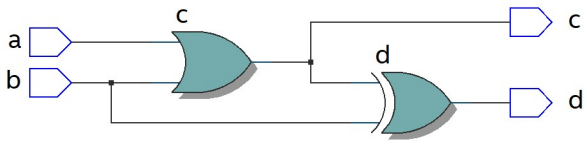
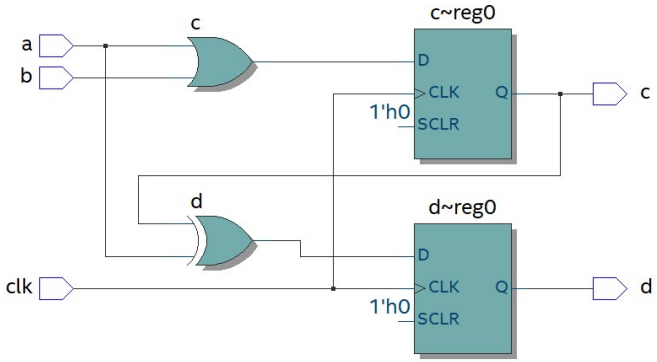
Continuous Assignment

```
wire a, b, c, d;
assign c = a | b;
assign d = c ^ b;
```



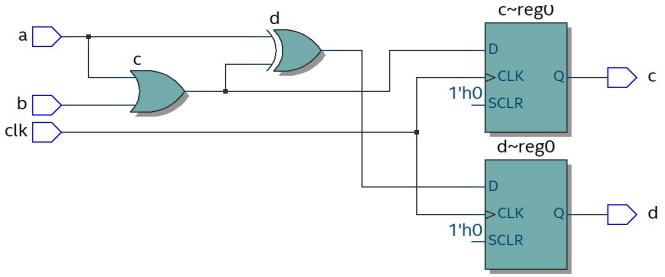
Blocking Assignment

```
wire a, b;
reg c, d;
always @(posedge clk) begin
    c <= a | b;
    d <= c ^ a;
end
```



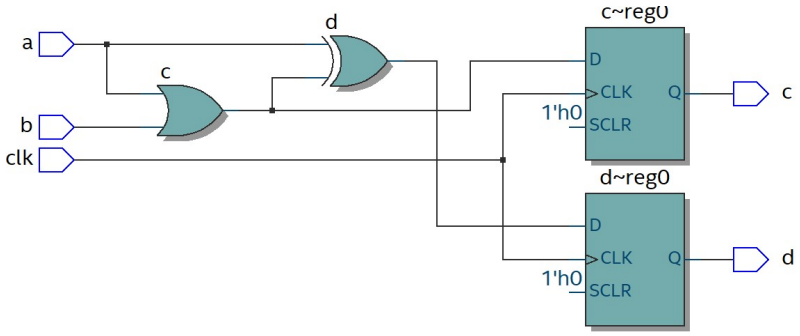
Non-blocking Assignment

```
wire a, b;
reg c, d;
always @(posedge clk) begin
    c = a | b;
    d = c ^ a;
end
```

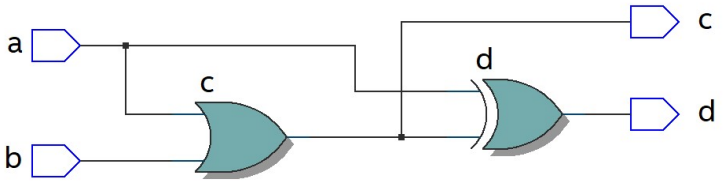


Always @(*) block

```
wire a, b;
reg c, d;
always @(posedge clk) begin
    c = a | b;
    d = c ^ a;
end
```

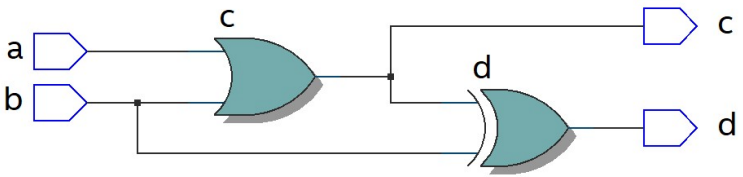



```
wire a, b;
reg c, d;
always @(*) begin
    c = a | b;
    d = c ^ a;
end
```

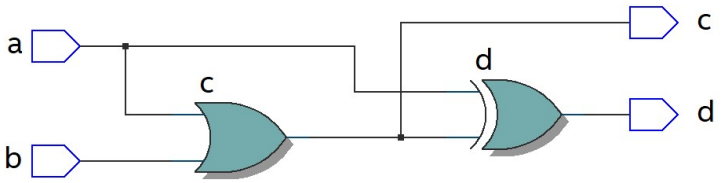


Always @(*) block

```
wire a, b, c, d;
assign c = a | b;
assign d = c ^ b;
```



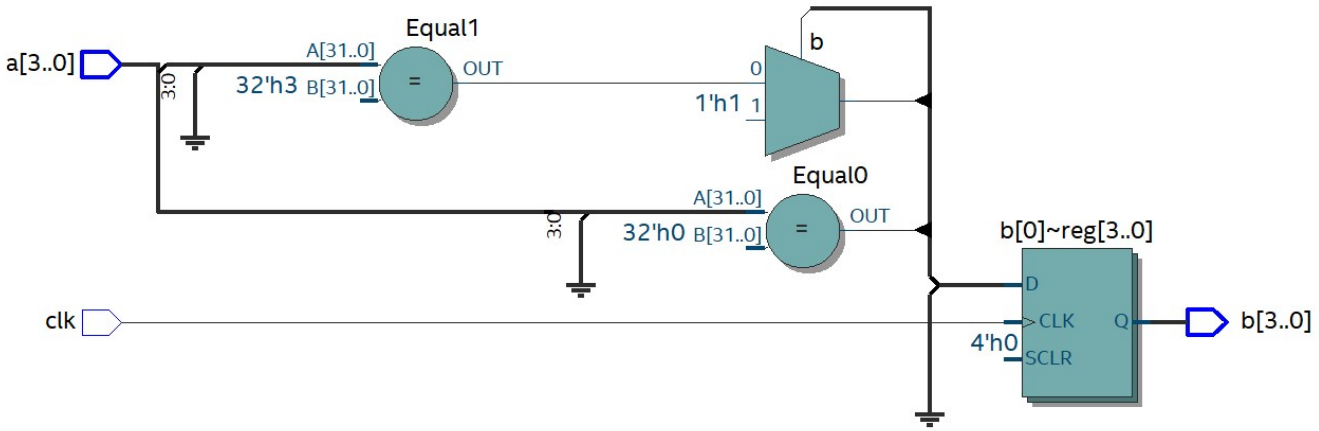

```
wire a, b;
reg c, d;
always @(*) begin
    c = a | b;
    d = c ^ a;
end
```



Зачем тогда нужна конструкция always `always @(*)` ?

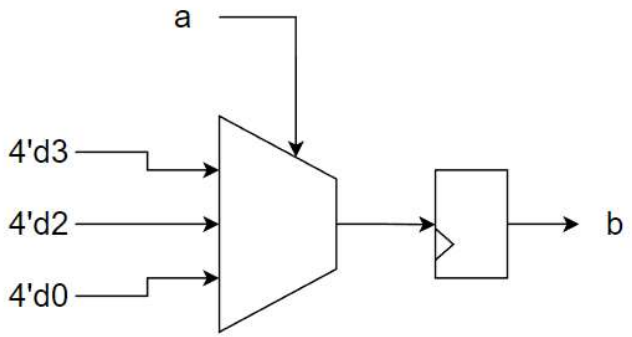
Условия (if else)

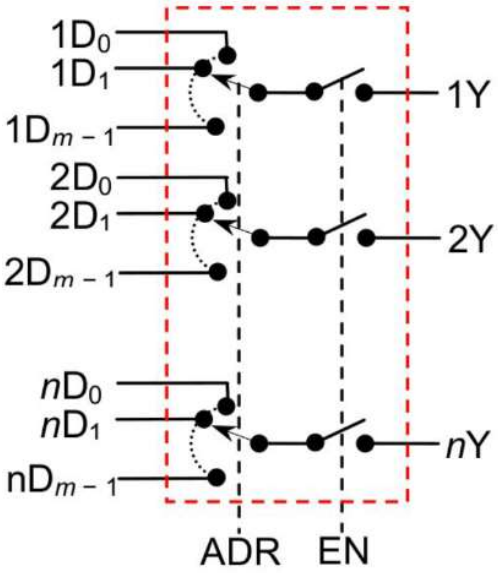
```
wire [3: 0] a;
reg [3: 0] b;
always @(posedge clk) begin
    if (a==4’d0) begin
        b <= 4’d3;
    end else if (a==4’d3) begin
        b <= 4’d2;
    end else begin
        b <= 4’d0;
    end
end
```



Мультиплексок (MUX)

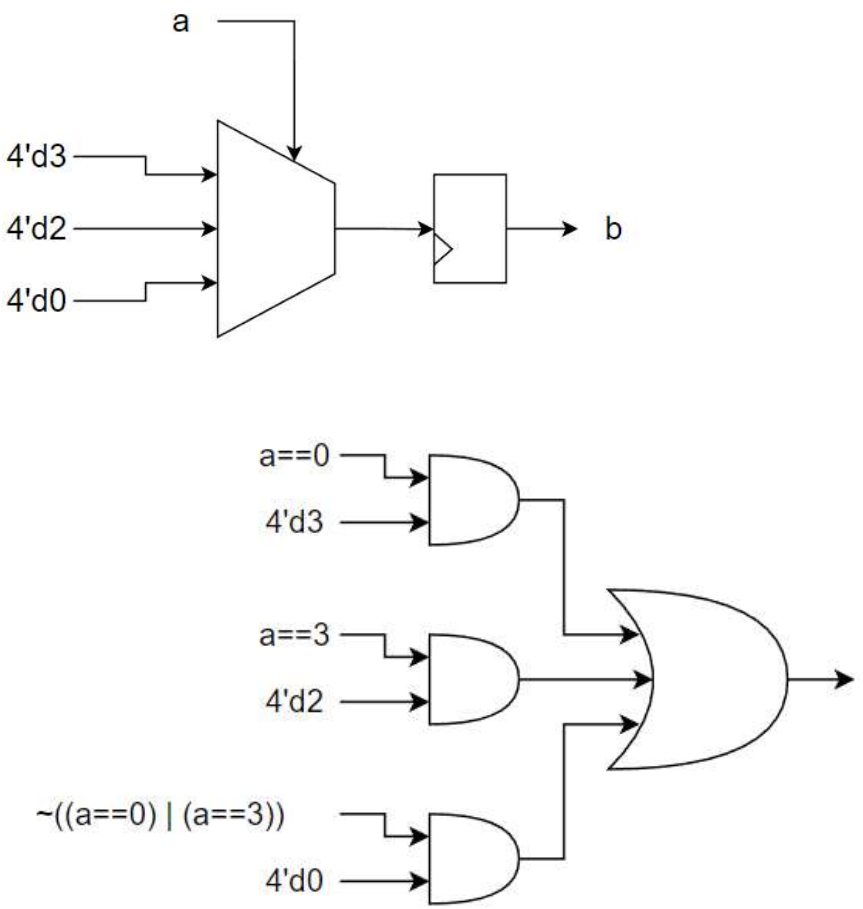
```
wire [3: 0] a;
reg [3: 0] b;
always @(posedge clk) begin
    if (a==4’d0) begin
        b <= 4’d3;
    end else if (a==4’d3) begin
        b <= 4’d2;
    end else begin
        b <= 4’d0;
    end
end
```





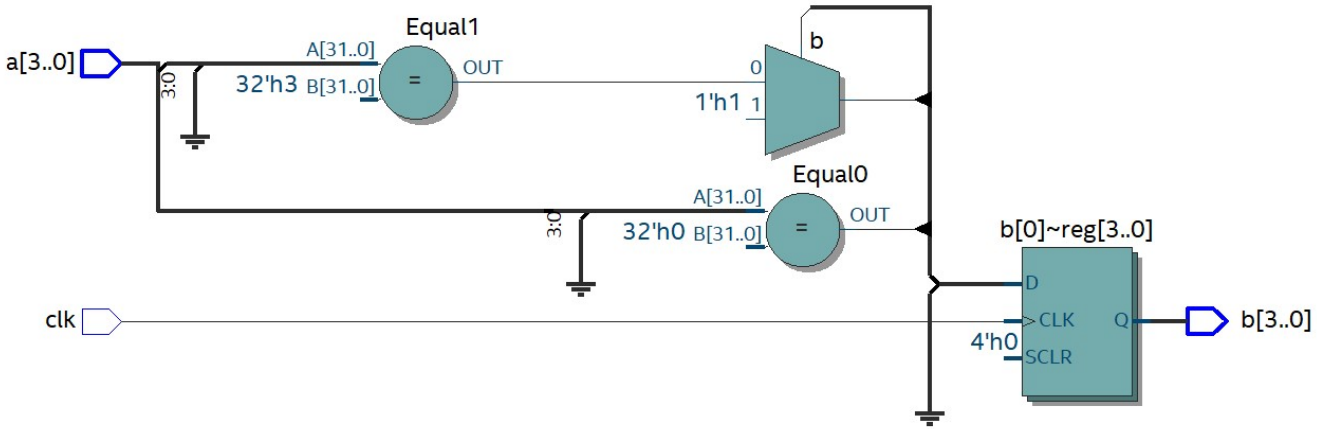
Мультиплексок (MUX)

```
wire [3: 0] a;
reg [3: 0] b;
always @(posedge clk) begin
    if (a==4’d0) begin
        b <= 4’d3;
    end else if (a==4’d3) begin
        b <= 4’d2;
    end else begin
        b <= 4’d0;
    end
end
```



Условия (case)

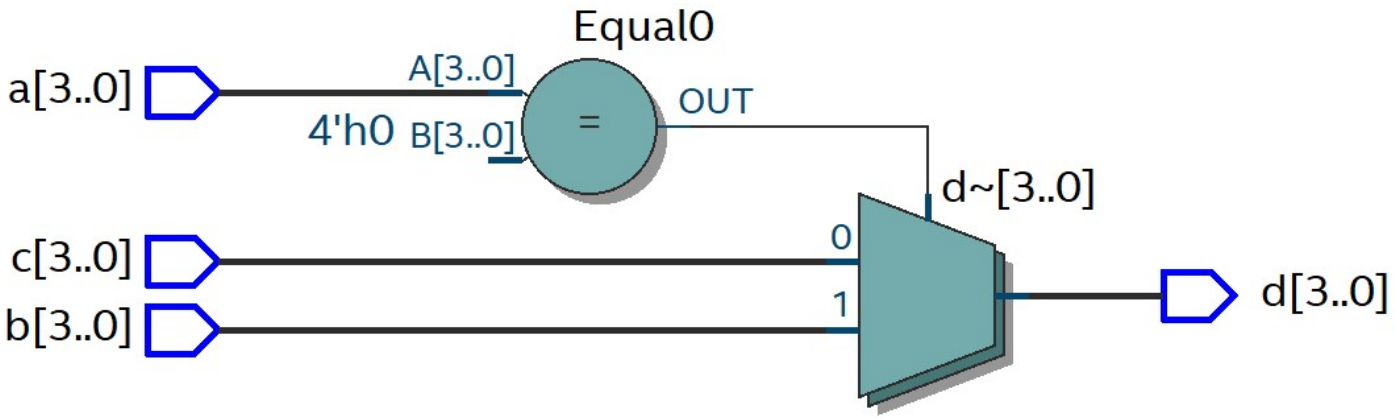
```
wire [3: 0] a;
reg [3: 0] b;
always @(posedge clk) begin
    case (a)
        4’d0:
            b <= 4’d3;
        4’d3:
            b <= 4’d2;
        default:
            b <= 4’d0;
    endcase
end
```



Тернарный оператор

```
assign cond ? <result if true> : <result if false>
```

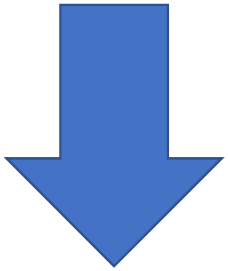
```
wire [3: 0] a, b, c, d;
assign d = (a==4’d0) ? b : c;
```



Тернарный оператор

```
assign cond ? <result if true> : <result if false>
```

```
wire [3: 0] a, b, c, d;
assign d = (a==4’d0) ? b : c;
```

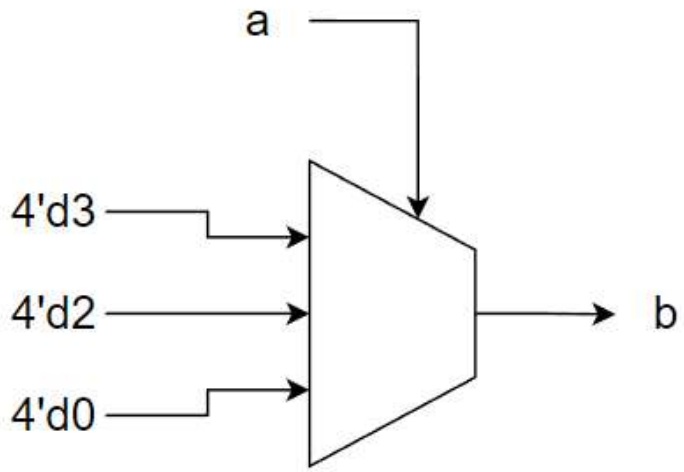


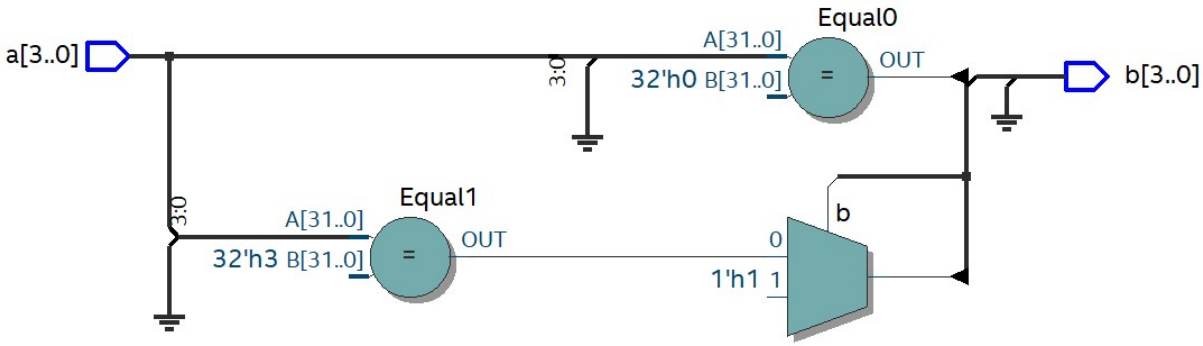
Что если условий много?

```
assign b = (a==4’d0 ? 4’d3 :(a==4’d3 ? 4’d2 : 4’d0));
```

Always @(*) block

```
wire [3: 0] a;
reg [3: 0] b;
always @(*) begin
    if (a==4’d0) begin
        b <= 4’d3;
    end else if (a==4’d3) begin
        b <= 4’d2;
    end else begin
        b <= 4’d0;
    end
end
```

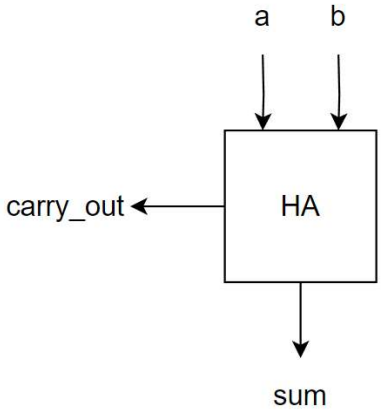




Half adder

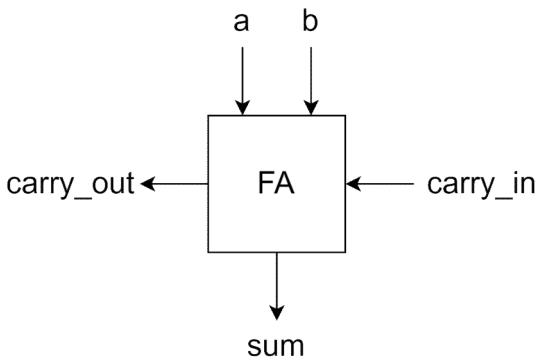
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

AND XOR

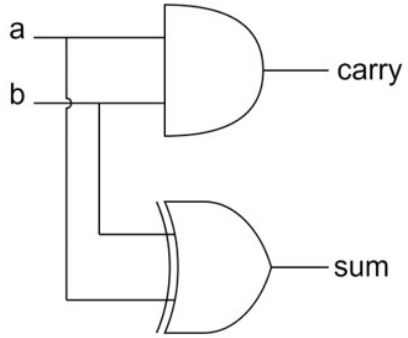


Full adder

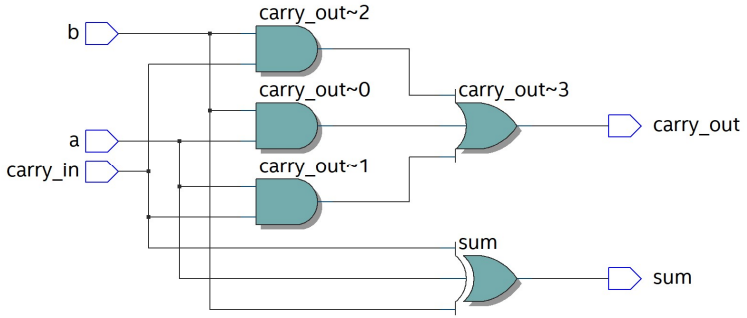
a	b	carry_in	carry_out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



half adder

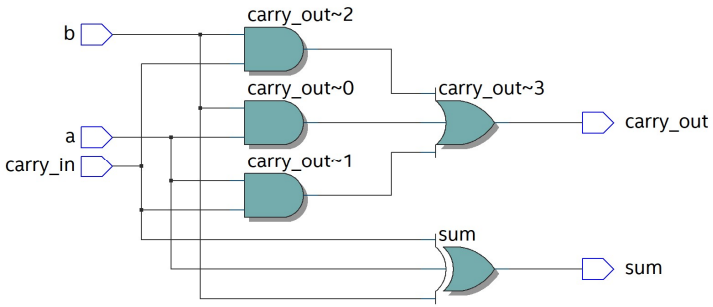


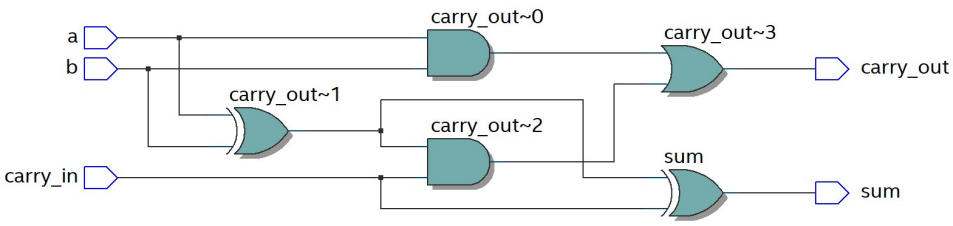
full adder



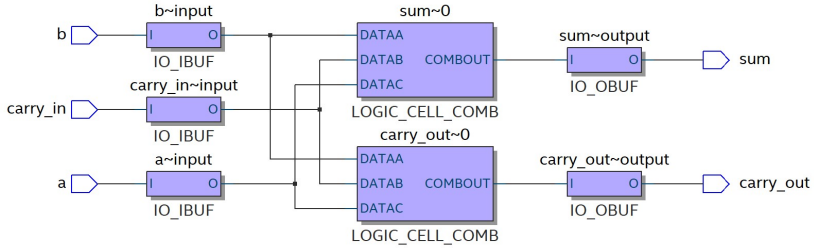
Реализация в FPGA

Различные схемы из логических элемментов



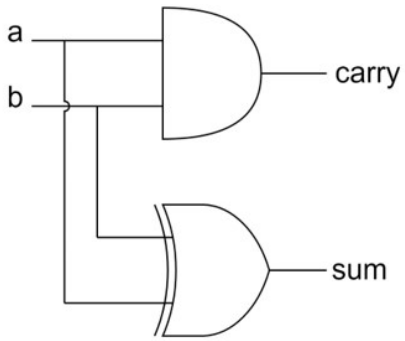


Реально т.к. Сложение – одна из наиболее распространенных операций, поэтому блоки FPGA специально оптимизированы под такую задачу

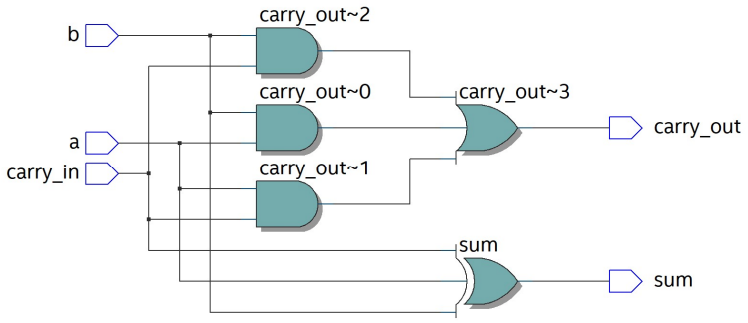


Модули FA и HA

```
module HA (
    input  a,
    input  b,
    output sum,
    output carry_out
);
    assign carry_out = a & b;
    sum    = a ^ b;
endmodule
```

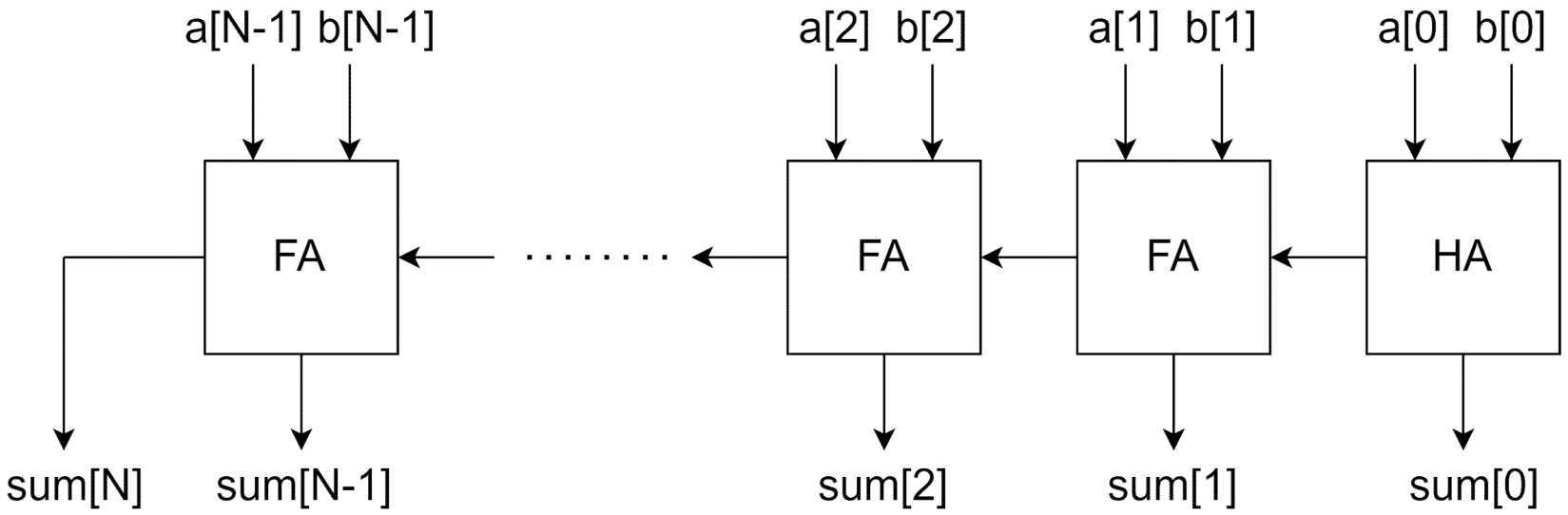



```
module FA(
    input a,
    input b,
    input carry_in,
    output carry_out,
    output sum
);
    assign sum = a ^ b ^ carry_in;
    assign carry_out = a&b | a&carry_in | b&carry_in;
endmodule
```



Сложение.

Схема выполнения беззнакового сложения двух N-разрядных чисел

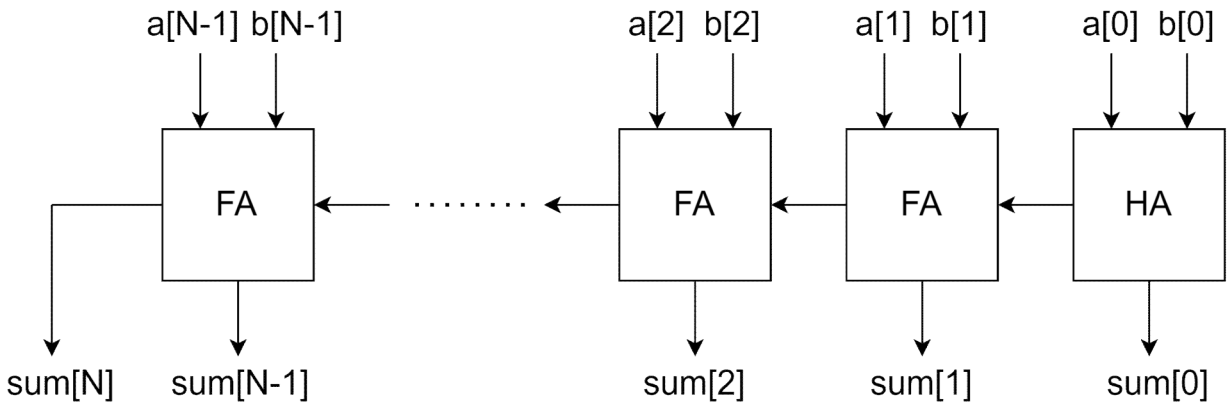


Восьмибитный сумматор

```
module adder8 (
    input  [7:0]a,
    input  [7:0]b,
    output [8:0]sum
);
    wire [7: 0] c;
    HA ha0( .a(a[0]), .b(b[0]), .sum(sum[0]), .carry_out(c[0]));
    FA fa1( .a(a[1]), .b(b[1]), .carry_in(c[0]), .sum(sum[1]), .carry_out(c[1]));
     …
    FA fa7( .a(a[7]), .b(b[7]), .carry_in(c[6]), .sum(sum[2]), .carry_out(c[7]));
     assign sum[8] = c[7];
endmodule
```

Но обычно все сильно проще.

```
module adder8 (
    input  [7:0] a,
    input  [7:0] b,
    output [8:0] sum
);
    assign sum = a + b;
endmodule
```



Параметры

- Параметры позволяют переиспользовать код для схожих задач.
- Значения параметров должны быть известны на момент компиляции
- Соответственно для их инициализации можно использовать только выражения с литералами и другими параметрами

```
module adder #(
    parameter W_ADD = 8
) (
     input  [W_ADD-1:0] a,
     input  [W_ADD-1:0] b,
     output [W_ADD:0] sum
);
    assign sum = a + b;
Endmodule


// create instance
adder #(W_ADD=10) add10 (
    .a(a),
    .b(b),
    .sum(sum)
)
```

Параметры (Два способа обьявления)

```
module module_name #(
    [parameter_list]
) (
    [port_list]
);
    …
endmodule

// example
module adder #(
    parameter W_ADD = 8
) (
    input  [W_ADD-1:0] a,
    input  [W_ADD-1:0] b,
    output [W_ADD:0] sum
);
    assign sum = a + b;
endmodule
```

```
module module_name ([port_list]);
    [declare parameters];
    [declare ports];
    …
endmodule

// example
module adder(a, b, sum);

    parameter W_ADD = 8;

    input [W_ADD-1: 0] a, b;
    output [W_ADD: 0] sum;

    assign sum = a + b;

endmodule
```

Localparam

localparam – параметр, который нельзя задать при создании модуля. Используется для промежуточных вычислений с параметрами.

Parameter – обязан быть задан при создании (или будет использовано значение по умолчанию)

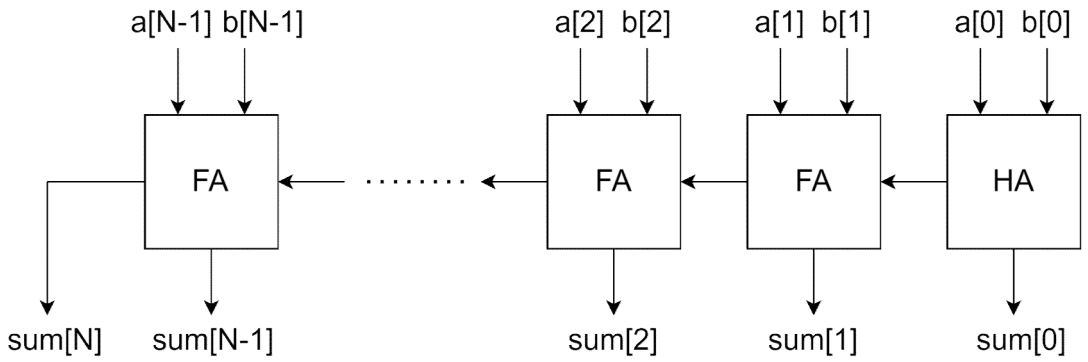
Localparam – нельзя задать при создании модуля.

```
module adder (a, b, sum);

    parameter W_ADD = 8;
    localparam W_SUM = W_ADD+1;

    input [W_ADD-1: 0] a, b;
    output [W_SUM-1: 0] sum;

    assign sum = a + b;

endmodule
```

```
// create instance
adder #(W_ADD=10) add10 (
    .a(a),
    .b(b),
    .sum(sum)
)
```

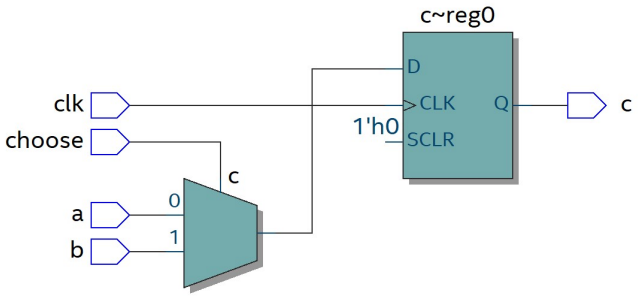
Generate Loop

```
module adder #(
     parameter W_ADD = 8
) (
     input  [W_ADD-1:0] a, b,
     output [W_ADD:0] sum
);
     wire [W_ADD-1: 0] c;
     HA ha0( .a(a[0]), .b(b[0]), .sum(sum[0]), .carry_out(c[0]));

     genvar i;
     generate
          for (i=1; i<W_ADD; i=i+1) begin
               FA fa(
                    .a(a[i]),
                    .b(b[i]),
                    .carry_in(c[i-1]),
                    .sum(sum[i]),
                    .carry_out(c[i])
               );
          end
     endgenerate

     assign sum[8] = c[7];
endmodule
```

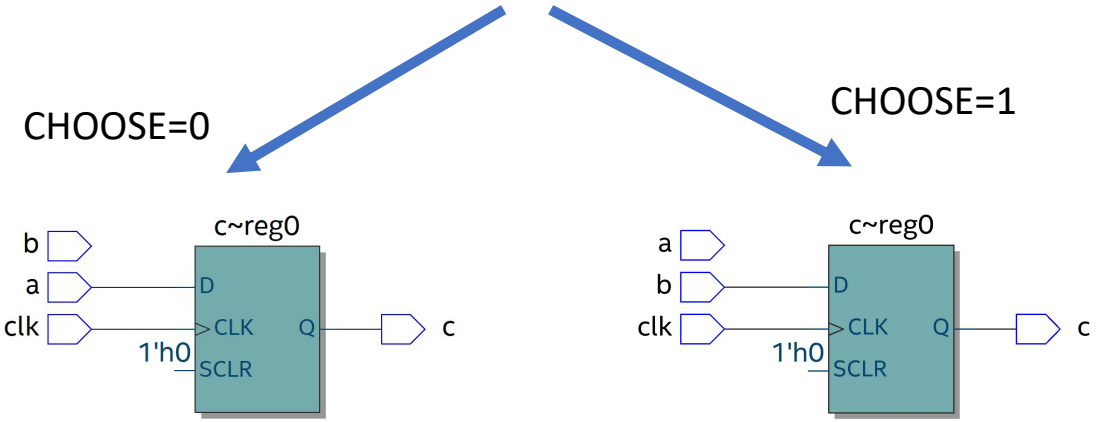


IF with parameter

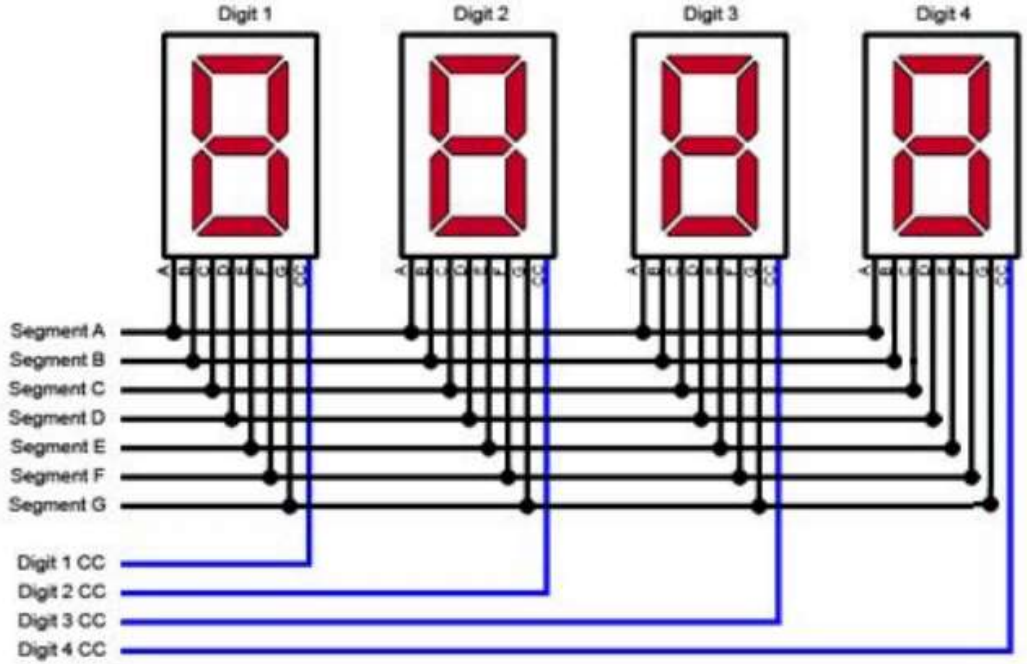
```
module (
    input clk,
    input a, b, choose,
    output reg c
);
    always @(posedge clk) begin
        if (choose==1'b0) begin
            c <= a;
        end else begin
            c <= b;
        end
    end
endmodule
```

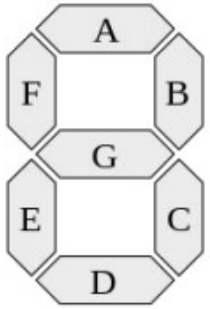


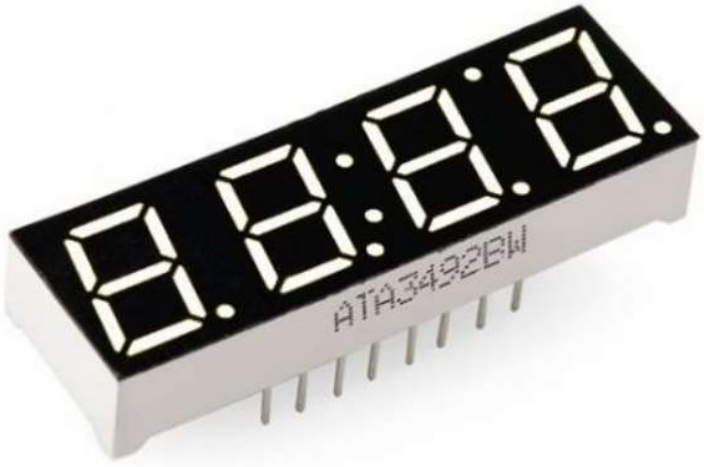

```
module #(
    parameter CHOOSE=0
) (
    input clk,
    input a, b,
    output reg c
);
    always @(posedge clk) begin
        if (CHOOSE==0) begin
            c <= a;
        end else begin
            c <= b;
        end
    end
endmodule
```



Семисегментный индикатор
(динамическая индикация)





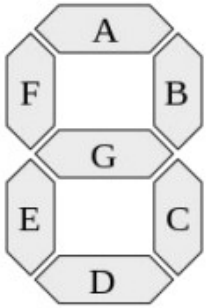


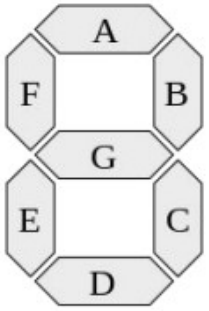
Делитель частоты

```
module clk_div(
    input clk,

    output clk2
);

reg [11:0]cnt = 0;

assign clk2 = cnt[11];

always @(posedge clk) begin
    cnt <= cnt + 12'b1;
end

endmodule
```

Семисегментный декодер

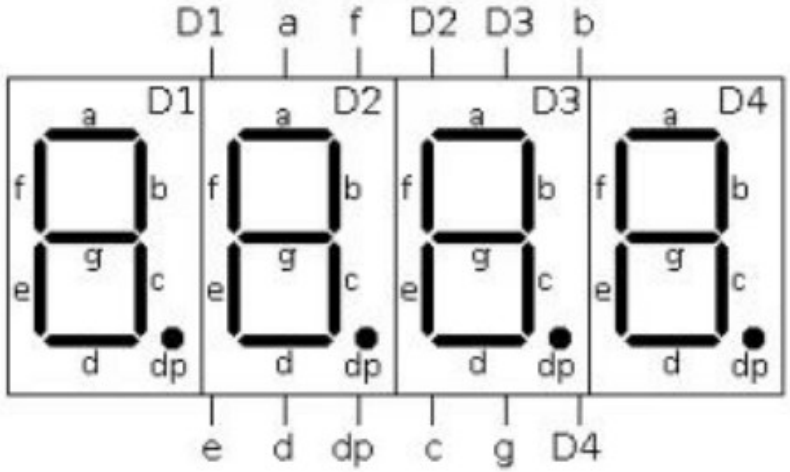
```
module bin_to_seg(
    input data,

    output reg [6:0]segments
);

always @(*) begin
    case (data)
        1'b0: segments = 7'b1111110;
        1'b1: segments = 7'b0110000;
    endcase
end

endmodule
```





Динамическая индикация

```
module bin_display(
   input clk, [3:0]data,

   output [3:0]anodes, [6:0]segments
);

reg [1:0]i = 0;
assign anodes = (4'b1 << i);

always @(posedge clk) begin
   i <= i + 2'b1;
end

wire b = data[i];
bin_to_seg bin_to_seg(.data(b), .segments(segments));

endmodule
```



Top.v

```
module top(
    input CLK,

    output DS_EN1, DS_EN2, DS_EN3, DS_EN4,
    output DS_A, DS_B, DS_C, DS_D, DS_E, DS_F, DS_G
);

wire [3:0]d = 4'b1011;
wire [3:0]anodes;
assign {DS_EN1, DS_EN2, DS_EN3, DS_EN4} = ~anodes;

wire [6:0]segments;
assign {DS_A, DS_B, DS_C, DS_D, DS_E, DS_F, DS_G} = segments;

clk_div clk_div(.clk(CLK), .clk2(clk2));

bin_display disp(.clk(clk2), .data(d), .anodes(anodes), .segments(segments));

endmodule
```