

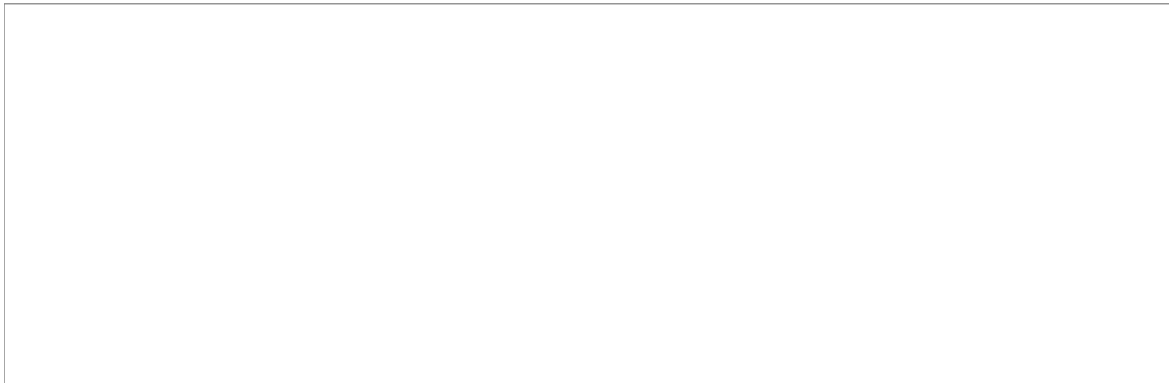
What is logic synthesis

Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation.

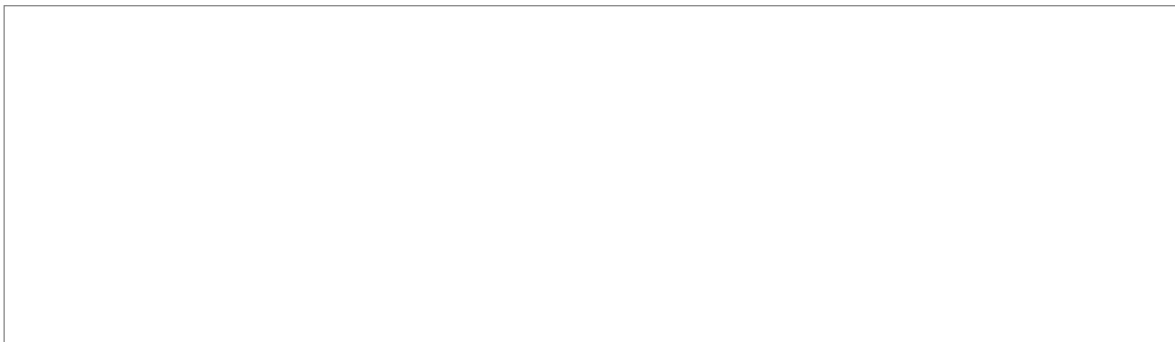
Logic synthesis uses standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, multiplexers, memory, and special flip-flops.

Use Design Compiler to synthesize the circuit in order to meet design constraints such as timing, area, testability, and power.

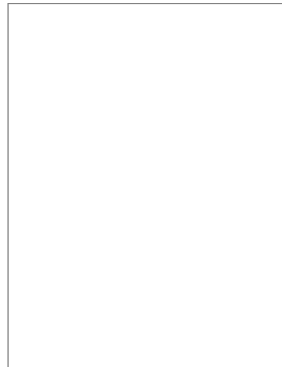
Encoder  $2n$  entrees et  $n$  sorties



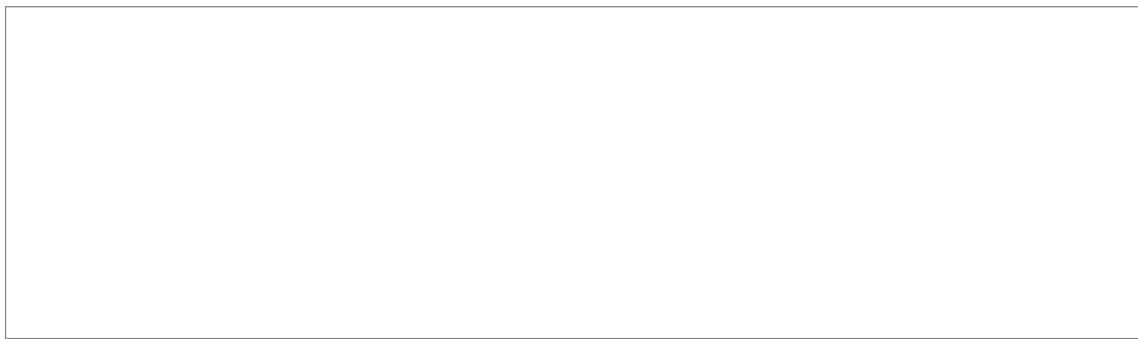
decoder  $n$  entree et  $2n$  sorties



multiplexeur:



demultiplexeur



```
// multiplexer
module mux4( input a, b, c, d
             input [1:0] sel,
             output out );

    assign out = ( sel == 0 ) ? a :
                 ( sel == 1 ) ? b :
                 ( sel == 2 ) ? c :
                 ( sel == 3 ) ? d :
                 0;

endmodule
```

```
module mux4(input a,b,c,d
            input [1:0] sel,
```



```
(in == 3'b111 ) ? 8'b1000_0000 : 8'h00;
```

```
endmodule
```

```
module decoder_always (in,out);
```

```
    input [2:0] in;  
    output [7:0] out;  
    reg [7:0] out;
```

```
    always @ (in)
```

```
    begin
```

```
        out = 0;
```

```
        case (in)
```

```
            3'b001 : out = 8'b0000_0001;
```

```
            3'b010 : out = 8'b0000_0010;
```

```
            3'b011 : out = 8'b0000_0100;
```

```
            3'b100 : out = 8'b0000_1000;
```

```
            3'b101 : out = 8'b0001_0000;
```

```
            3'b110 : out = 8'b0100_0000;
```

```
            3'b111 : out = 8'b1000_0000;
```

```
        endcase
```

```
    end
```

```
endmodule
```

```
module flif_flop (clk,reset, q, d);
```

```
    input clk, reset, d;
```

```
    output q;
```

```
    reg q;
```

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

```
    begin
```

```
        if (reset == 1) begin
```

```
            q <= 0;
```

```
        end else begin
```

```
            q <= d;
```

```
        end
```

```
    end
```

```
endmodule
```

```

module decoder_using_case (
binary_in  , // 4 bit binary input
decoder_out , // 16-bit out
enable     // Enable for the decoder
);
input [3:0] binary_in ;
input  enable ;
output [15:0] decoder_out ;

reg [15:0] decoder_out ;

always @ (enable or binary_in)
begin
    decoder_out = 0;
    if (enable) begin
        case (binary_in)
            4'h0 : decoder_out = 16'h0001;
            4'h1 : decoder_out = 16'h0002;
            4'h2 : decoder_out = 16'h0004;
            4'h3 : decoder_out = 16'h0008;
            4'h4 : decoder_out = 16'h0010;
            4'h5 : decoder_out = 16'h0020;
            4'h6 : decoder_out = 16'h0040;
            4'h7 : decoder_out = 16'h0080;
            4'h8 : decoder_out = 16'h0100;
            4'h9 : decoder_out = 16'h0200;
            4'hA : decoder_out = 16'h0400;
            4'hB : decoder_out = 16'h0800;
            4'hC : decoder_out = 16'h1000;
            4'hD : decoder_out = 16'h2000;
            4'hE : decoder_out = 16'h4000;
            4'hF : decoder_out = 16'h8000;
        endcase
    end
end

endmodule

```

```

module encoder_using_case(
binary_out , // 4 bit binary Output
encoder_in , // 16-bit Input
enable     // Enable for the encoder
);
output [3:0] binary_out ;
input  enable ;
input [15:0] encoder_in ;

reg [3:0] binary_out ;

always @ (enable or encoder_in)
begin
    binary_out = 0;
    if (enable) begin
        case (encoder_in)
            16'h0002 : binary_out = 1;
            16'h0004 : binary_out = 2;
            16'h0008 : binary_out = 3;
            16'h0010 : binary_out = 4;
            16'h0020 : binary_out = 5;

```

```

        16'h0040 : binary_out = 6;
        16'h0080 : binary_out = 7;
        16'h0100 : binary_out = 8;
        16'h0200 : binary_out = 9;
        16'h0400 : binary_out = 10;
        16'h0800 : binary_out = 11;
        16'h1000 : binary_out = 12;
        16'h2000 : binary_out = 13;
        16'h4000 : binary_out = 14;
        16'h8000 : binary_out = 15;
    endcase
end
end

endmodule

```

```

module up_counter    (
    out      , // Output of the counter
    enable   , // enable for counter
    clk      , // clock Input
    reset     // reset Input
);
//-----Output Ports-----
    output [7:0] out;
//-----Input Ports-----
    input enable, clk, reset;
//-----Internal Variables-----
    reg [7:0] out;
//-----Code Starts Here-----
    always @(posedge clk)
    if (reset) begin
        out <= 8'b0 ;
    end else if (enable) begin
        out <= out + 1;
    end

endmodule

```



