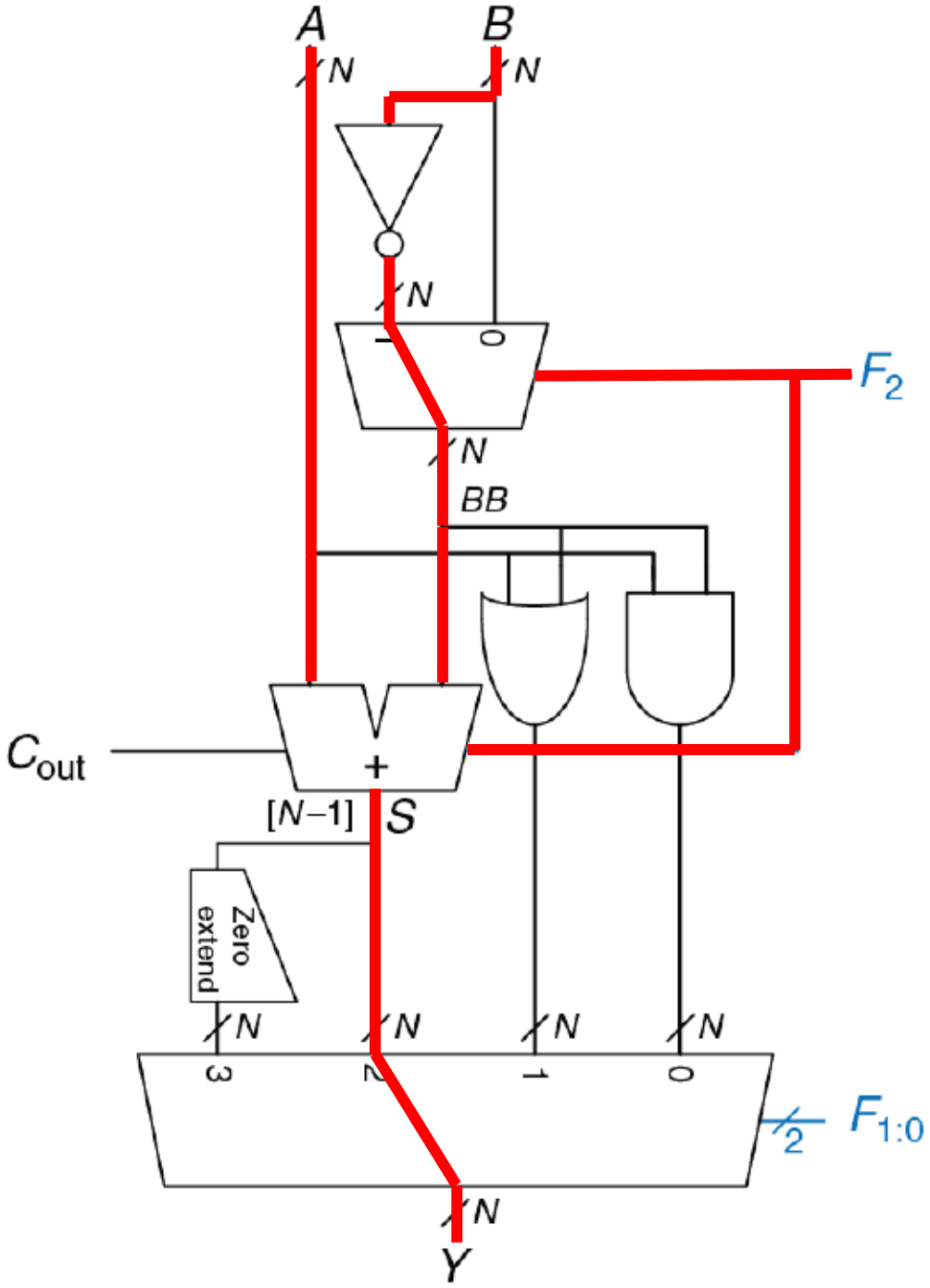
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



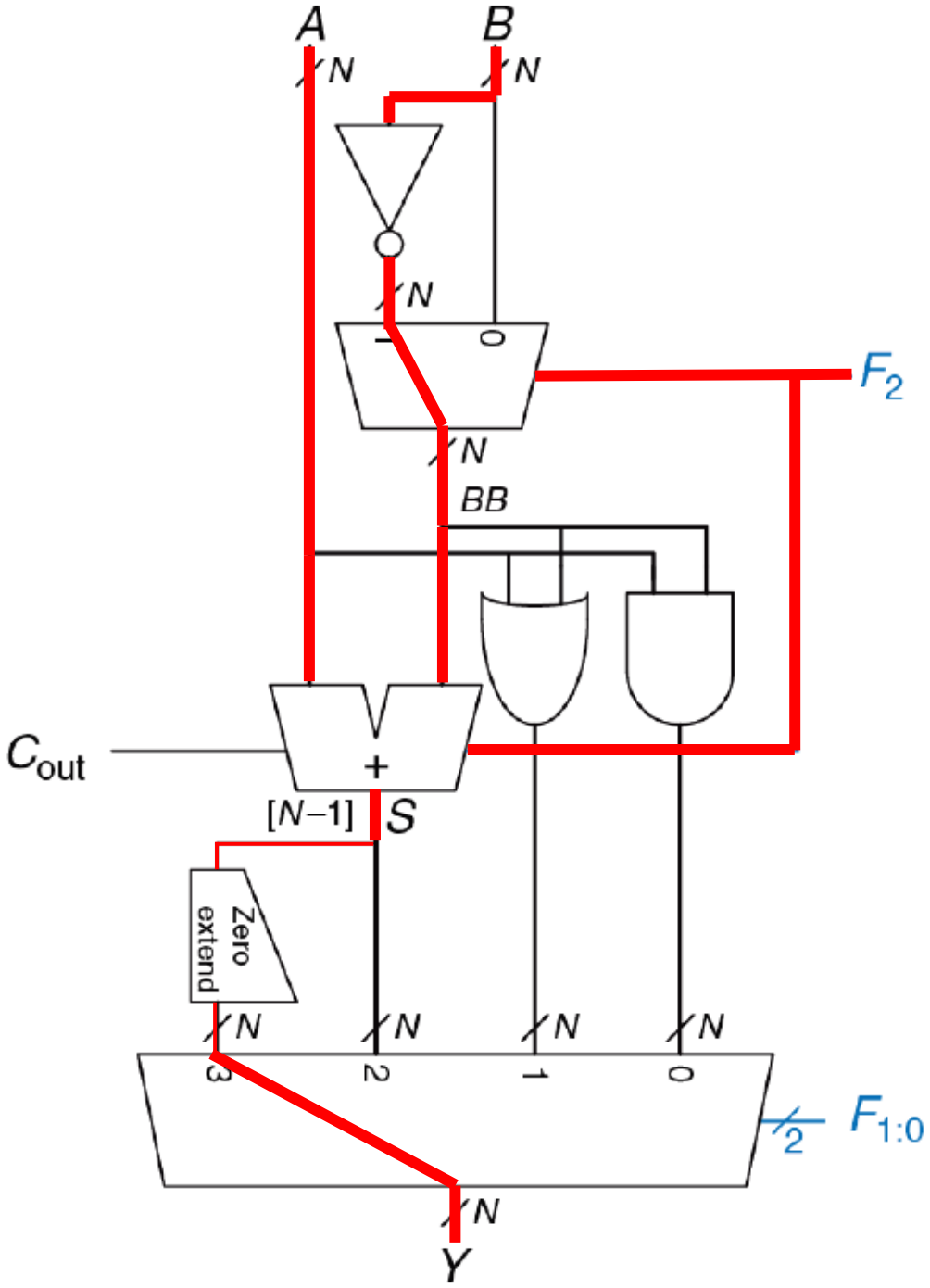
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



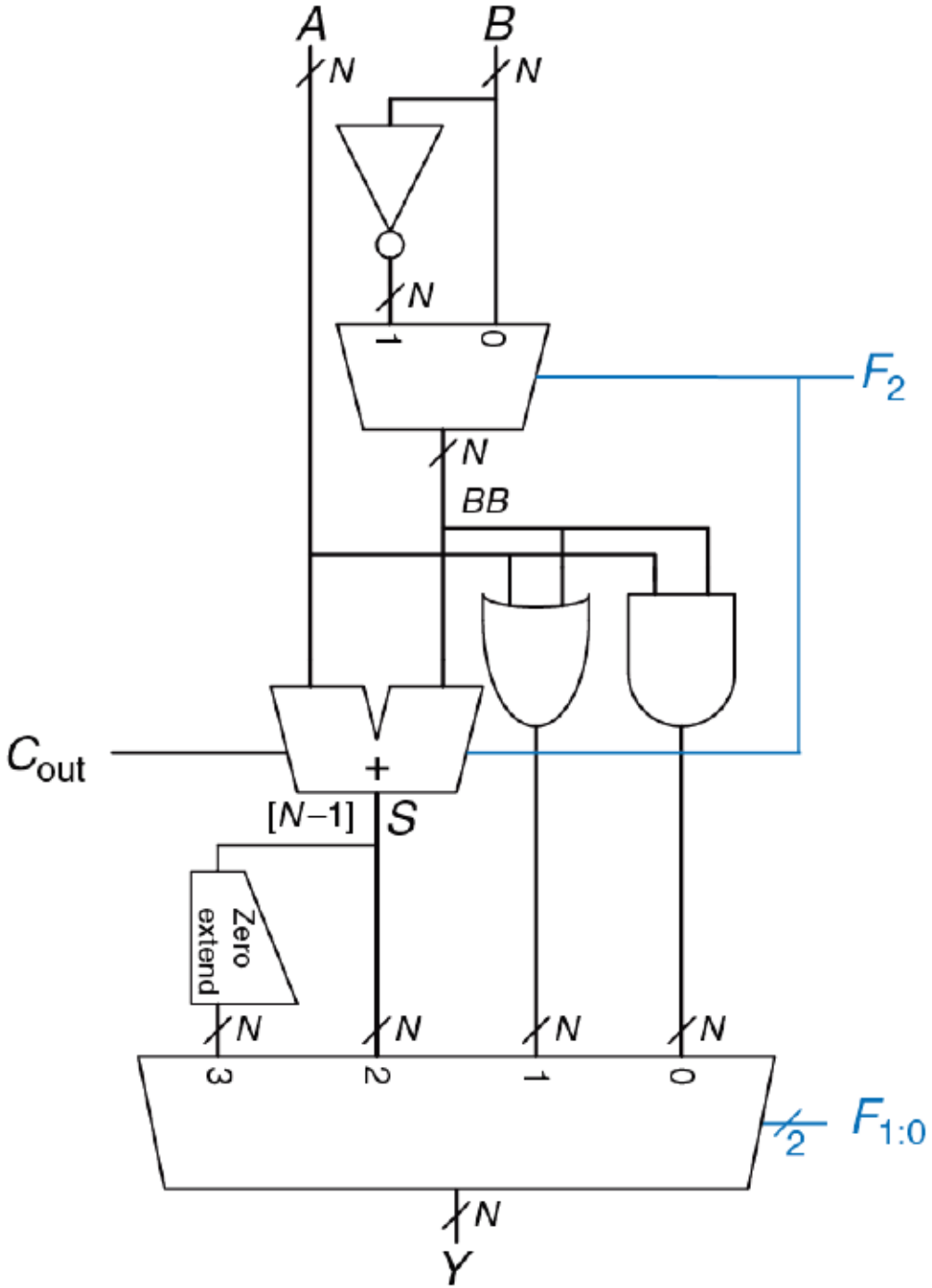
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



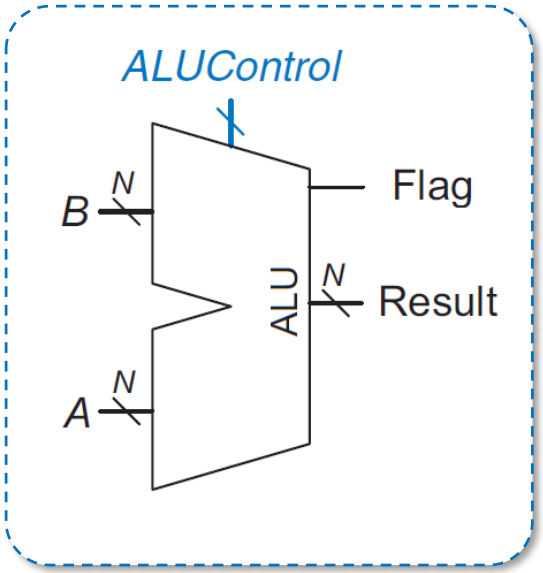
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

План лабораторной работы

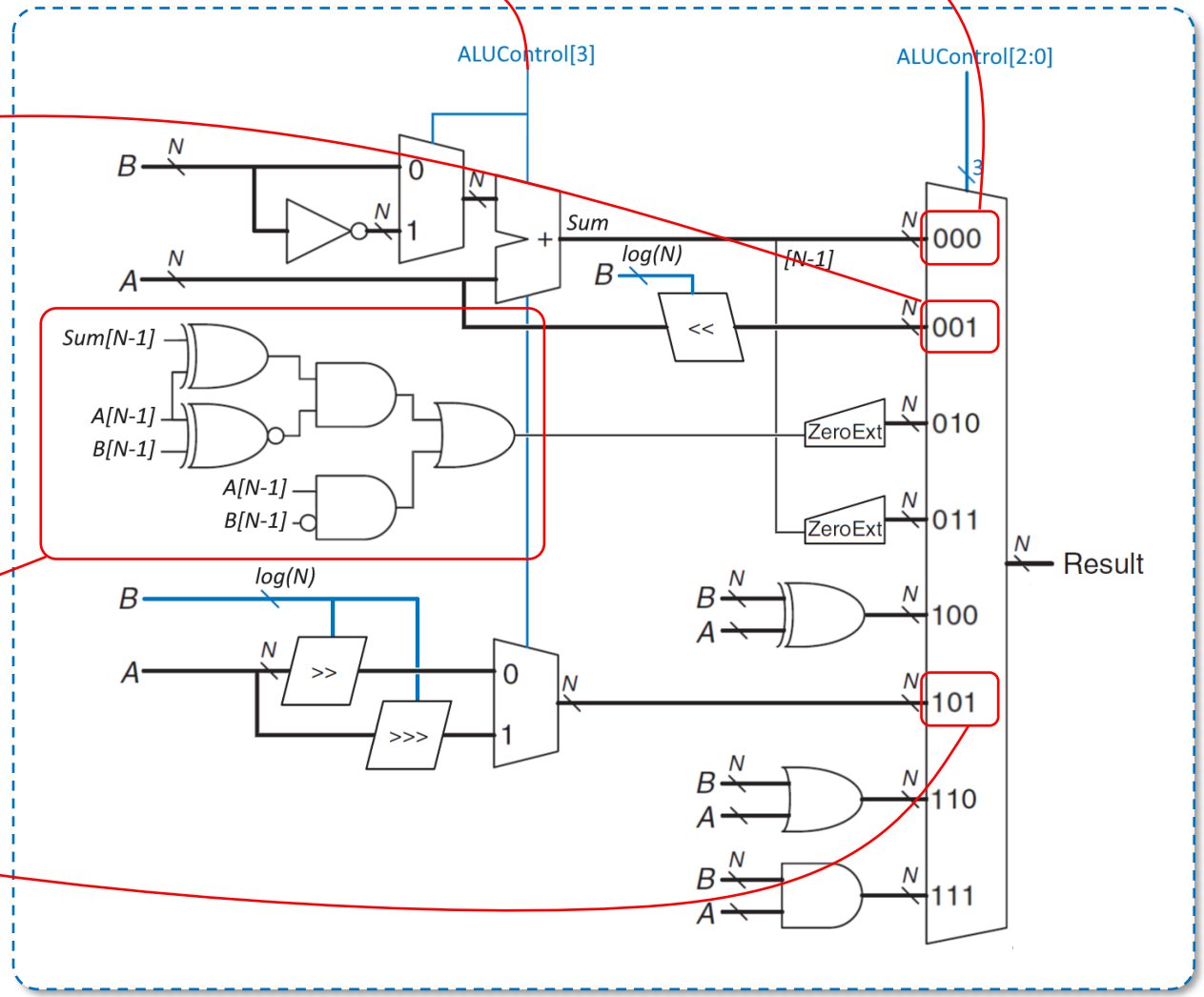
- ~~1 пара~~
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - ~~Тренинг по Vivado и Verilog HDL (**TS**)~~
  - ~~Задание на отладочном стен (**S**)~~
- 2 пара
  - ~~Арифметико-логическое устройство (**T**)~~
  - Описание АЛУ на Verilog HDL (**S**)
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

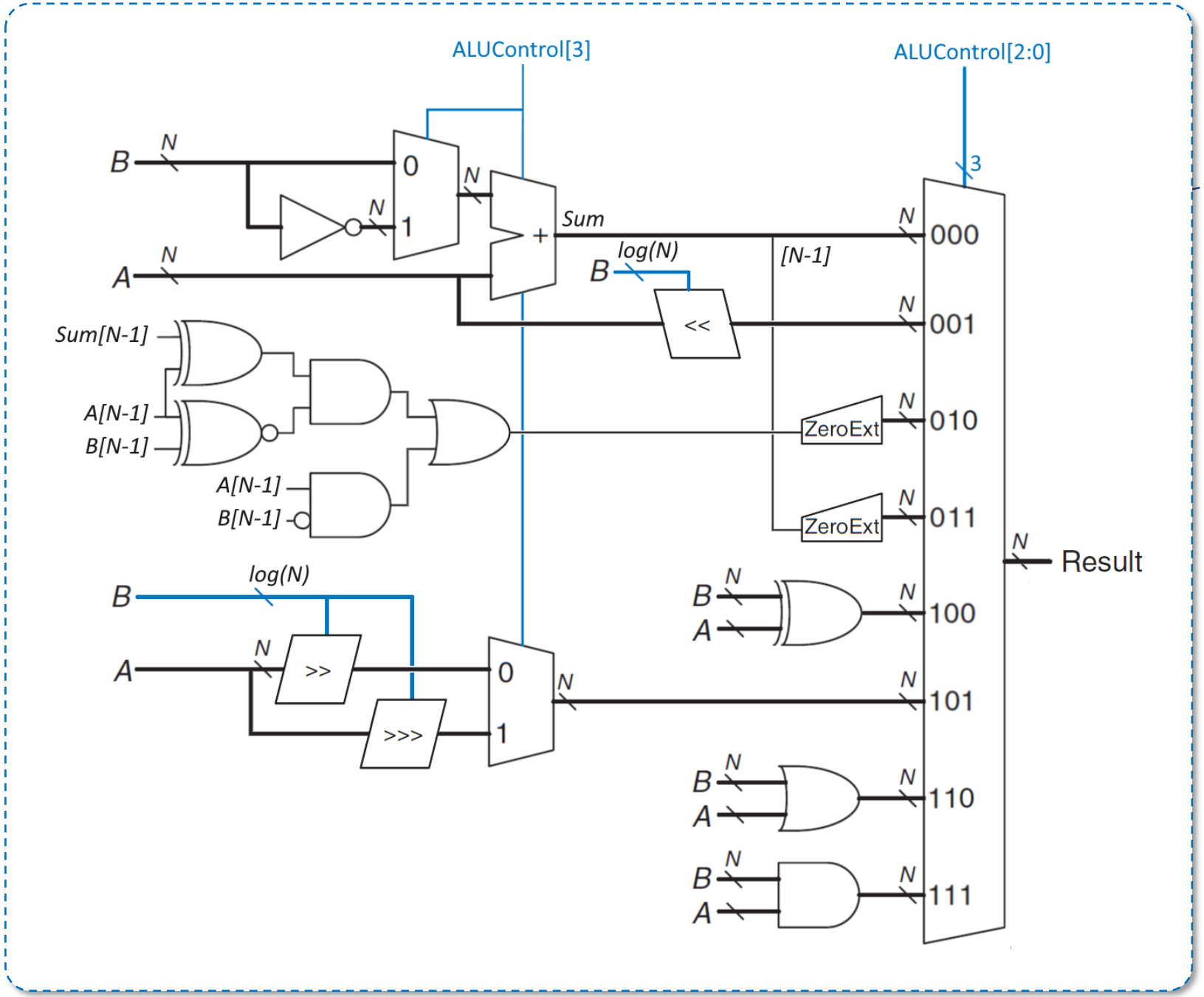
АЛУ RISC-V

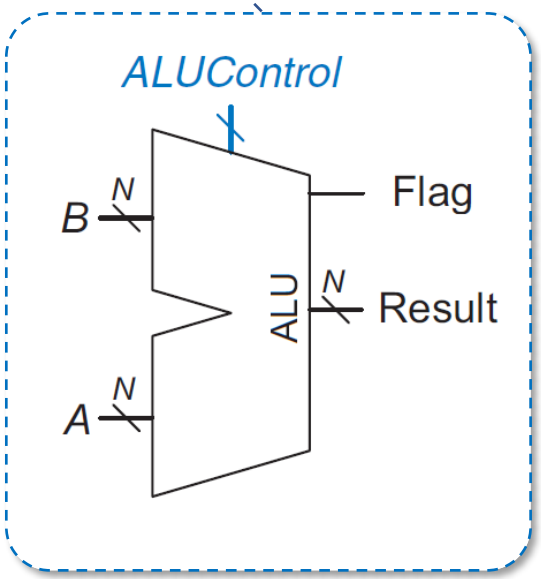
funct3	funct7	Description	Operation
000	0000000	add	rd = rs1 + rs2
000	0100000	sub	rd = rs1 − rs2
001	0000000	shift left logical	rd = rs1 << $rs2_{4:0}$
010	0000000	set less than	rd = (rs1 < rs2)
011	0000000	set less than unsigned	rd = (rs1 < rs2)
100	0000000	xor	rd = rs1 ^ rs2
101	0000000	shift right logical	rd = rs1 >> $rs2_{4:0}$
101	0100000	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
110	0000000	or	rd = rs1 | rs2
111	0000000	and	rd = rs1 & rs2
000	–	branch if =	if (rs1 == rs2) PC = BTA
001	–	branch if ≠	if (rs1 ≠ rs2) PC = BTA
100	–	branch if <	if (rs1 < rs2) PC = BTA
101	–	branch if ≥	if (rs1 ≥ rs2) PC = BTA
110	–	branch if < unsigned	if (rs1 < rs2) PC = BTA
111	–	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA



funct3	funct7	Description	Operation
000	0000000	add	rd = rs1 + rs2
000	0100000	sub	rd = rs1 − rs2
001	0000000	shift left logical	rd = rs1 << $rs2_{4:0}$
010	0000000	set less than	rd = (rs1 < rs2)
011	0000000	set less than unsigned	rd = (rs1 < rs2)
100	0000000	xor	rd = rs1 ^ rs2
101	0000000	shift right logical	rd = rs1 >> $rs2_{4:0}$
101	0100000	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
110	0000000	or	rd = rs1 | rs2
111	0000000	and	rd = rs1 & rs2
000	–	branch if =	if (rs1 == rs2) PC = BTA
001	–	branch if ≠	if (rs1 ≠ rs2) PC = BTA
100	–	branch if <	if (rs1 < rs2) PC = BTA
101	–	branch if ≥	if (rs1 ≥ rs2) PC = BTA
110	–	branch if < unsigned	if (rs1 < rs2) PC = BTA
111	–	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA




ALUControl[3]
ALUControl[2:0]
B
A
N
0
1
+
Sum
log(N)
<<
[N-1]
3
000
001
010
011
100
101
110
111
Result
ZeroExt
Sum[N-1]
A[N-1]
B[N-1]
>>
>>>
&

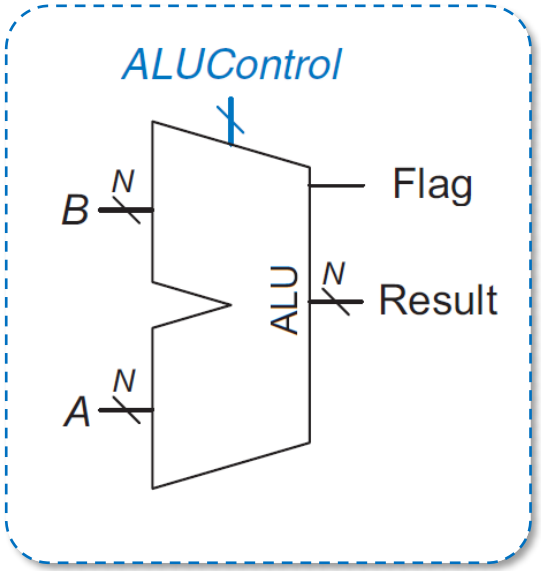

ALUControl
B
A
N
ALU
Flag
Result

АЛУ RISC-V

```
`define ADD 5'b00000
…
…
case (ALUOp)
        `ADD : Result = …
…
…
```

ALUOp = {flag, addsub, aluop}		
0 0 000		
0 1 000		
0 0 001		
0 0 010	Result = signed(A < B)	Flag = 0
0 0 011	Result = (A < B)	Flag = 0
0 0 100	Result = A ^ B	Flag = 0
0 0 101	Result = A >> B	Flag = 0
0 1 101	Result = signed(A) >>> B	Flag = 0
0 0 110	Result = A | B	Flag = 0
0 0 111	Result = A & B	Flag = 0
1 1 000	Result = 0	Flag = (A == B)
1 1 001	Result = 0	Flag = (A != B)
1 1 100	Result = 0	Flag = signed(A < B)
1 1 101	Result = 0	Flag = signed(A ≥ B)
1 1 110	Result = 0	Flag = (A < B)
1 1 111	Result = 0	Flag = (A ≥ B)

```
le ALU_RISCV (
  put  [4:0]   ALUOp,
  put  [31:0]  A,
  input  [31:0]  B,
  output [31:0]  Result,
  output         Flag
);
```

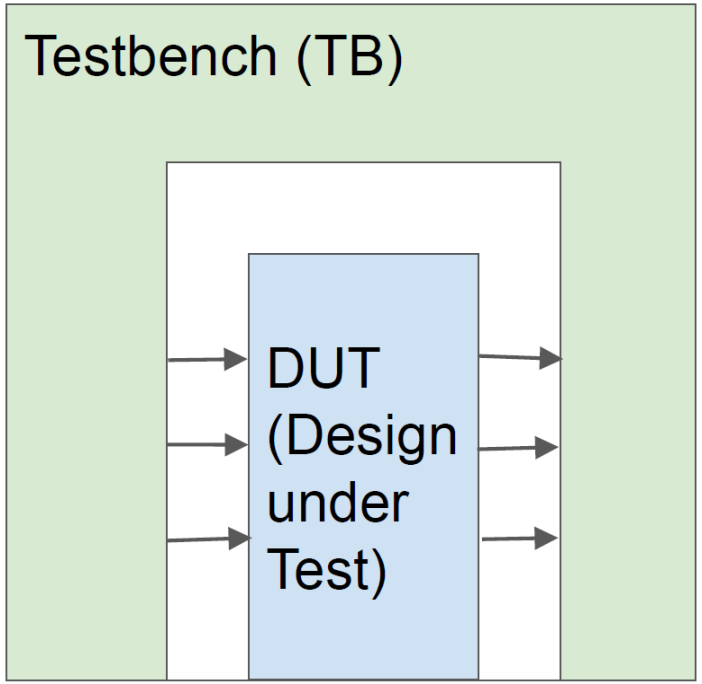


План лабораторной работы

- ~~1 пара~~
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - ~~Тренинг по Vivado и Verilog HDL (**TS**)~~
  - ~~Задание на отладочном стен (**S**)~~
- 2 пара
  - ~~Арифметико-логическое устройство (**T**)~~
  - ~~Описание АЛУ на Verilog HDL (**S**)~~
  - Основы верификации цифровых блоков (**TS**)
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

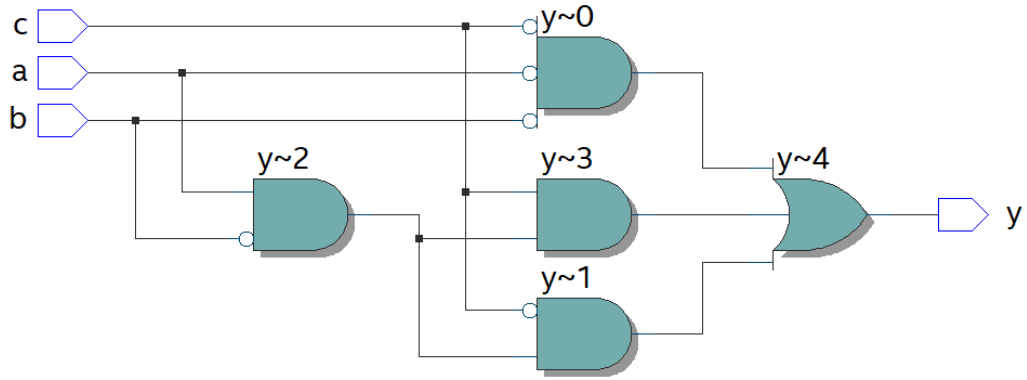
Тестовое окружение

Тестовое окружение (среда тестирования) – это модуль на HDL, который используется для тестирования другого модуля, называемого тестируемое устройство (Device under test, DUT)



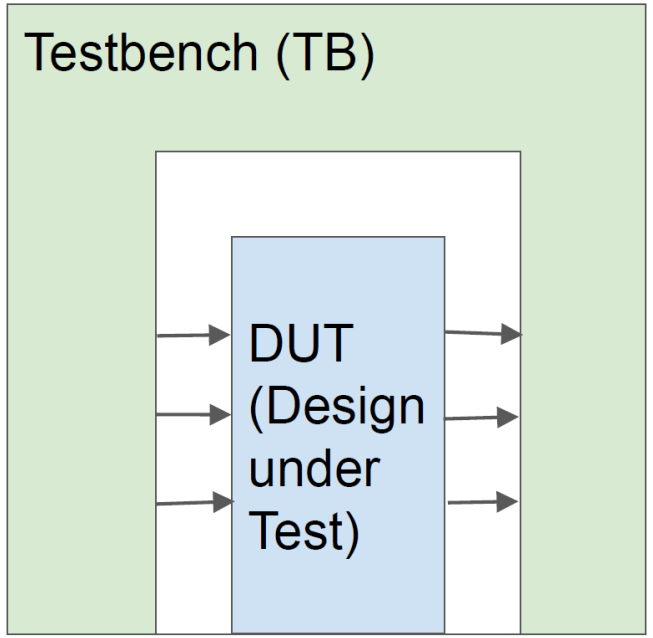
Device under test (DUT)

```
module my_module (
       input  a, b, c,
       output y
       );

assign y = a & ~b & ~c | a & ~b & c | a & ~b & c;

endmodule
```



Тестовое окружение (testbench)

```
`timescale 1ns / 1ps

module testbench ();

reg     a, b, c;
wire    y;

my_module dut (a, b, c, y);

initial begin
        a = 0; b = 0; c = 0; #10;
        if (y === 1)
                $display("Good");
        else
                $display("Bad");
        c = 1; #10;
end

endmodule
```



Пример тестирования модуля ALU

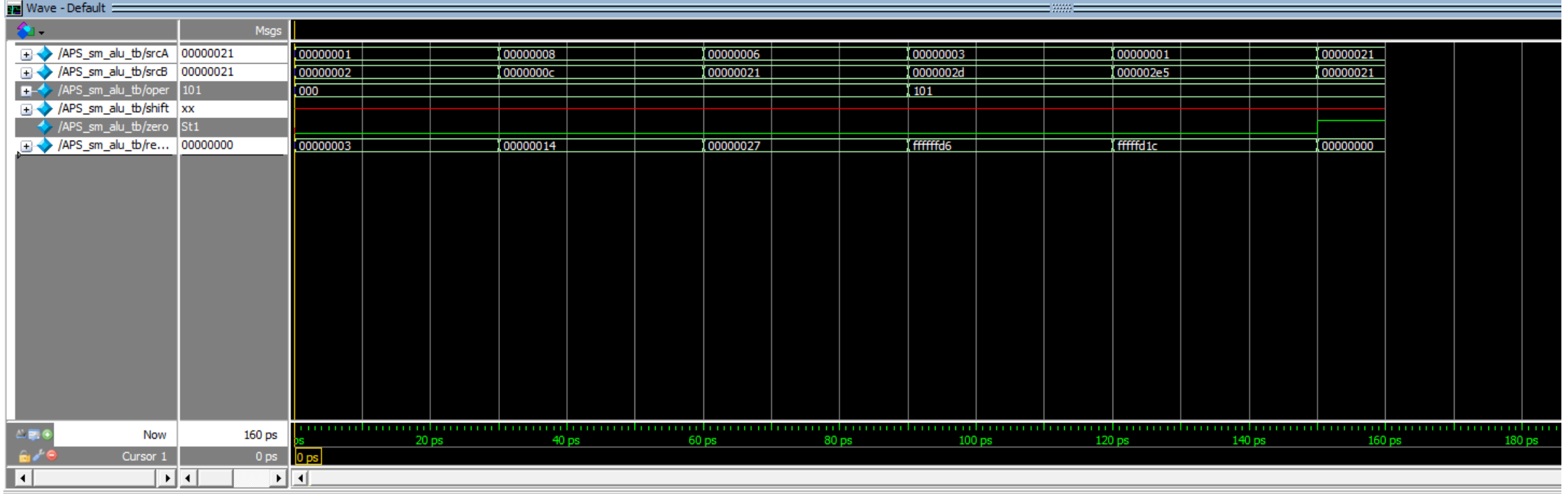
```
1  `timescale 1ps / 1ps
2  //ALU commands
3
4  module APS_sm_alu_tb();
5      reg  [31:0] srcA;
6      reg  [31:0] srcB;
7      reg  [ 2:0] oper;
8      reg  [ 4:0] shift;
9      wire        zero;
10     wire [31:0] result;
11
12 APS_sm_alu APS_sm_alu_inst
13 (
14    .srcA(srcA),
15    .srcB(srcB),
16    .oper(oper),
17    .shift(shift),
18    .zero(zero),
19    .result(result)
20 );
```

```
task alu_oper_test;
  input integer oper_tb;
  input integer srcA_tb;
  input integer srcB_tb;

  begin
  oper = oper_tb;
  srcB = srcB_tb;
  srcA = srcA_tb;
  #10;
  $display("srcA= %d", srcA, " srcB = %d", srcB, " result = %d", result);
  $display("Time = %t",$realtime);
 end
endtask
```

Все операции содержат *oper*, операнды *srcA* и *srcB*. Почему бы не упростить читабельность и работы с tb, используя **task**?
Входные аргументы **task**, разделяются на input/output.
Изменяя значение поля *oper* и операндов, можно задать любую из поддерживаемых операций и любые операнды, над которыми будет совершаться операция.
Так же, мы можем задать вывод необходимой информации в лог, а затем в tb, обращаться к **task** с необходимыми входными аргументами.

Пример использования **task**

```
initial begin
alu_oper_test(3'b000,1,2);
#20;
alu_oper_test(3'b000,8,12);
#20;
alu_oper_test(3'b000,6,33);
#20;
alu_oper_test(3'b101,3,45);
#20;
alu_oper_test(3'b101,1,741);
#20;
alu_oper_test(3'b101,33,33);
end
```



Transcript

```
# Errors: 0, Warnings: 0
#
# vsim -t 1ps -L altera_ver -L lpm_ver -L sgate_ver -L altera_mf_ver -L altera_lnsim_ver -L fiftyfivenm_ver -L rtl_work -L work -voptargs="+acc"  APS_sm_alu_tb
# vsim -t 1ps -L altera_ver -L lpm_ver -L sgate_ver -L altera_mf_ver -L altera_lnsim_ver -L fiftyfivenm_ver -L rtl_work -L work -voptargs=""+acc"" APS_sm_alu_tb
# Start time: 03:07:04 on Sep 23,2019
# Loading work.APS_sm_alu_tb
# Loading work.APS_sm_alu
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# srcA=          1 srcB =          2 result =          3
# Time =                    10
# srcA=          8 srcB =         12 result =         20
# Time =                    40
# srcA=          6 srcB =         33 result =         39
# Time =                    70
# srcA=          3 srcB =         45 result = 4294967254
# Time =                   100
# srcA=          1 srcB =        741 result = 4294966556
# Time =                   130
# srcA=         33 srcB =         33 result =          0
# Time =                   160
```

План лабораторной работы

- ~~1 пара~~
  - ~~О лабораторных работах (**T**)~~
  - ~~Введение в FPGA и Verilog HDL (**T**)~~
  - ~~Тренинг по Vivado и Verilog HDL (**TS**)~~
  - ~~Задание на отладочном стен (**S**)~~
- 2 пара
  - ~~Арифметико-логическое устройство (**T**)~~
  - ~~Описание АЛУ на Verilog HDL (**S**)~~
  - ~~Основы верификации цифровых блоков (**TS**)~~
  - Верификация АЛУ (**S**)
  - Проверка на отладочном стенде (**S**)

Задание

- Верифицировать разработанное АЛУ с помощью testbench. Продемонстрировать полученный результат преподавателю
- (*если осталось время*) Внедрить верифицированное АЛУ в отладочный стенд. В качестве входных данных и сигналов управления использовать переключатели switch на стенде. Информацию выводить на семисегментный индикатор в 16-ричном формате