

ОСНОВЫ

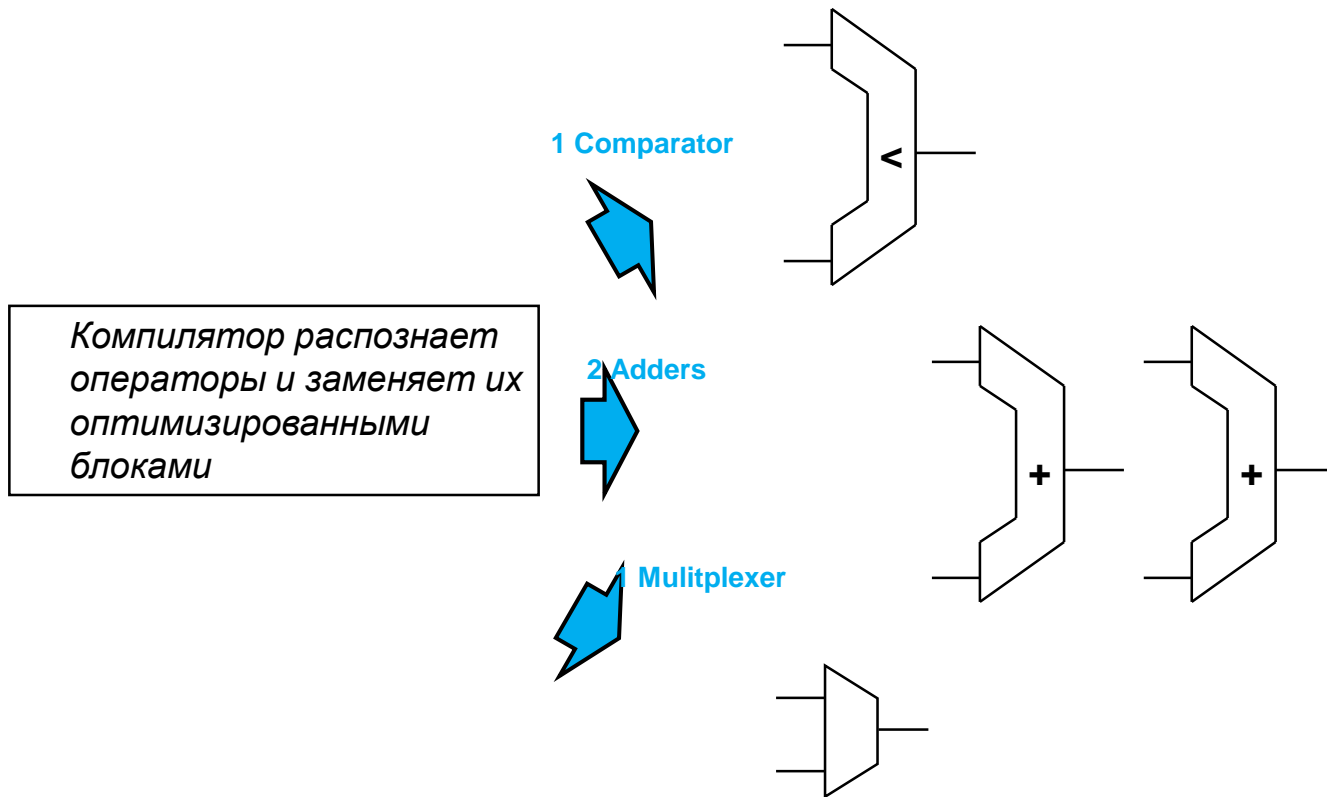
VerilogHDL/SystemVerilog

(синтез и моделирование)



Оптимизация Verilog описания

Замена операторов арифметическими блоками



Оптимизация Verilog описания

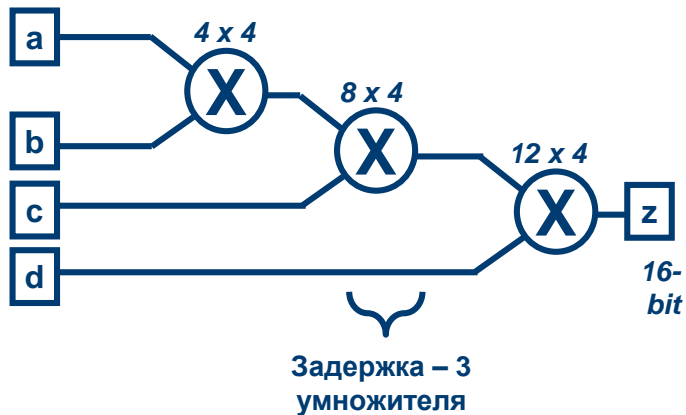
- ❑ Симметрирование операторов
- ❑ Конвейеризация
- ❑ Совместное использование ресурсов

Симметрирование операторов (пример)

□ a, b, c, d: 4-bit векторы

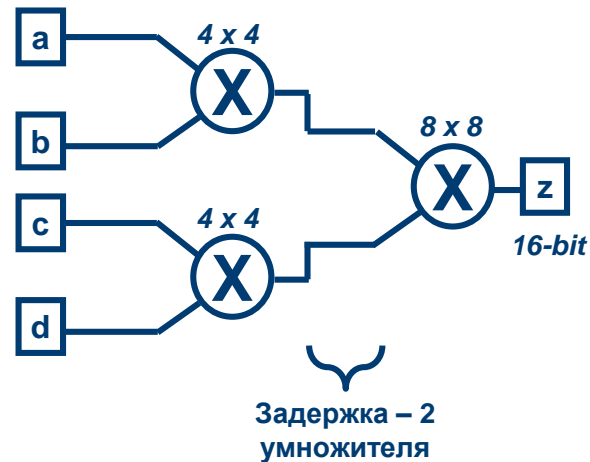
Не симметрированный

$$z \leq a * b * c * d$$



Симметрированный

$$z \leq (a * b) * (c * d)$$

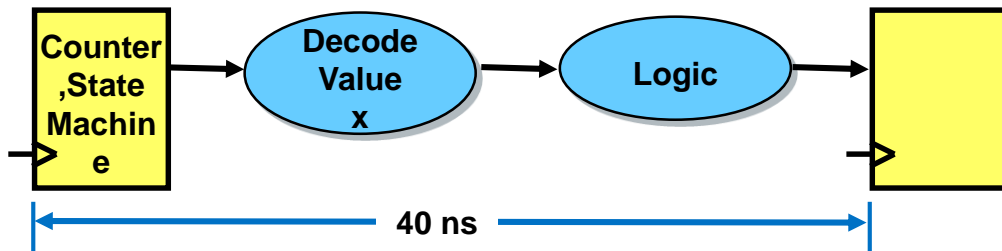


Конвейеризация

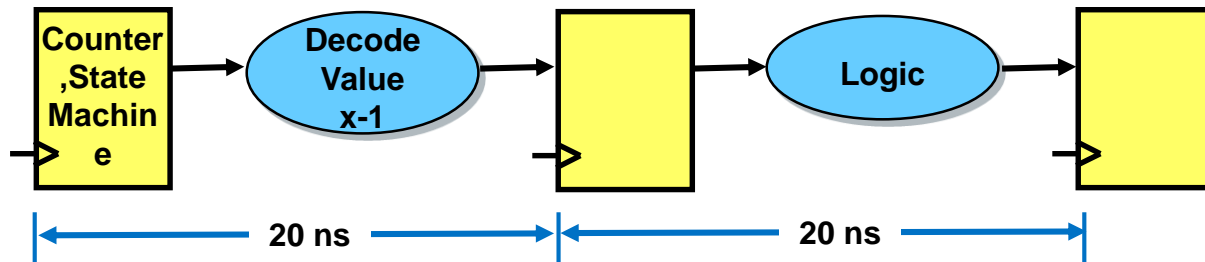
- ❑ Целенаправленное добавление триггеров в критическую цепь распространения комбинационных сигналов
- ❑ Увеличивает максимальную частоту работы
- ❑ Добавляет такт (такты) задержки
 - ✓ Больше тактов требуется для получения результата на выходе
- ❑ Некоторые компиляторы могут осуществлять конвейеризацию автоматически

Добавление уровня конвейеризации

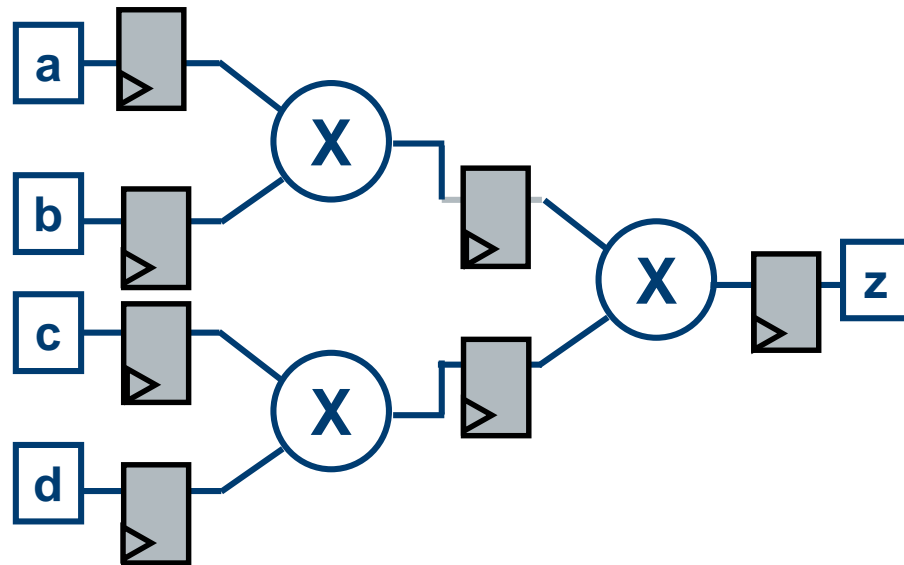
25 MHz System



50 MHz System



Конвейеризация 4-х входового умножителя



Пример: Умножитель 4-х чисел

```
module ex1 (clk, a, b, c, d, res);  
input [7:0] a, b, c, d;  
output reg [31:0] res;  
input clk;
```

```
reg [7:0] a_, b_, c_, d_;
```

```
always @(posedge clk)  
begin
```

```
a_ <= a;
```

```
b_ <= b;
```

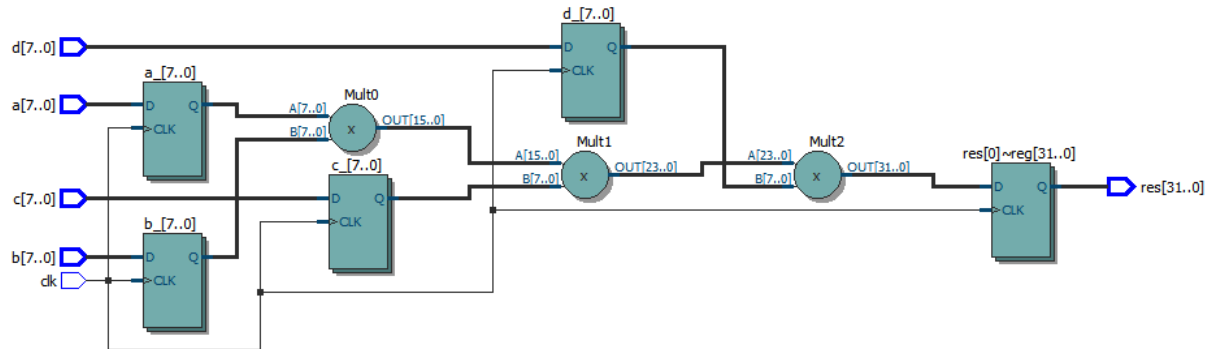
```
c_ <= c;
```

```
d_ <= d;
```

```
end
```

```
always @(posedge clk)  
res <= a_ * b_ * c_ * d_;
```

```
endmodule
```



Быстродействие

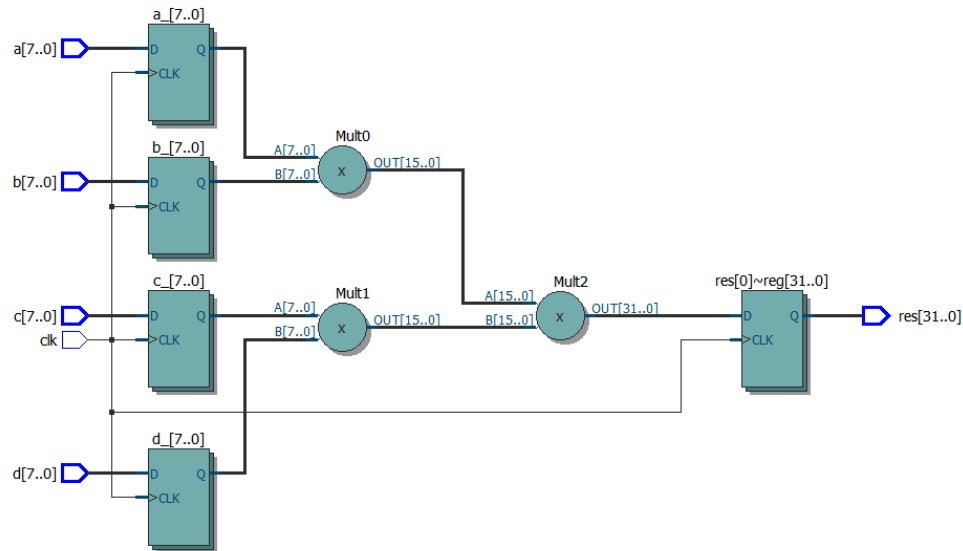
Fmax — 62
MHz

Аппаратные затраты

Логических элементов — 32
Триггеров —
32
DSP модулей (9*9) —
6

Умножитель 4-х чисел (симметрирован)

```
module ex2 (clk, a, b, c, d, res);  
input [7:0] a, b, c, d;  
output reg [31:0] res;  
input clk;  
  
reg [7:0] a_, b_, c_, d_;  
  
always @(posedge clk)  
begin  
a_ <= a;  
b_ <= b;  
c_ <= c;  
d_ <= d;  
end  
  
always @(posedge clk)  
res <= (a_ * b_) * (c_ * d_);  
  
endmodule
```



Быстродействие

Fmax

— **123 MHz**

Аппаратные затраты

Логических элементов — 0

Триггеров — 0

DSP модулей (9*9) — 4

Умножитель 4-х чисел (конвейеризированный)

```
module ex3 (clk, a, b, c, d, res);
input [7:0] a, b, c, d;
output reg [31:0] res;
input clk;
```

```
reg [7:0] a_, b_, c_, d_;
reg [15:0] int_a, int_b;
always @(posedge clk)
```

```
begin
```

```
a_ <= a;
b_ <= b;
c_ <= c;
d_ <= d;
end
```

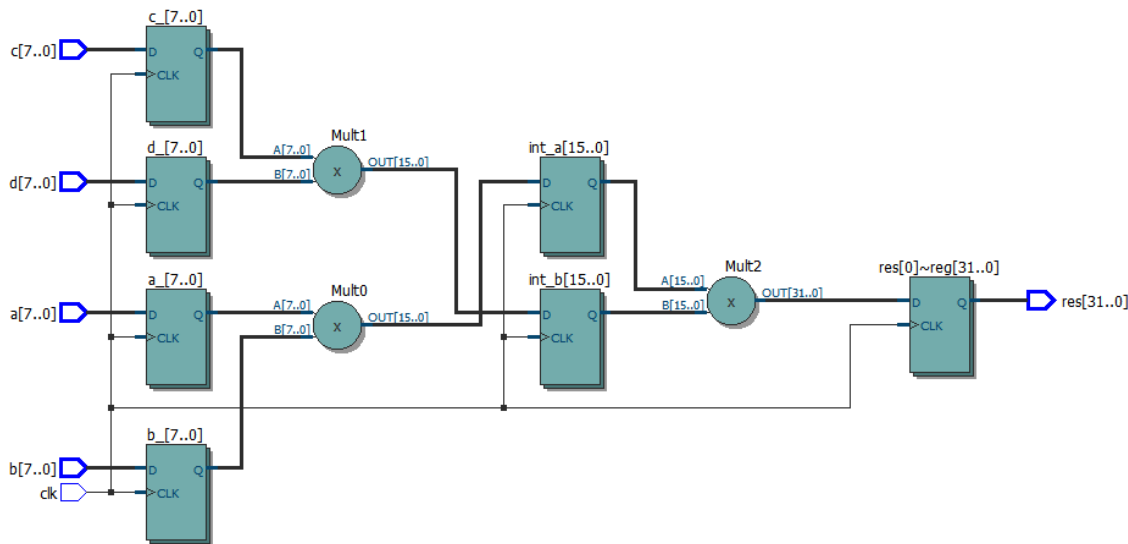
```
always @(posedge clk)
```

```
begin
```

```
int_a <= a_ * b_;
int_b <= c_ * d_;
end
```

```
always @(posedge clk)
res <= int_a * int_b;
```

```
endmodule
```

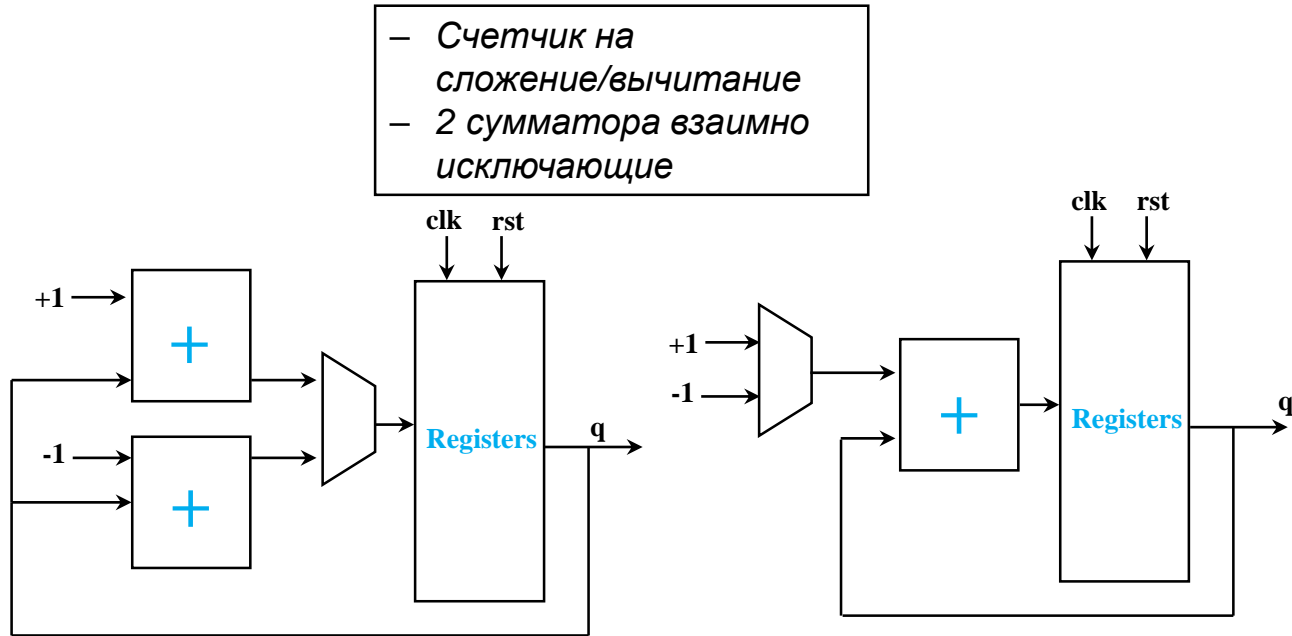


Быстродействие		Аппаратные затраты	
Fmax	– 215 MHz	Логических элементов	– 0
		Триггеров	– 0
		DSP модулей (9*9)	– 4

Совместное использование ресурсов

- ❑ Сокращает количество арифметических блоков для реализации операторов
 - ✓ Уменьшается используемая площадь СБИС
- ❑ Два типа
 - ✓ Совместное использование взаимно исключающих операторов
 - ✓ Совместное использование общей части арифметических выражений
- ❑ Компилятор может осуществлять совместное использование ресурсов автоматически
 - ✓ Опция может быть включена/выключена

Взаимно исключающие операторы

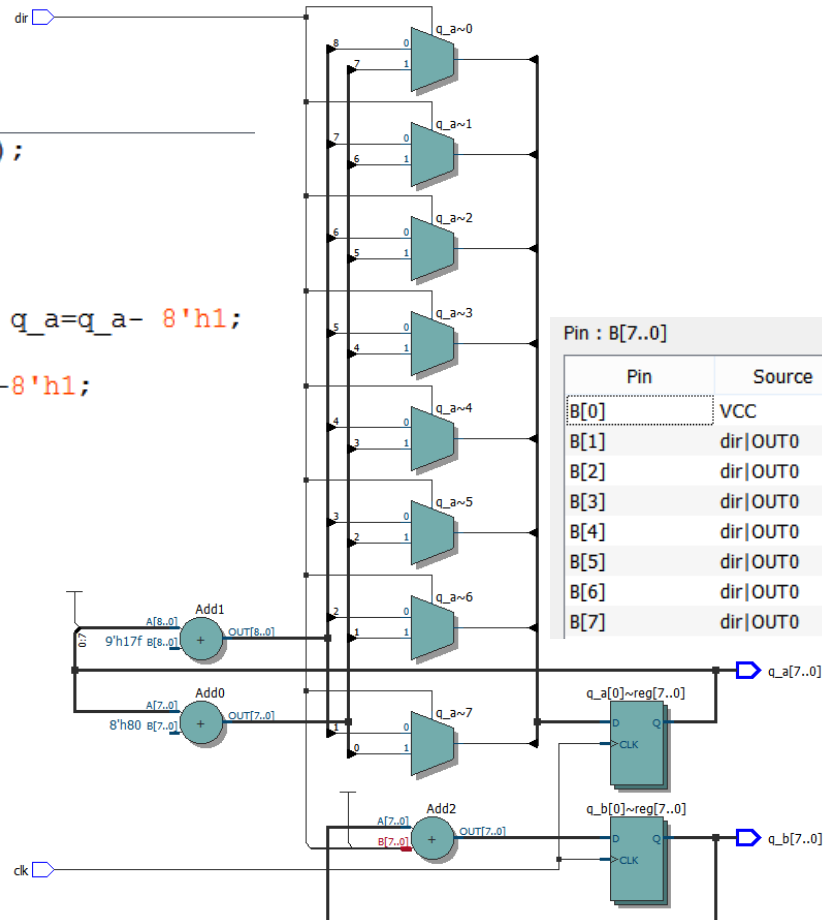


Пример

```
module ex4 (clk, dir, q_a, q_b);
input clk, dir;
output reg [7:0] q_a, q_b;

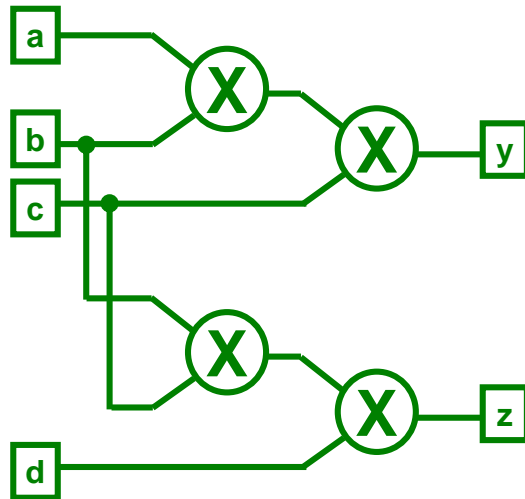
always @(posedge clk)
if (dir) q_a <= q_a+8'h1; else q_a=q_a- 8'h1;

wire [7:0] tmp = (dir)? 8'h1: -8'h1;
always @(posedge clk)
q_b <= q_b + tmp;
endmodule
```



Сколько блоков умножения?

$$\begin{aligned} y &\leftarrow a * b * c \\ z &\leftarrow b * c * d \end{aligned}$$

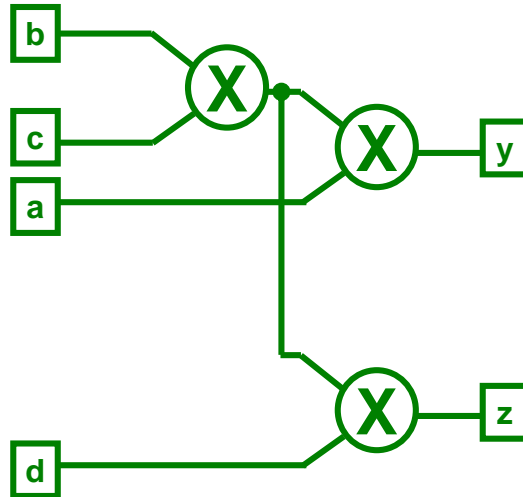


4 блока

Сколько блоков умножения?

3 блока

$$\begin{aligned}y &\leq a * (b * c) \\z &\leq (b * c) * d\end{aligned}$$



- Совместное использование общих частей арифметических выражений
- Некоторые средства синтеза реализуют это автоматически, а некоторые нет.
- Использование скобок помогает компилятору минимизировать число блоков
- Если $(b*c)$ используется многократно, то целесообразно использовать отдельный сигнал

Пример: Два умножителя с общими операндами

```
module ex5 (clk, a, b, c, d, res_a, res_b);  
input [7:0] a, b, c, d;  
output reg [23:0] res_a, res_b;  
input clk;
```

```
reg [7:0] a_, b_, c_, d_;
```

```
always @(posedge clk)
```

```
begin
```

```
a_ <= a;
```

```
b_ <= b;
```

```
c_ <= c;
```

```
d_ <= d;
```

```
end
```

```
always @(posedge clk)
```

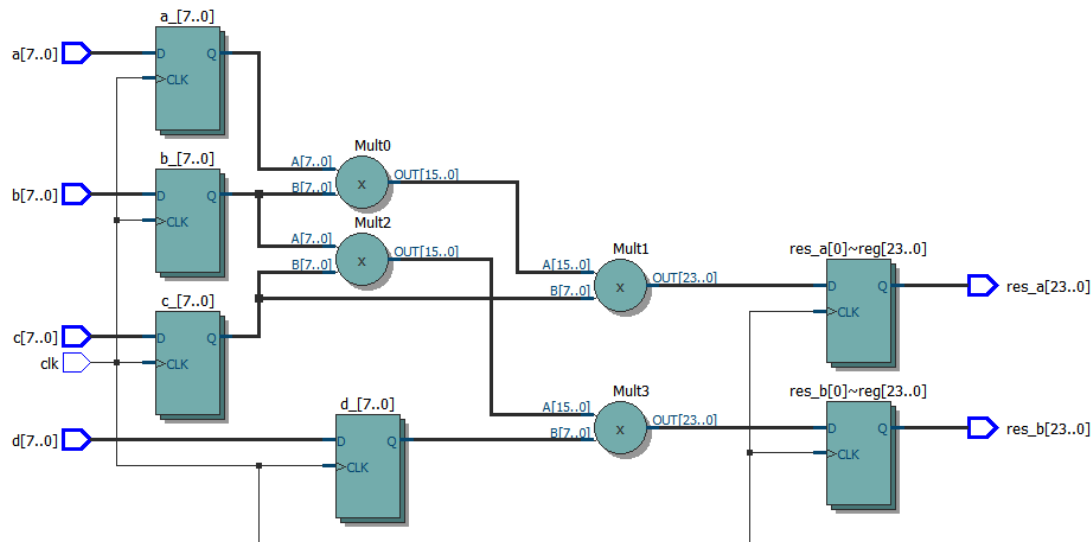
```
begin
```

```
res_a <= a_ * b_ * c_;
```

```
res_b <= b_ * c_ * d_;
```

```
end
```

```
endmodule
```



Fmax

– 124 MHz

Аппаратные затраты

Логических элементов – 0

Триггеров – 0

DSP модулей (9*9) – 6

Два умножителя с общими операндами

```
module ex5 (clk, a, b, c, d, res_a, res_b);  
input [7:0] a, b, c, d;  
output reg [23:0] res_a, res_b;  
input clk;
```

```
reg [7:0] a_, b_, c_, d_;
```

```
always @(posedge clk)
```

```
begin
```

```
a_ <= a;
```

```
b_ <= b;
```

```
c_ <= c;
```

```
d_ <= d;
```

```
end
```

```
always @(posedge clk)
```

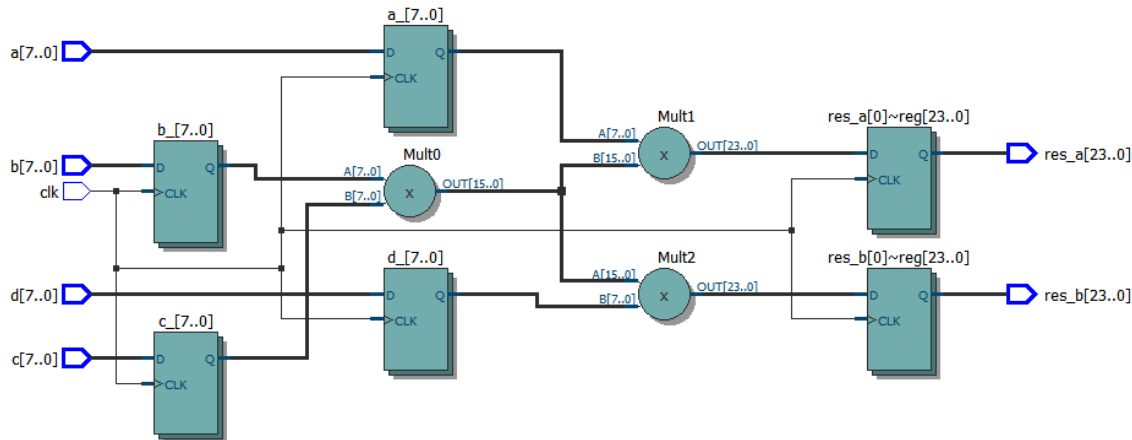
```
begin
```

```
res_a <= a_ * (b_ * c_);
```

```
res_b <= (b_ * c_) * d_;
```

```
end
```

```
endmodule
```



Fmax – 124 MHz

Аппаратные затраты

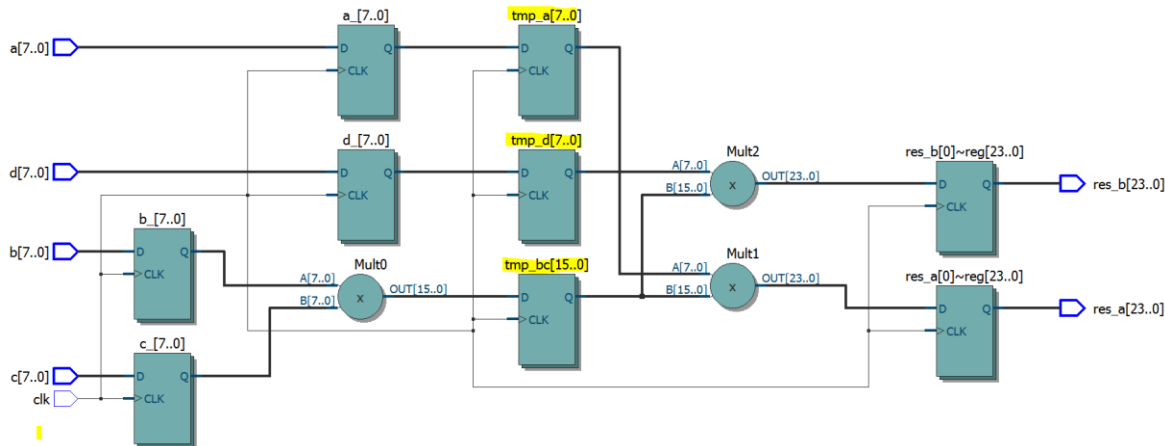
Логических элементов – 0

Триггеров – 0

DSP модулей (9*9) – 5

Конвейеризация

```
1 module ex6 (clk, a, b, c, d, res_a, res_b);
2   input [7:0] a, b, c, d;
3   output reg [23:0] res_a, res_b;
4   input clk;
5
6   reg [7:0] a_, b_, c_, d_;
7   reg [7:0] tmp_a, tmp_d;
8   reg [15:0] tmp_bc;
9
10  always @(posedge clk)
11  begin
12    a_ <= a;
13    b_ <= b;
14    c_ <= c;
15    d_ <= d;
16  end
17
18  always @(posedge clk)
19  begin
20    tmp_a <= a_;
21    tmp_bc <= b_ * c_;
22    tmp_d <= d_;
23  end
24
25  always @(posedge clk)
26  begin
27    res_a <= tmp_a * tmp_bc;
28    res_b <= tmp_d * tmp_bc;
29  end
30
31 endmodule
```



Fmax – 183 MHz

Аппаратные затраты

Логических элементов	– 0
Триггеров	– 0
DSP модулей (9*9)	– 5



Триггеры Защелки (Latch)

Защелки (Latches) vs. Триггеры (Registers)

- ❑ В СБИС Altera реализованы синхронные триггеры, а не триггеры защелки
- ❑ Триггеры защелки реализуются с помощью комбинационной логики, что делает временной анализ более сложным
 - ✓ СБИС с Look-up table (LUT) – FPGA, используют LUT (таблицы перекодировки)
 - ✓ СБИС с Product-term – MAX, используют дополнительные product-terms (программируемые матрицы И и ИЛИ)
- ❑ Рекомендации
 - ✓ Использовать в проектах синхронные триггеры
 - ✓ Следить за появлением триггеров защелок.
 - Триггеры защелки могут появляться на выходах комбинационных цепей, когда результат не определен для набора входных значений.

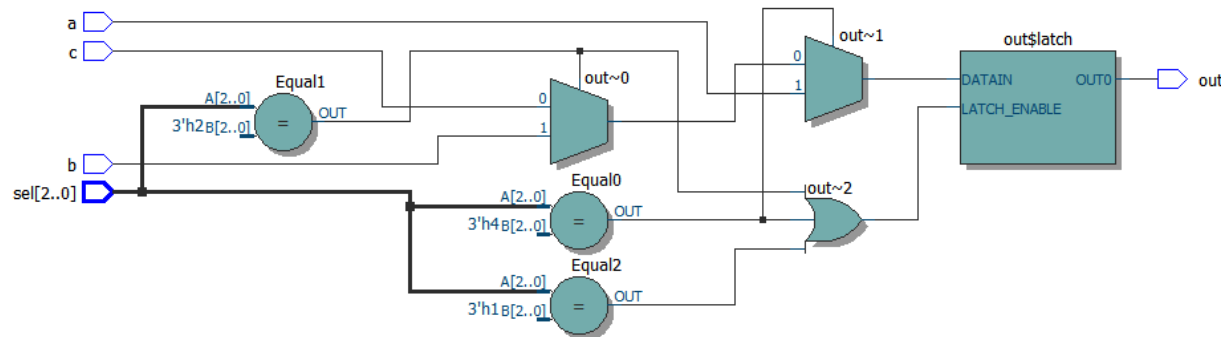
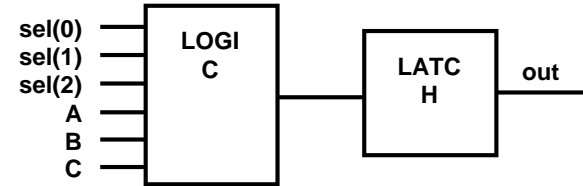
При использовании оператора IF-ELSE ...

- ❑ Перебирайте все варианты
 - ✓ Не перечисленные варианты порождают триггеры защелки
- ❑ В комбинационных процессах инициализируйте переменные (сигналы) до их использования
 - ✓ Использование не инициализированных переменных (сигналов) может порождать триггеры защелки
- ❑ Для создания эффективного кода продумайте
 - ✓ Назначение начальных значений и явное покрытие вариантов, отличных от начальных значений

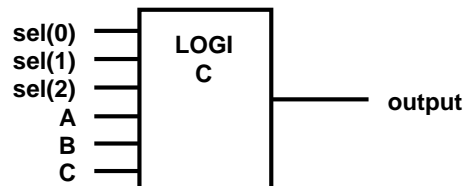
Нежелательные триггеры защелки

- ❑ Комбинационный процесс, который не покрывает все возможные комбинации входных сигналов, порождает триггер-защелку.

```
1 module ex1 (sel, a, b, c, out);  
2   input [2:0] sel;  
3   input a, b, c;  
4   output reg out;  
5  
6   always @*  
7     if (sel == 3'b001) out = a;  
8     else if (sel == 3'b010) out = b;  
9         else if (sel == 3'b100) out = c;  
10  
11 endmodule
```



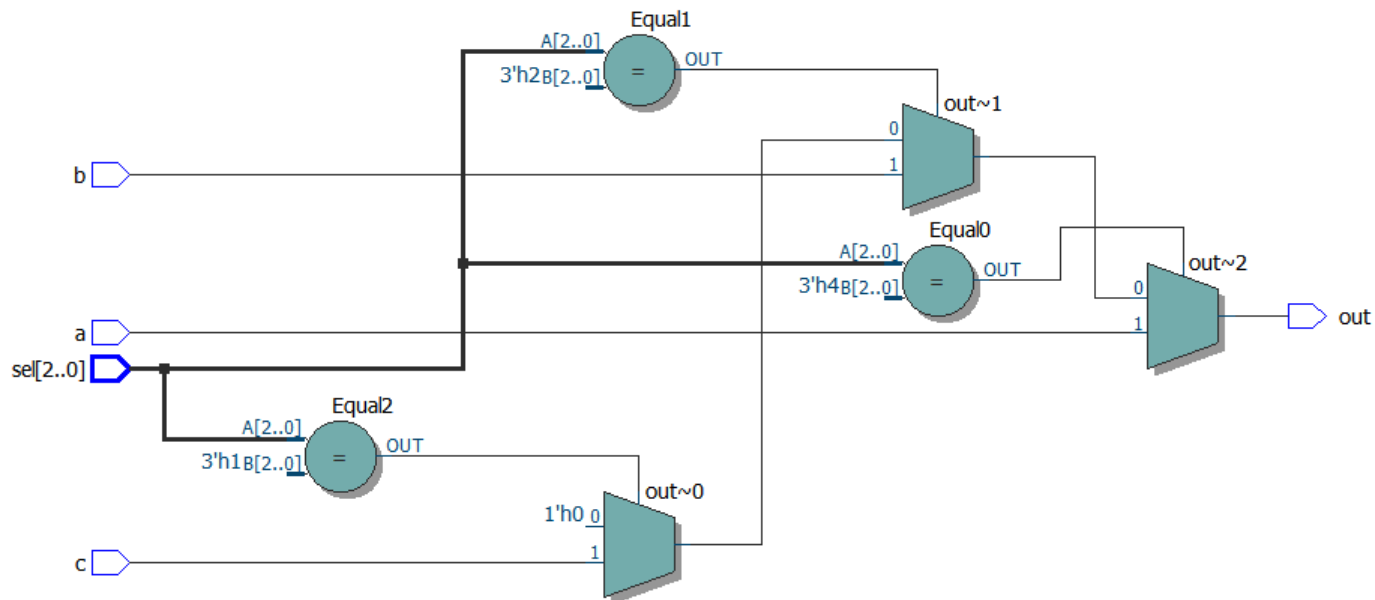
❑ Терминируйте IF-ELSE оператор



```
1 module ex1_1 (sel, a, b, c, out);
2   input [2:0] sel;
3   input a, b, c;
4   output reg out;
5
6   always @*
7     if (sel == 3'b001) out = a;
8     else if (sel == 3'b010) out = b;
9         else if (sel == 3'b100) out = c;
10        else out = 1'b0;
11
12 endmodule
```

```
1 module ex1_2 (sel, a, b, c, out);
2   input [2:0] sel;
3   input a, b, c;
4   output reg out;
5
6   always @*
7   begin
8     out = 1'b0;
9
10    if (sel == 3'b001) out = a;
11    else if (sel == 3'b010) out = b;
12        else if (sel == 3'b100) out = c;
13    end
14
15 endmodule
```

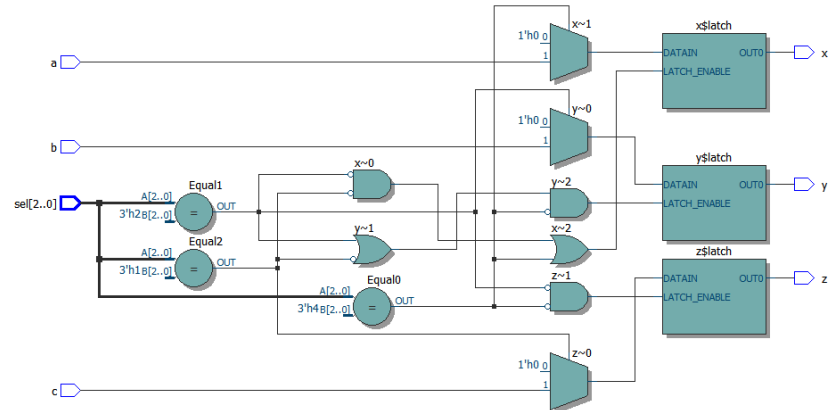

RTL Viewer



Взаимоисключающие IF-ELSE порождающие триггеры защелки

- ❑ Остерегайтесь использования нежелательной взаимозависимости условий

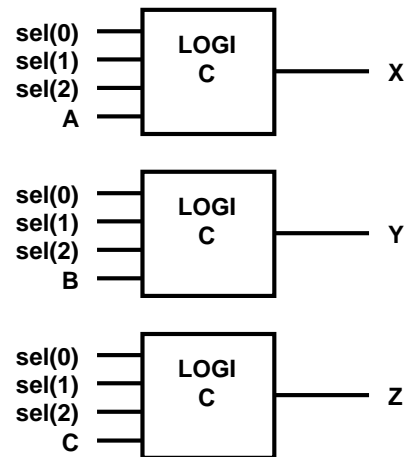
```
1 module ex2 (sel, a, b, c, x, y, z);
2   input [2:0] sel;
3   input a, b, c;
4   output reg x, y, z;
5
6   always @*
7     if (sel == 3'b001) x = a;
8     else if (sel == 3'b010) y = b;
9     else if (sel == 3'b100) z = c;
10    else
11      begin
12        x = 1'b0;
13        y = 1'b0;
14        z = 1'b0;
15      end
16  end
17 endmodule
```



Исправление (1)

- Используйте независимые операторы IF и терминируйте их

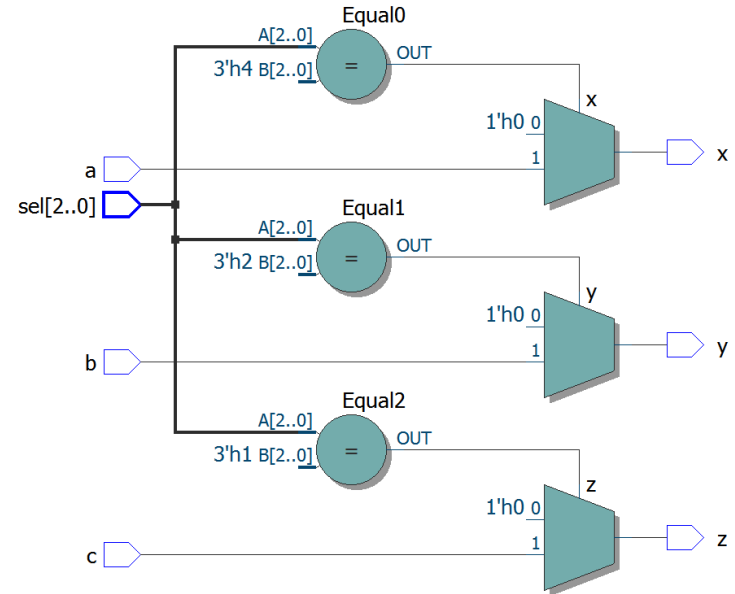
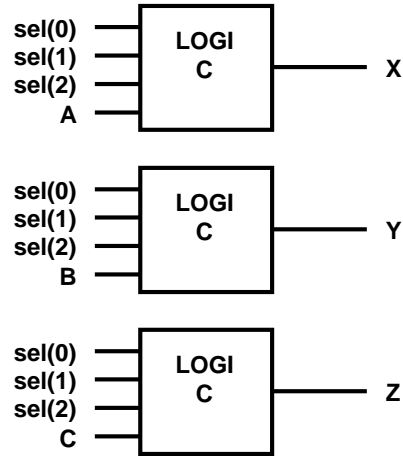
```
1 module ex2_1 (sel, a, b, c, x, y, z);
2   input [2:0] sel;
3   input a, b, c;
4   output reg x, y, z;
5
6   always @*
7     if (sel == 3'b001) x = a; else x = 1'b0;
8
9   always @*
10    if (sel == 3'b010) y = b; else y = 1'b0;
11
12   always @*
13    if (sel == 3'b100) z = c; else z = 1'b0;
14
15 endmodule
```



Исправление (2)

- Используйте независимые операторы IF и терминируйте их

```
1 module ex2_2 (sel, a, b, c, x, y, z);
2   input [2:0] sel;
3   input a, b, c;
4   output reg x, y, z;
5
6   always @*
7   begin
8     x = 1'b0;
9     y = 1'b0;
10    z = 1'b0;
11
12    if (sel == 3'b001) x = a;
13    if (sel == 3'b010) y = b;
14    if (sel == 3'b100) z = c;
15
16  end
17  |
18 endmodule
```

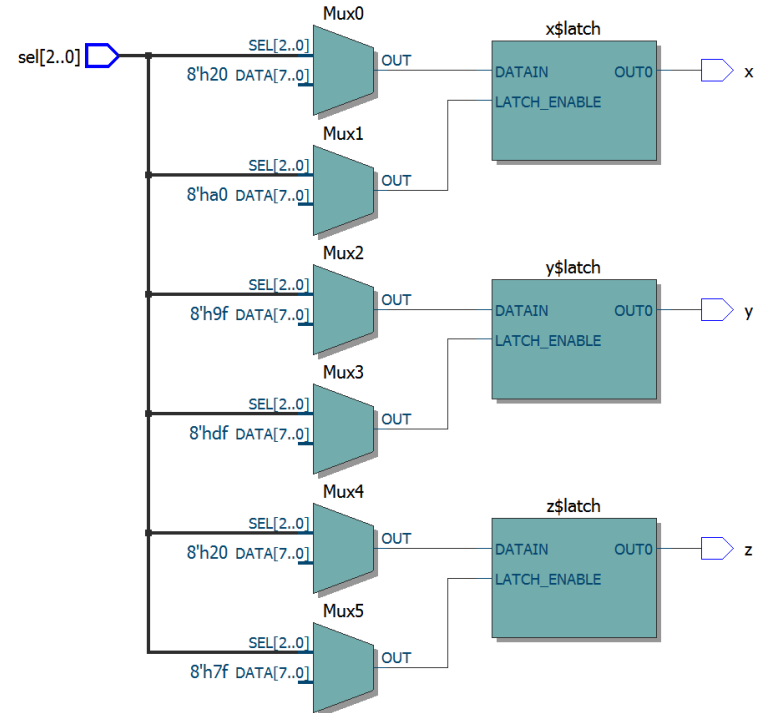


Нежелательные защелки в операторе Case

□ Условия, в которых не определены выходы

```
1 module ex3 (sel, x, y, z);
2   input [2:0] sel;
3   output reg x, y, z;
4
5   always @*
6   case (sel)
7     3'b000 : begin x = 1'b0; y = 1'b1; end
8     3'b001 : begin z = 1'b0; y = 1'b0; end
9     3'b010 : begin x = 1'b1; z = 1'b1; end
10    default : begin z = 1'b0; y = 1'b1; end
11  endcase
12
13 endmodule
```

- 3 выхода: x, y, z
- На не полностью определенном выходе формируется триггер-защелка
- Триггеры-защелки будут сформированы на ВСЕХ трех выходах.



Исправление

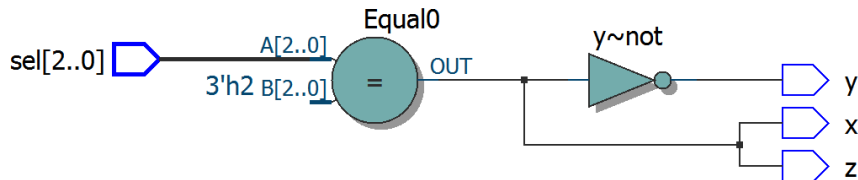
❑ Доопределение выходов

```
1  module ex3_ (sel,  x, y, z);
2  input  [2:0] sel;
3  output reg x, y, z;
4
5  always @*
6  begin
7      x = 1'b0;
8      y = 1'b0;
9      z = 1'b0;
10 case (sel)
11     3'b000 : y= 1'b1;
12     3'b010 : begin x =1'b1; z= 1'b1; end
13     default: y= 1'b1;
14 endcase
15 end
16 endmodule
```

Инициализация

Для исключения триггеров-
защелок и полного
определения всех выходов:

- Используйте инициализацию сигналов перед оператором case (в операторе будут описаны только отличные от базовых значения сигналов)



Триггеры-защелки в схемах с регистрами

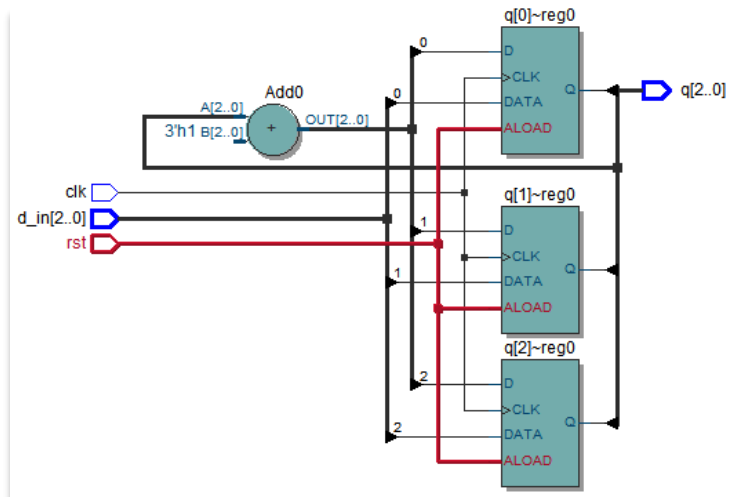
❑ Асинхронная запись данных в регистр

```

module latch_in_ff(
input      clk,
input      [2:0] d_in,
input      rst,
output reg [2:0] q    );

always @(posedge clk, posedge rst)
    if (rst)
        q<=d_in;
    else
        q<=q+3'h1;
endmodule

```

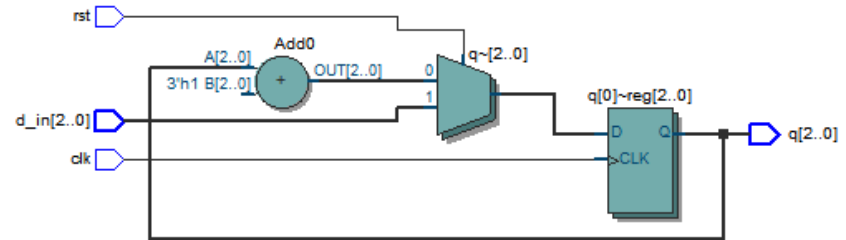


13004 Presettable and clearable registers converted to equivalent circuits with latches.

Как избежать Latch в схемах с регистрами

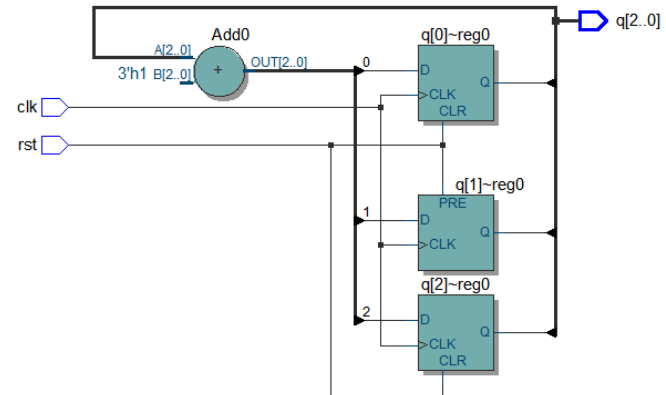
❑ Синхронная запись данных в регистр

```
module latch_in_ff(  
    input      clk,  
    input [2:0] d_in,  
    input      rst,  
    output reg [2:0] q );  
  
    always @(posedge clk)  
        if (rst)  
            q<=d_in;  
        else  
            q<=q+3'h1;  
        endmodule
```



❑ Асинхронная запись фиксированных данных

```
module latch_in_ff(  
    input      clk,  
    input      rst,  
    output reg [2:0] q );  
  
    always @(posedge clk, posedge rst)  
        if (rst)  
            q<=3'h2;  
        else  
            q<=q+3'h1;  
        endmodule
```





Функции и задачи

Функция и задача Functions and Tasks

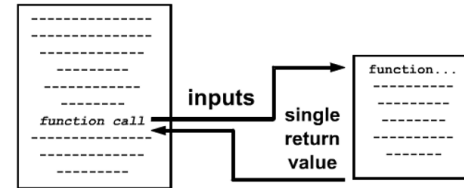
❑ Для чего используются

- ✓ Замена повторяющегося кода
- ✓ Улучшение читаемости кода

❑ Функция

- ✓ Возвращает одно значение, полученное на основе входных данных
- ✓ Реализуется как комбинационная схема

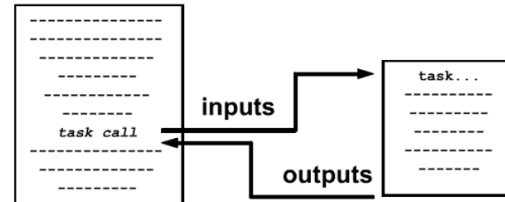
```
assign mult_out = mult (ina, inb);
```



❑ Задача

- ✓ Может быть реализована как комбинационная или регистровая схема

```
stm_out (nxt, first, sel, filter);
```



Сравнение функций и задач

Функции

- ❑ Всегда выполняются в 0 момент времени
 - ✓ Выполнение не может быть приостановлено
 - ✓ Не могут содержать delay, event, или timing control операторы
- ❑ Должны иметь хотя бы один вход
 - ✓ Вход нельзя изменять в функции
- ❑ Аргументы не могут быть output и inout
- ❑ Всегда возвращает одно значение
- ❑ Может вызывать другую функцию но не задачу

Задачи

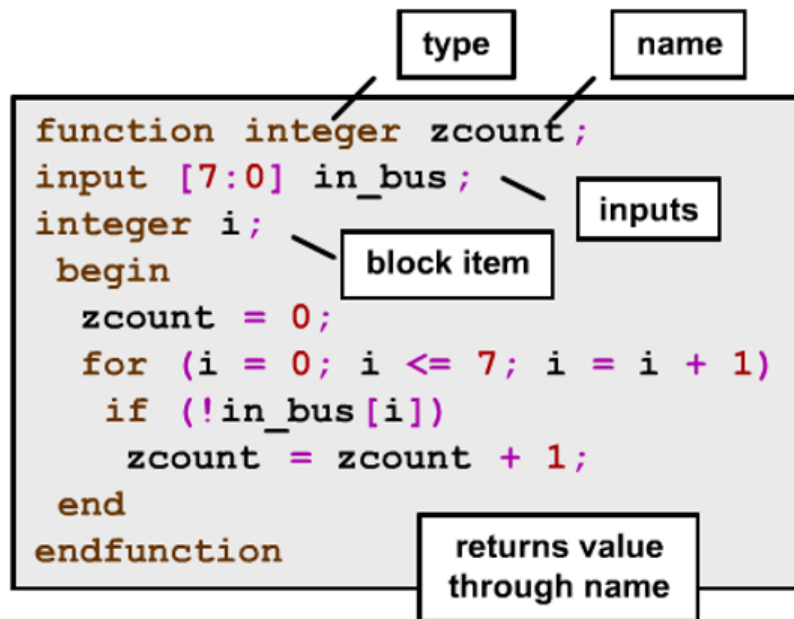
- ❑ Могут выполняться в любой момент времени моделирования
 - ✓ Могут содержать delay, event или timing control операторы
- ❑ Могут не иметь input, output, inout, или иметь более чем один
- ❑ Могут модифицировать значения
- ❑ Могут вызывать функции и задачи

Объявление функции (старый формат)

```
function < range or type > func_name;  
    // argument ports  
    // declarations  
    // statements  
endfunction
```

Syntax:

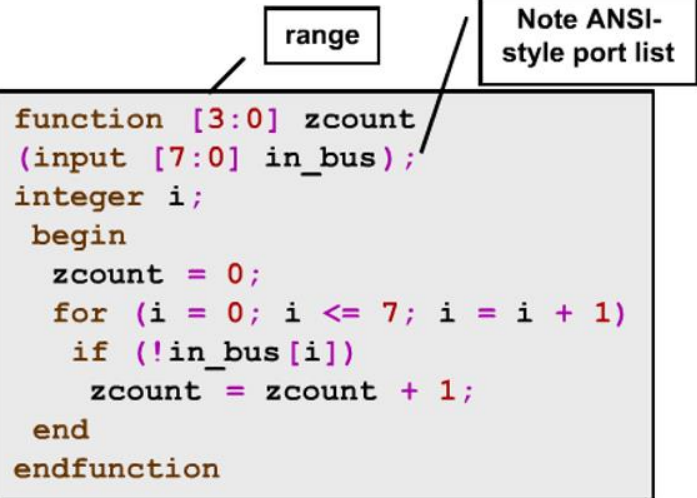
```
function [automatic]  
[signed] [range_or_type]  
function_identifier;  
    tf_input_declaration;  
    {tf_input_declaration;}  
    {block_item_declaration}  
    function_statement  
endfunction
```



Объявление функции (ANSI формат)

```
function < range or type > func_name (argument ports);  
    // declarations  
    // statements  
endfunction
```

```
function [automatic]  
[signed] [range_or_type]  
function_identifier  
(function_port_list);  
{block_item_declaration}  
function_statement  
endfunction
```



The diagram illustrates the ANSI format for a function declaration and its implementation. A box labeled "range" points to the `[3:0]` in the function signature. Another box labeled "Note ANSI-style port list" points to the `(input [7:0] in_bus);` line. The implementation is shown in a separate box below the declaration.

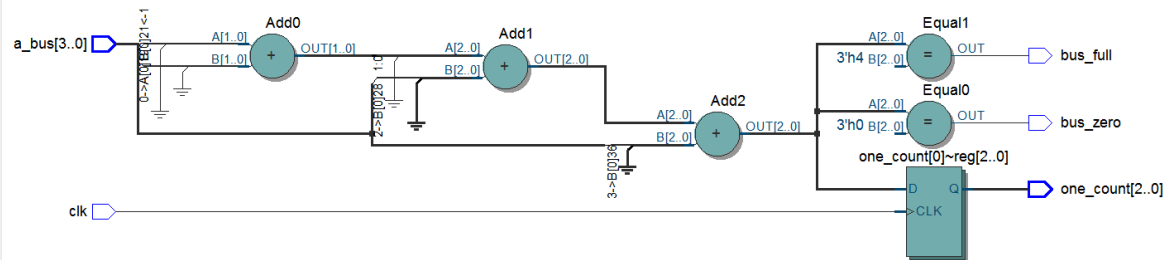
```
function [3:0] zcount  
(input [7:0] in_bus);  
integer i;  
begin  
    zcount = 0;  
    for (i = 0; i <= 7; i = i + 1)  
        if (!in_bus[i])  
            zcount = zcount + 1;  
        end  
    endfunction
```

Пример функции

```
function      [2:0] bus_cnt;  
input        [3:0] in_bus;  
  
integer i;  
  
begin  
    bus_cnt = 0;  
    for (i=0; i<=3; i=i+1)  
        bus_cnt = bus_cnt+in_bus[i];  
end;  
endfunction
```



```
module zfunc (a_bus, clk, one_count,  
             bus_zero, bus_full );  
input [3:0] a_bus;  
input clk;  
output reg [2:0] one_count;  
output reg bus_full;  
output bus_zero;  
  
//function declaration  
assign bus_zero = bus_cnt(a_bus)==3'd0;  
always @(a_bus)  
    bus_full = (bus_cnt(a_bus)==3'd4);  
  
always @(posedge clk)  
    one_count = bus_cnt(a_bus);  
  
endmodule
```



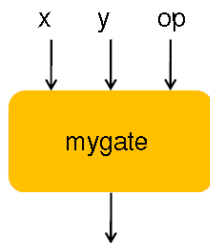
Пример функции

- Использование в procedural assignments (initial; always)

```
...  
reg [7:0] dout;  
...  
  
function [7:0] shift_right;  
input [7:0] data;  
shift_right = {1'b0, data[7:1]};  
endfunction  
  
...  
always@*  
dout = shit_right (7'b10001000);  
// dout = 7'b01000100
```

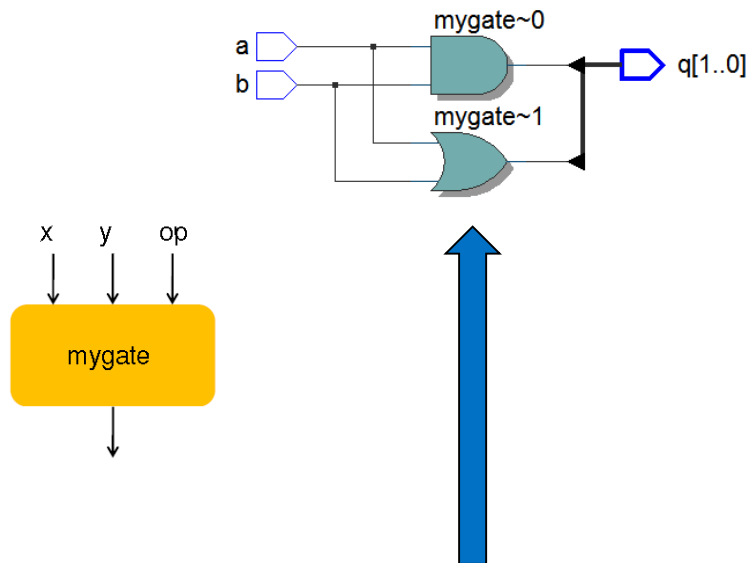
Пример

```
module andorgate (  
  output [1:0] q,  
  input  a, b);  
  function mygate;  
    input x, y, op;  
    case (op)  
      1'b0: mygate = x & y;  
      1'b1: mygate = x | y;  
      default: mygate = 1'b0;  
    endcase  
  endfunction  
  
  assign q[0] = mygate(a, b, 0); // and gate  
  assign q[1] = mygate(a, b, 1); // or gate  
  
endmodule
```



Пример

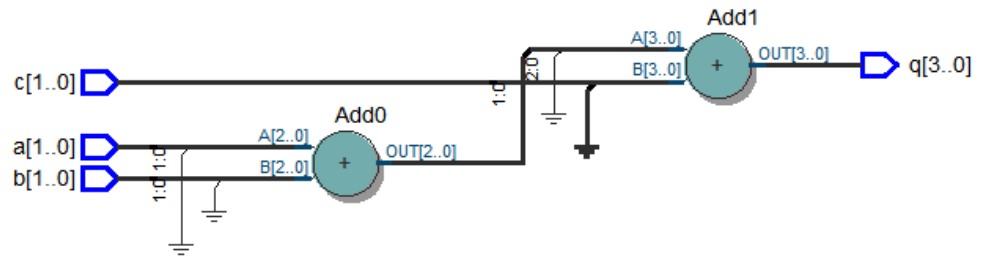
```
module andorgate (  
  output [1:0] q,  
  input a, b);  
  function mygate;  
    input x, y, op;  
    case (op)  
      1'b0: mygate = x & y;  
      1'b1: mygate = x | y;  
      default: mygate = 1'b0;  
    endcase  
  endfunction  
  
  assign q[0] = mygate(a, b, 0); // and gate  
  assign q[1] = mygate(a, b, 1); // or gate  
  
endmodule
```



➡ Реализация «по вызову функции»
а не по объявлению функции

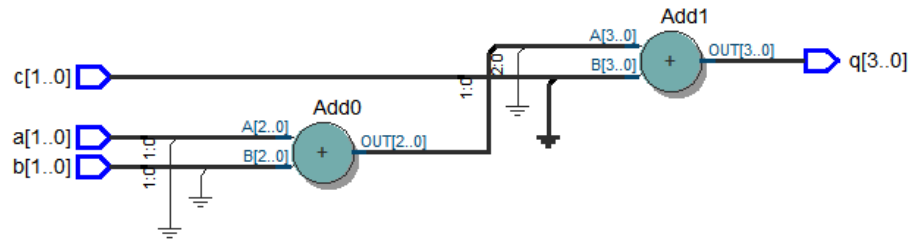
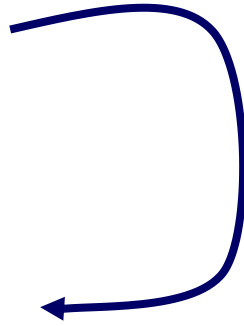
Функция может иметь локальные переменные

```
module addit(  
    output [3:0] q,  
    input  [1:0] a, b, c;  
    function [3:0] myadder;  
        input [1:0] x, y, z;  
        reg [2:0] w; // local variable  
        begin  
            w          = x + y;  
            myadder = w + z;  
        end  
    endfunction  
  
    assign q = myadder(a, b, c);  
  
endmodule
```



Функции (доступ к переменным модуля)

```
module addit(  
  output [3:0] q,  
  input  [1:0] a, b, c;  
  function [3:0] myadder;  
    input [1:0] x;  
    reg [2:0] w;  
    begin  
      w = x + a;  
      myadder = w + c;  
    end  
  endfunction  
  
  assign q = myadder( b );  
  
endmodule
```



Константные функции

- ❑ Не требуют аппаратных ресурсов, вычисляются на этапе компиляции

```
module ram_const_func #(parameter D_WIDTH = 8, RAM_DEPTH = 2048)
```

```
( output reg [D_WIDTH-1:0] q,
```

```
  input [D_WIDTH-1:0] d,
```

```
  input [clogb2(RAM_DEPTH)-1:0] addr,
```

```
  input we, clk);
```

```
function integer clogb2;
```

```
  input integer mem_depth;
```

```
  begin for (clogb2 = 0; mem_depth > 0; clogb2 = clogb2 + 1)
```

```
    mem_depth = mem_depth >> 1; end
```

```
endfunction
```

```
  reg [7:0] mem [0:RAM_DEPTH];
```

```
always @(posedge clk) begin
```

```
  if (we)    mem[addr] <= d;
```

```
  q <= mem[addr]; end
```

```
endmodule
```

*Используется для
определения
разрядности
адреса*

Пример: Функция Gray to Bin преобразования

❑ Помните Gray Code ?

- ✓ Код, в котором 1 bit изменяется в последовательных кодовых комбинациях

000

001

011

010

110

111

101

100

❑ Как построить функцию для преобразования?

Пример: Функция Gray to Bin преобразования

□ Таблица:

Gray	Bin
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

Пример: Функция Gray to Bin преобразования

□ Разряд 3:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

bit3 = gray3 = 1

Пример: Функция Gray to Bin преобразования

- Разряды 3 и 2:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

$$\text{bit3} = \text{gray3} = 1$$

$$\text{bit2} = \text{gray3} \text{ xor } \text{gray2} = 1 \text{ xor } 1 = 0$$

Пример: Функция Gray to Bin преобразования

□ Разряды 3 2 1:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

$$\text{bit3} = \text{gray3} = 1$$

$$\text{bit2} = \text{gray3} \text{ xor } \text{gray2} = 1 \text{ xor } 1 = 0$$

$$\text{bit1} = \text{gray3} \text{ xor } \text{gray2} \text{ xor } \text{gray1} = 1 \text{ xor } 1 \text{ xor } 0 = 0$$

Пример: Функция Gray to Bin преобразования

❑ Функция для 3-х разрядов

```
function [2:0] gray_to_bin;  
input [2:0] a;  
reg [2:0] q;  
    begin  
        q[2] = a[2];  
        q[1] = a[1] ^ a[2];  
        q[0] = a[0] ^ a[1] ^ a[2];  
        gray_to_bin = q;  
    end  
endfunction
```

bit3 = gray3 = 1

bit2 = gray3 xor gray2

bit1 = gray3 xor gray2 xor gray1

❑ Параметризированная версия функции с использованием цикла

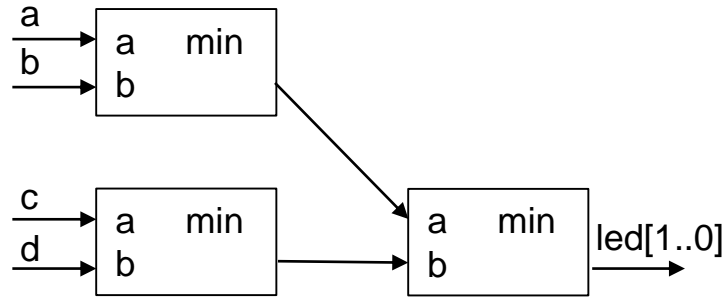
```
function [W:0] gray_to_bin;  
input [W:0] a;  
reg [W:0] q;  
    integer i;  
    begin  
        for (i=0; i<=W; i = i + 1)  
            q[i] = ^(a >> i);  
        gray_to_bin = q;  
    end  
endfunction
```

Объявление задачи

```
task <task_name>(<arg_decls>;  
    // Optional Block Item Declarations, e.g Local Variables  
begin  
    // Statements  
end  
endtask
```

Пример использования задачи

- Необходимо разработать модуль поиска минимума



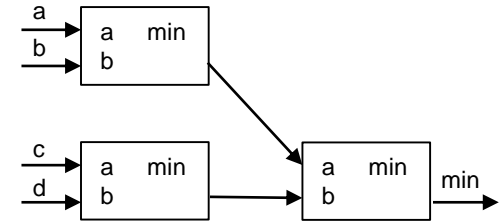
Пример (часть 1)

```
module task_1 (a,b,c,d,clk,arst,min);  
    input  [3:0] a, b, c, d;  
    input          clk, arst;  
    output reg [3:0] min;  
  
    task min_search;  
        output [3:0] minimum;  
        input  [3:0] in1, in2;  
        begin  
            minimum = (in1>in2)?in2:in1;  
        end  
    endtask
```

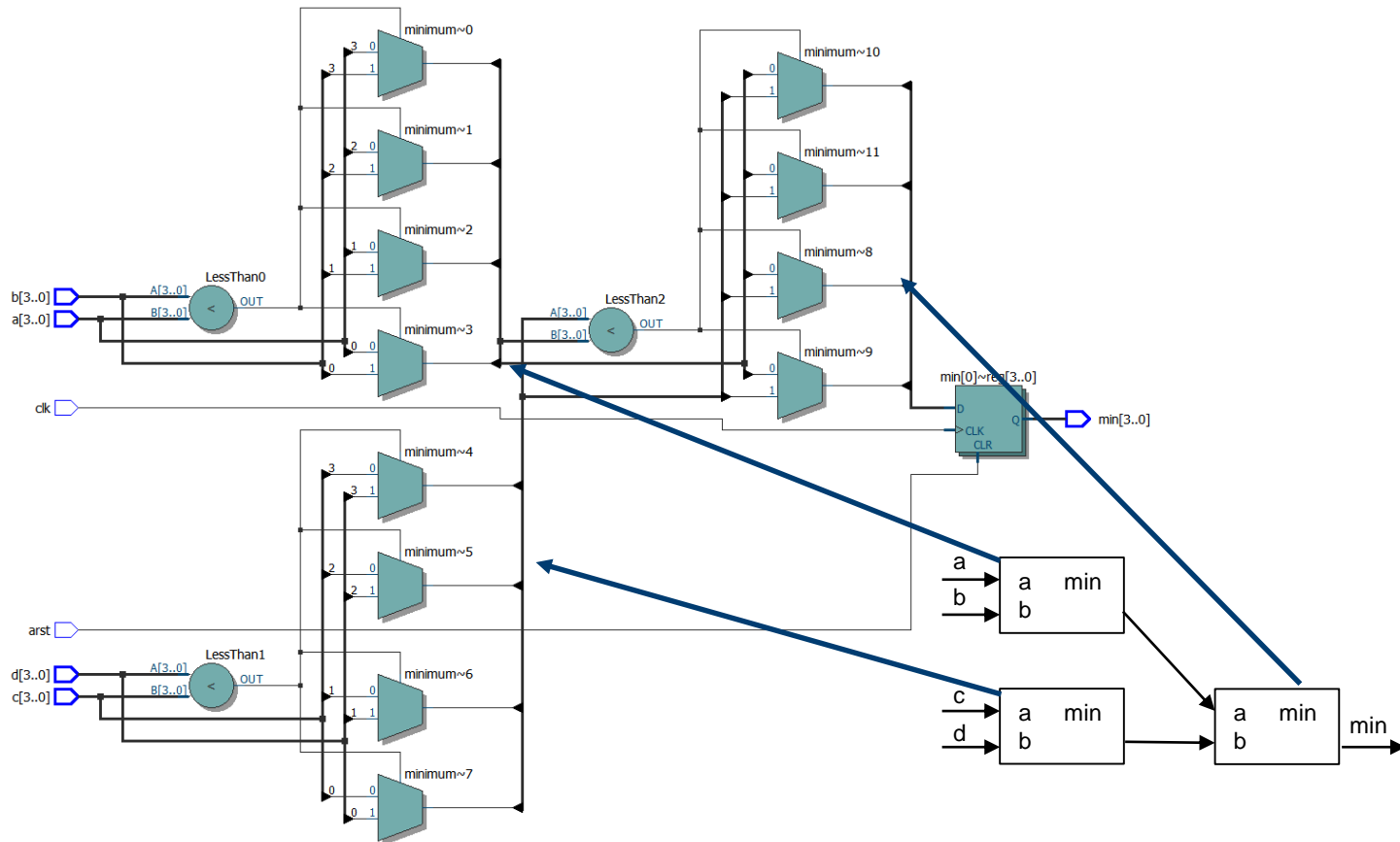
Пример (часть 1)

```
reg [3:0] min_one, min_two;

always @(posedge clk, posedge arst)
    if (arst)
        begin
            min_one = 4'h0;
            min_two = 4'h0;
            min      = 4'h0;
        end
    else
        begin
            min_search(min_one, a, b);
            min_search(min_two, c, d);
            min_search(min, min_one, min_two);
        end
endmodule
```



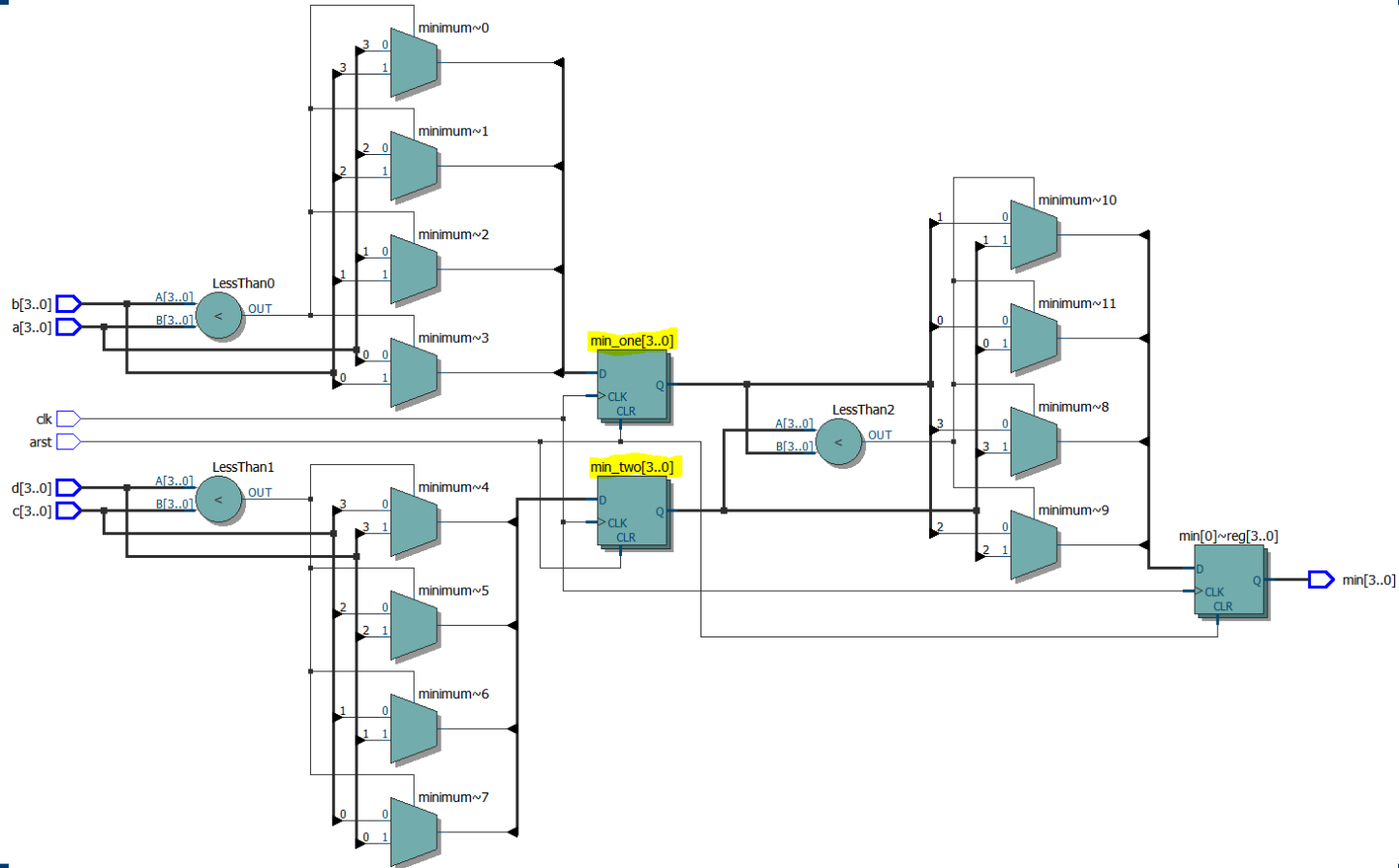
Результат синтеза



Что измениться в результатах синтеза?

```
module task_1 (a,b,c,d,clk,arst,min);  
    input  [3:0] a, b, c, d;  
    input          clk, arst;  
    output reg [3:0] min;  
  
    task min_search;  
        output [3:0] minimum;  
        input  [3:0] in1, in2;  
        begin  
            minimum <= (in1>in2)?in2:in1;  
        end  
    endtask
```


Результат синтеза

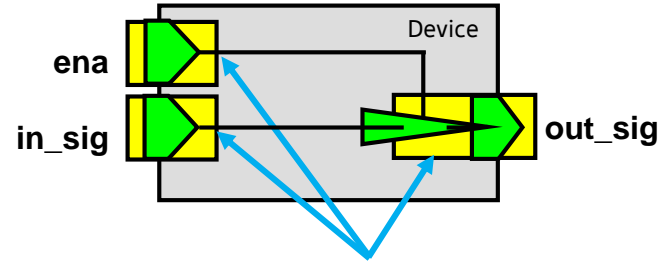
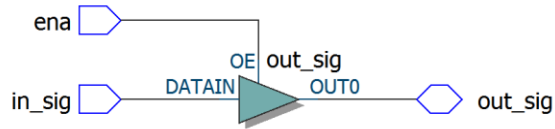




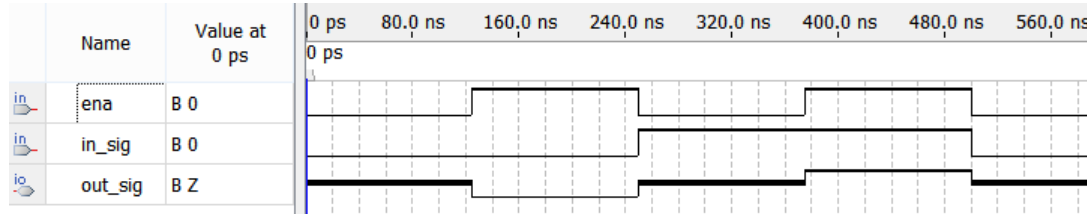
Выходы с Z состоянием

Использование выхода с Z состоянием

```
1 module tri_ex1 (ena, in_sig, out_sig);  
2   inout out_sig;  
3   input ena, in_sig;  
4  
5   assign out_sig = ( ena)? in_sig : 1'bZ;  
6  
7 endmodule
```

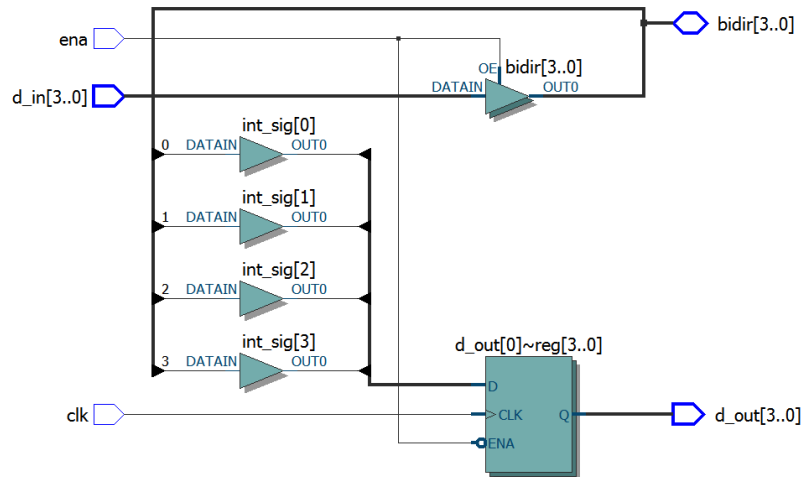


Элементы ввода/вывода



Двунаправленный вывод

```
1 module bidir_io
2   #(parameter WIDTH=4)
3   (input ena, clk, input [WIDTH-1:0] d_in, inout [WIDTH-1:0] bidir, output reg [WIDTH-1:0] d_out);
4
5   wire [WIDTH-1:0] int_sig;
6
7   assign bidir = (ena) ? d_in : {WIDTH{1'bz}};
8   assign int_sig = bidir;
9
10  always @(posedge clk)
11    if (ena==0) d_out <= int_sig;
12
13 endmodule
```



Двухнаправленный вывод - моделирование

