# DIGITAL CIRCUIT DESIGN WITH HDL

# Course outline

- Chapter 1: Introduction
- Chapter 2: Verilog Language Concepts
- Chapter 3: Structural modeling
- Chapter 4: Behavioral modeling
- Chapter 5: Finite State machines
- Chapter 6: Tasks and Functions
- Chapter 7: Functional Simulation/Verification (Testbench)
- Chapter 8: Synthesis of Combinational and Sequential Logic
- Chapter 9: Post-synthesis design tasks
- Chapter 10: VHDL introduction

**Ref: Michael D. Ciletti_Advanced Digital Design with the Verilog HDL– Chapter 6**

# Task of the designer

Understand how to synthesize combinational logic

Understand how to synthesize sequential logic

Understand how language constructs synthesize

Anticipate the results of synthesis

Adhere to style conventions

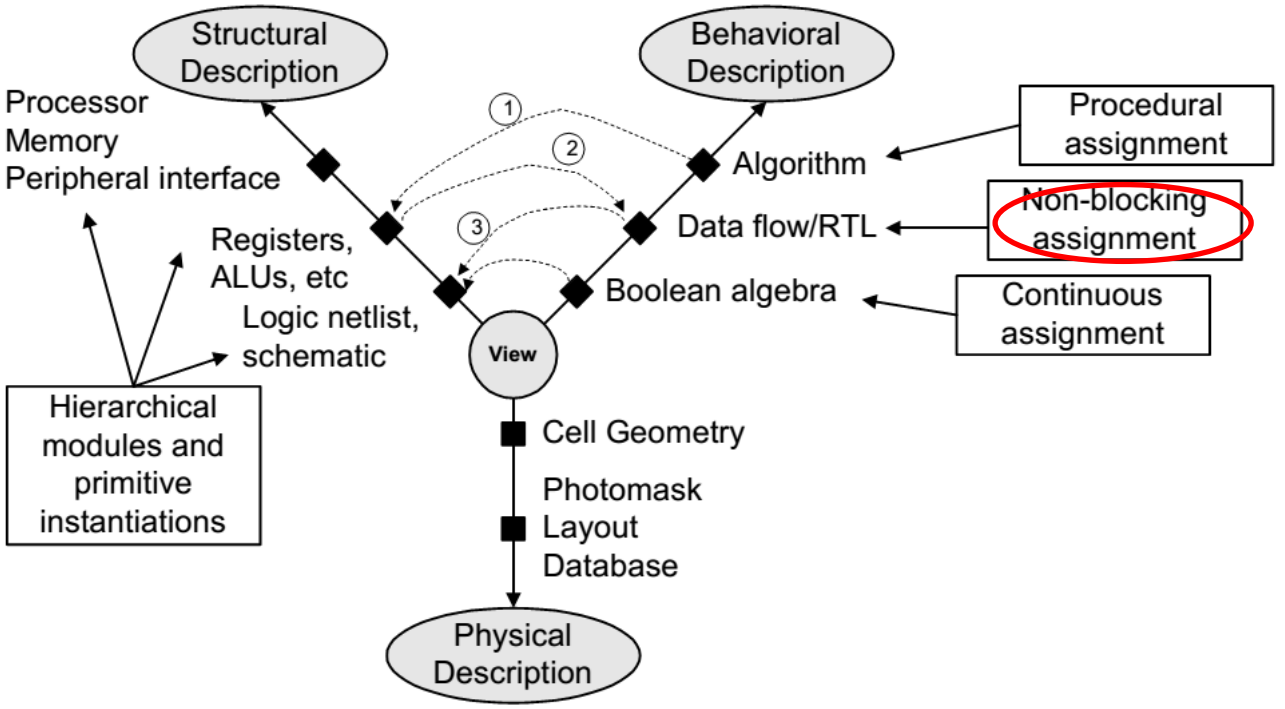
- Understanding how to write synthesis-friendly Verilog model is the key to automated design methods.

# Synthesis tool

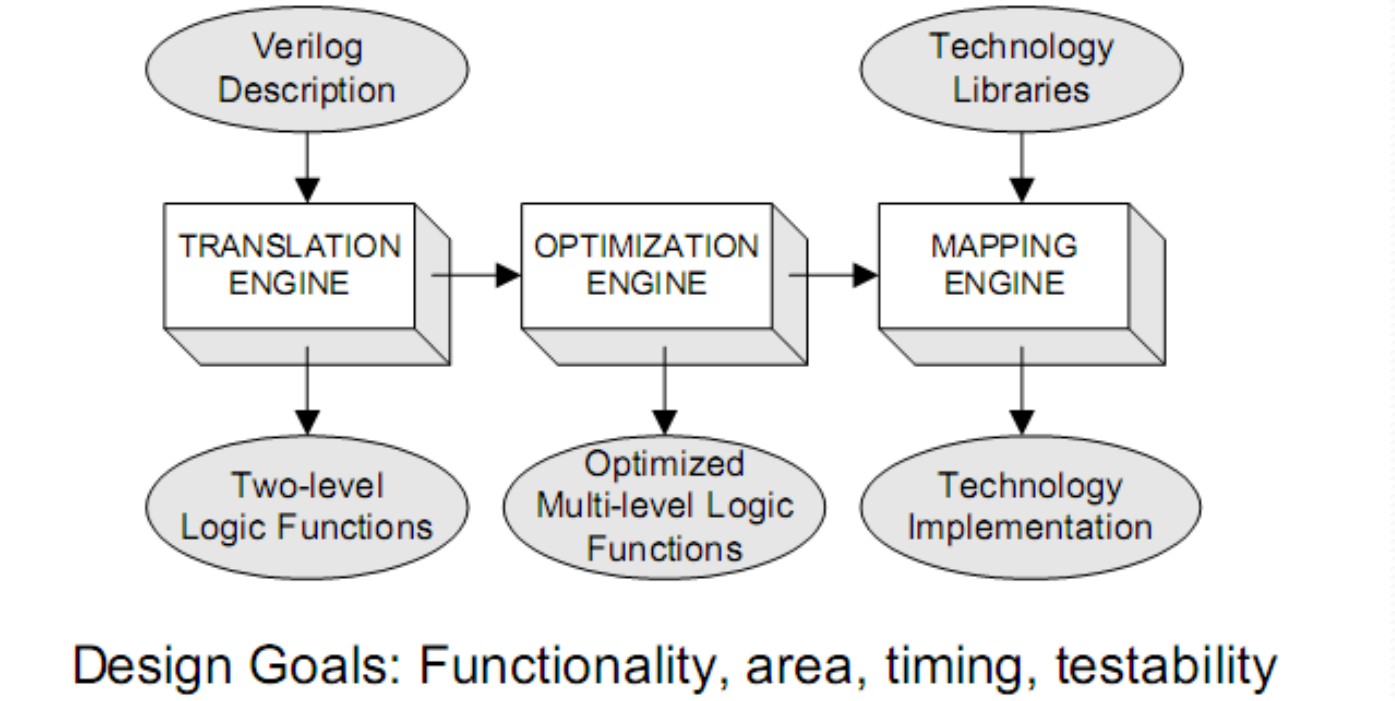
- Synthesis tools perform many tasks:
  - Detect and eliminate redundant logic
  - Detect combination feedback loops
  - Exploit don’t care condition
  - Detect unused states
  - Detect and collapse equivalent states
  - Synthesize optimal, technology mapping
  - …

# Modified Y-chart

Synthesis creates a sequence of transformations between views of a circuit, from a higher level of abstraction to a lower one, with each step leading to a more detailed description of the physical reality.



# Synthesis tool organization

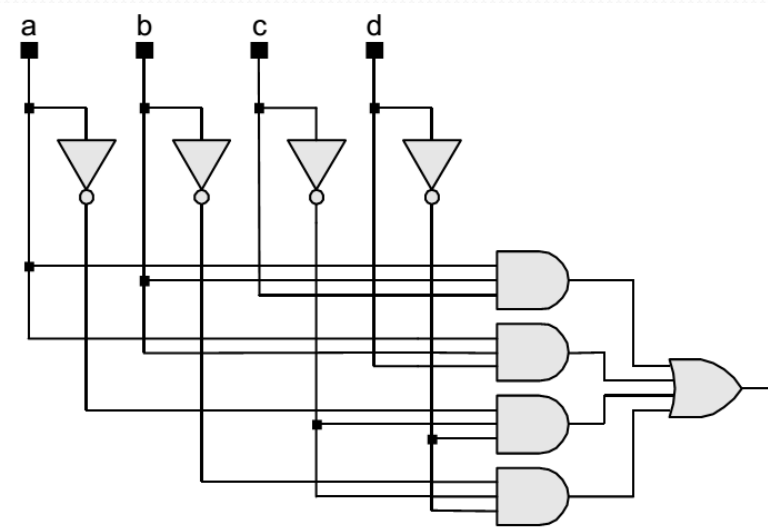


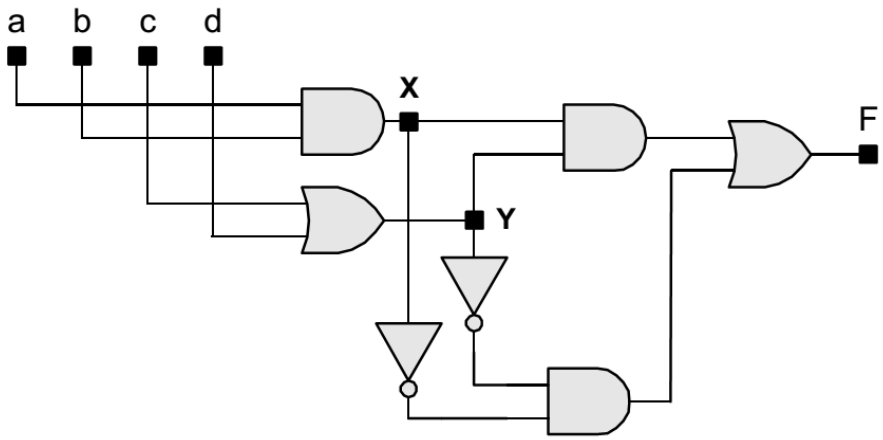

- Optimization: with logic transformation (ex: Decomposition, Factoring, Extraction, Substitution…)

# Decomposition

- Decompose a function into new nodes
- Re-use the new nodes in fanout paths

F = abc + abd + a'b'c' + b'c'd'  Two-level logic!

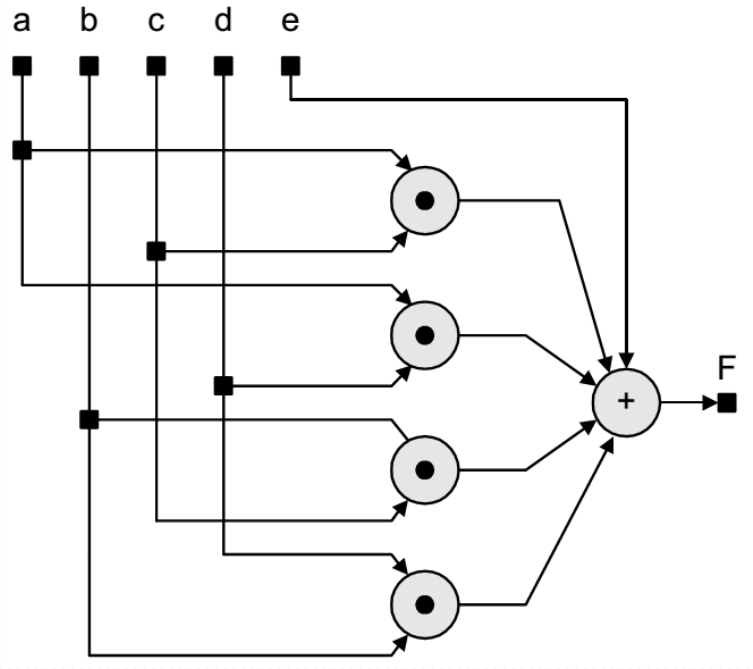


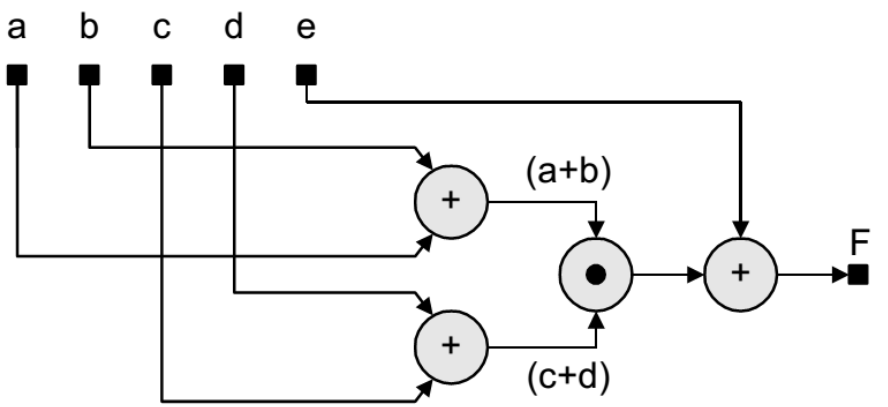

F = XY + X' Y'
X = ab
Y = c + d

# Factoring

$$F = ac + ad + bc + bd + e$$




$$F = (a + b)(c + d) + e$$



# Synthesis of Combinational Logic

Options:

- Netlist of primitives
- User-defined primitive
- Continuous assignments
- Level-sensitive cyclic behavior
- Procedural continuous assignment (***assign ... deassign*** )

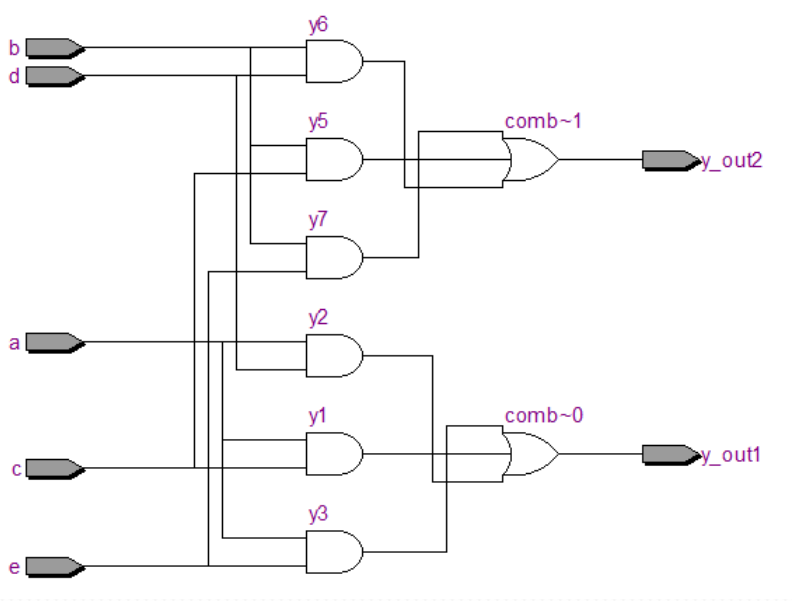
*Some EDA vendors do not support synthesis of UDP and assign…deassign*

# Synthesis: Netlist of Primitives

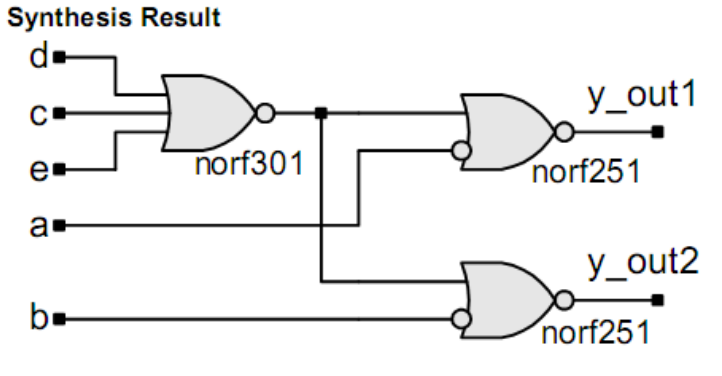
**Example:**

```
module boole_opt     (y_out1, y_out2, a, b, c, d, e);
 output          y_out1, y_out2;
 input           a, b, c, d, e;

 and             (y1, a, c);
 and             (y2, a, d);
 and             (y3, a, e);
 or              (y4, y1, y2);
 or              (y_out1, y3, y4);
 and             (y5, b, c);
 and             (y6, b, d);
 and             (y7, b, e);
 or              (y8, y5, y6);
 or              (y_out2, y7, y8);
endmodule
```

# Synthesis: Netlist of Primitives




*Quartus synthesis result*



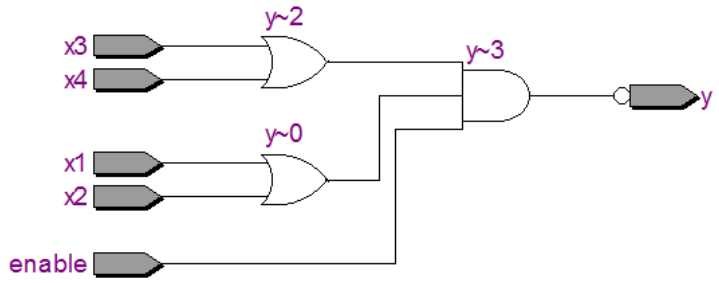

*Another synthesis result*

- Loại bỏ những dư thừa
- Bảo đảm mạch nhỏ nhất

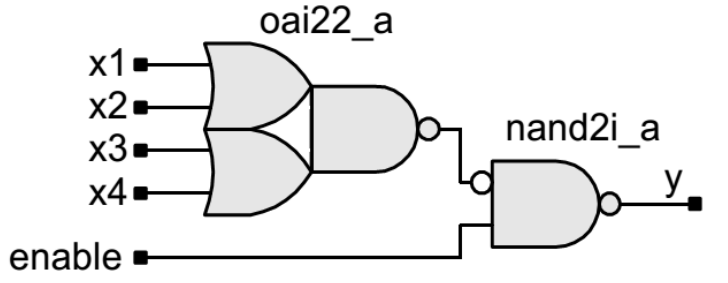
# Synthesis: Continuous Assignment

- Built-in operators have physical counterparts
- Continuous assignment statements are synthesizable
- Will produce (1) combinational logic, (2) latch, (3) three-state output

```
module or_nand (y, enable, x1, x2, x3, x4);
  output y;
  input     enable, x1, x2, x3, x4;

  assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```




*Quartus synthesis result*

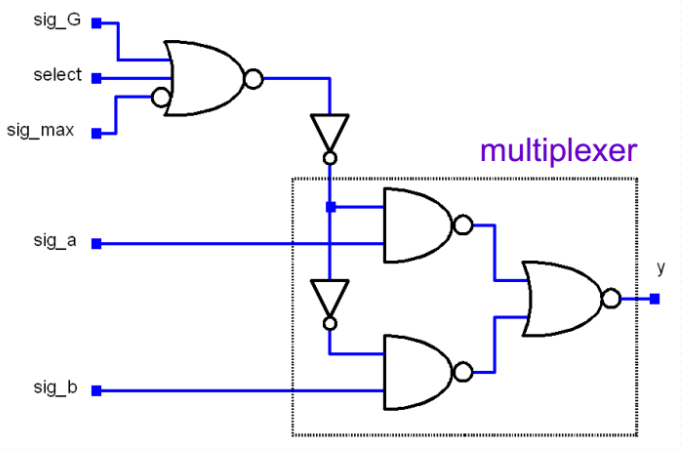



*Another synthesis result*

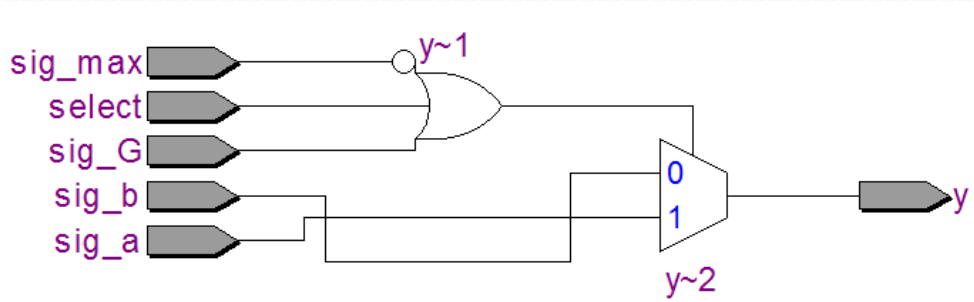
# Synthesis: Continuous Assignment

Synthesis: conditional operator in continuous assignment

```
module udp(y, select, sig_G, sig_max, sig_a, sig_b);
output y;
input select, sig_G, sig_max, sig_a, sig_b;
assign y = (select == 1) || (sig_G ==1) || (sig_max == 0) ? sig_a : sig_b;
endmodule
```




*Another synthesis result*

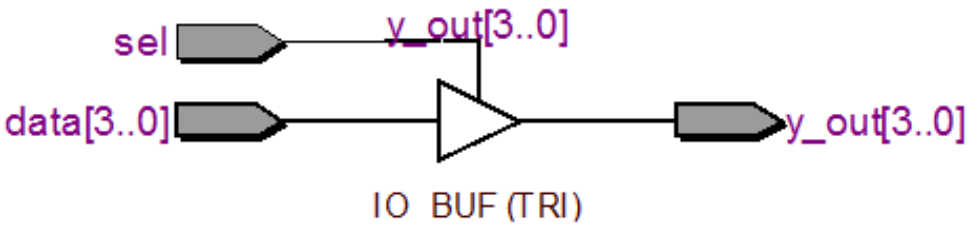



*Quartus synthesis result*

# Synthesis: Continuous Assignment

A conditional operator assigns the value z to the right-hand side expression of a continuous assignment will synthesize to a three-state device.

```
module udp (y_out, sel, data);
output [3:0] y_out;
input [3:0] data;
input sel;
assign y_out = sel ? data :  4'bz;
endmodule
```




*Quartus synthesis result*

# Synthesis: Level-sensitive cyclic behavior

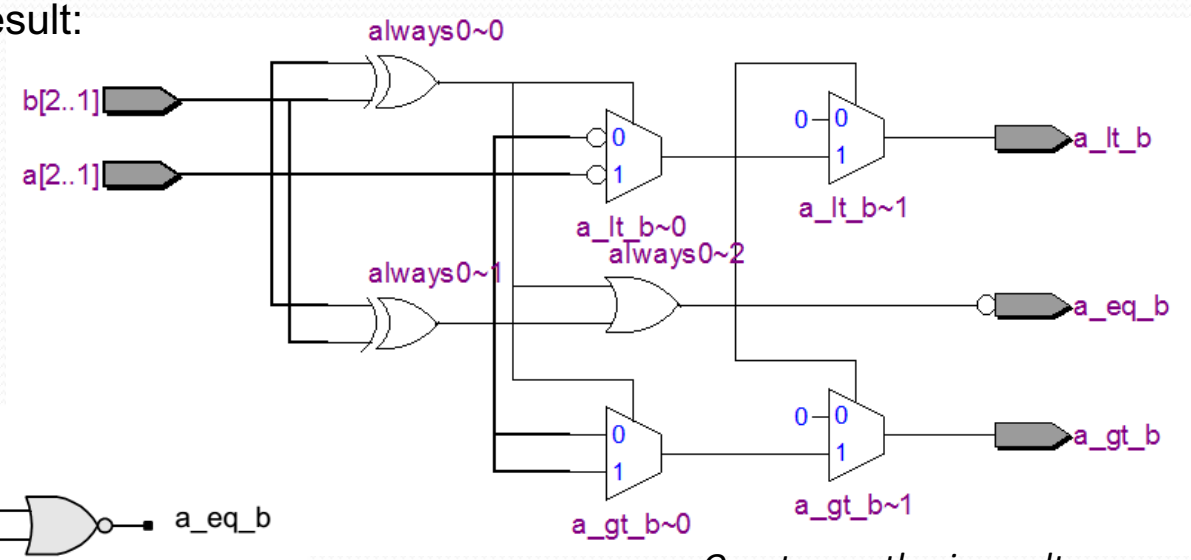
A level-sensitive cyclic behavior will synthesize to combinational logic if it assigns a value to each output for every possible value of its inputs.

- The event control expression of the behavior must be sensitive to every input
- Every path of the activity flow must assign value to every output.

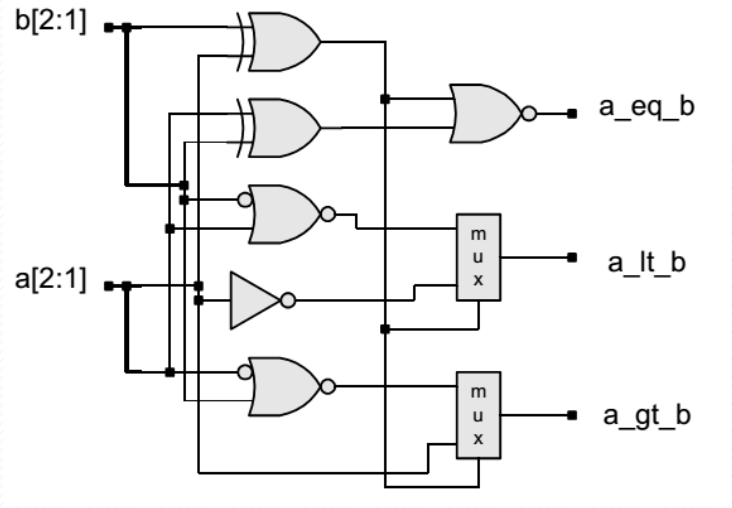
```
module comparator (a_gt_b, a_lt_b, a_eq_b, a, b);
 parameter     size = 2;
 output                a_gt_b, a_lt_b, a_eq_b;
 input         [size: 1]       a, b;
 reg                   a_gt_b, a_lt_b, a_eq_b;
 integer               k;

 always @  ( a or b) begin: compare_loop
  for (k = size; k > 0; k = k-1) begin
    if (a[k] != b[k]) begin
      a_gt_b = a[k];
      a_lt_b = ~a[k];
      a_eq_b = 0;
    disable compare_loop;
   end          // if
  end           // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
 end            // compare_loop
endmodule
```

# Synthesis: Level-sensitive cyclic behavior

Comparator synthesis result:




*Quartus synthesis result*

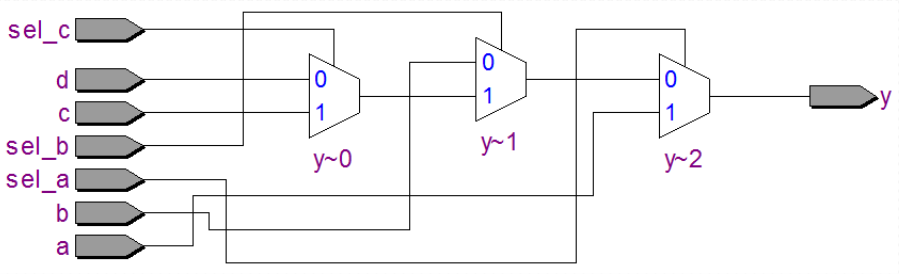



*Another synthesis result*

# Synthesis: Priority structure

- An ***if*** statement implies higher priority to the first branch than to the remaining branches.
- If branching is mutually exclusive, synthesis produces a mux structure
- Otherwise create a priority structure

```
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
 output   y;
 input    a, b, c, d, sel_a, sel_b, sel_c;
 reg  y;

   always @  (sel_a or sel_b or sel_c or a or b or c or d)
    begin
      if     (sel_a == 1)        y = a; else      // highest priority
      if     (sel_b == 0)        y = b; else
      if     (sel_c == 1)        y = c; else
                                 y = d;           // lowest priority
    end
endmodule
```




*Quartus synthesis result*

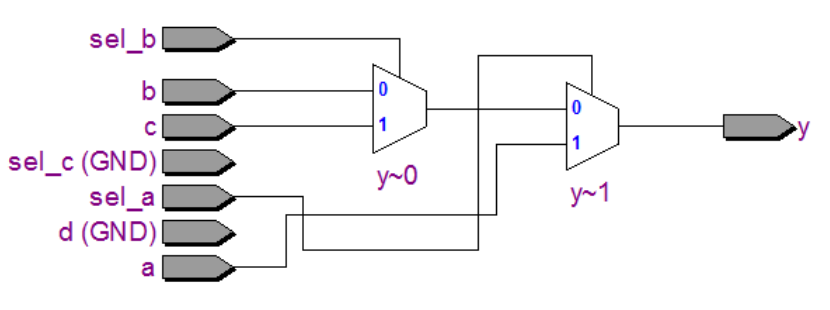
*Change to y=1'bx (or 1'bz) -> check synthesis (next slide)*

- Case is synthesized similarly.

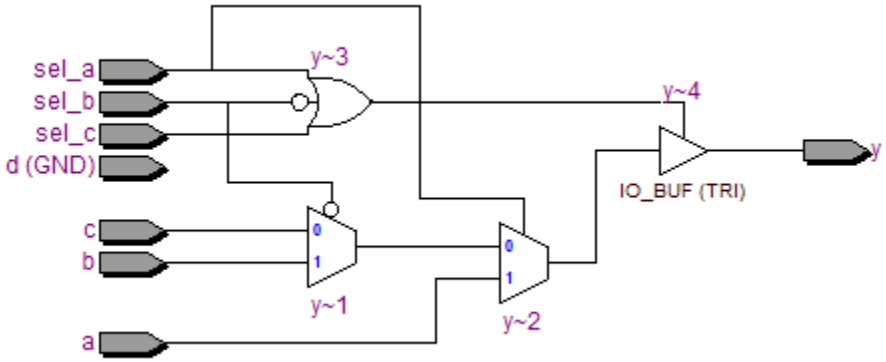
Even when the list of case items is not mutually exclusive a synthesis tool might allow the user to direct that they be treated without priority (e.g., Synopsys *parallel_case* directive).

# Exploit don’t care condition

An assignment to **x** in a **case** or an **if** statement will be treated as a don't care condition in synthesis.

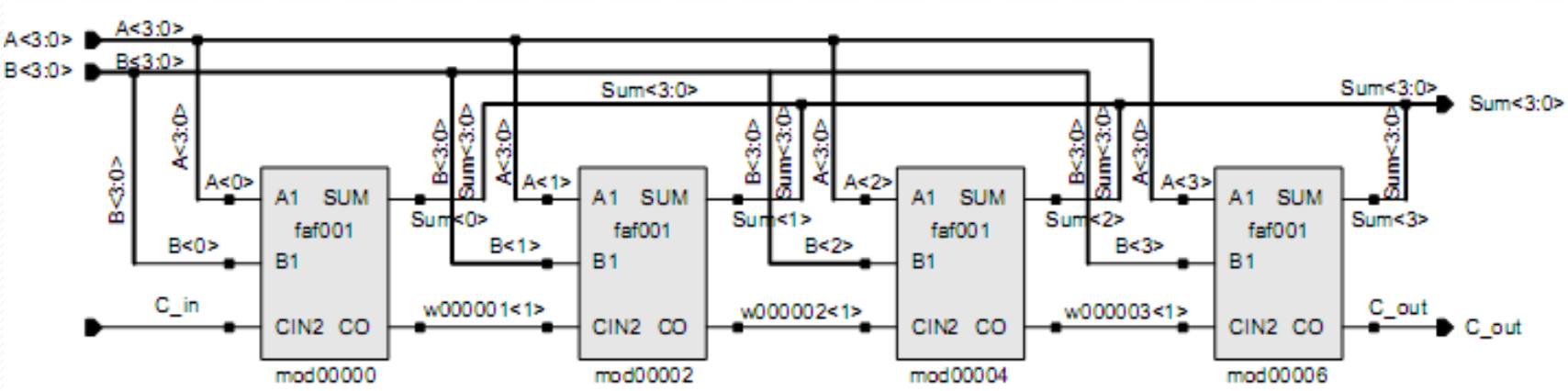



An assignment to **z** in **case** or if statement -> tri-state driver

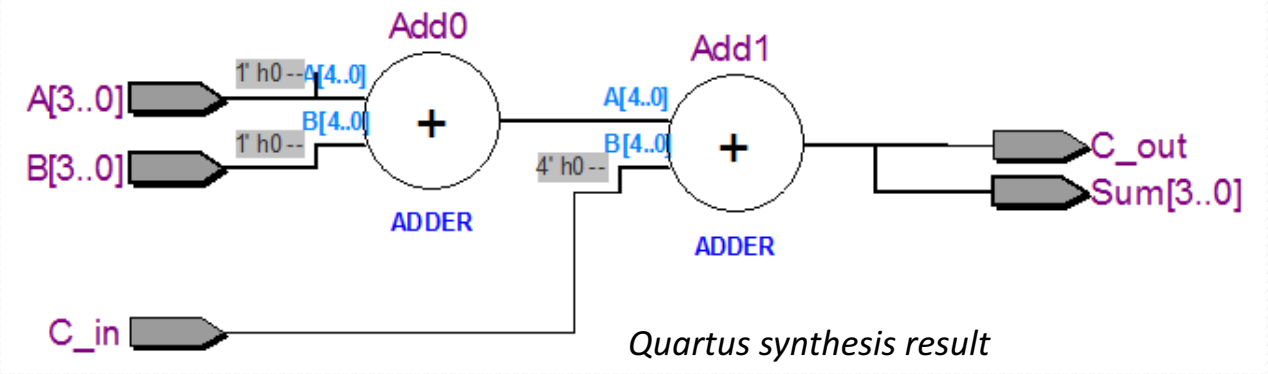


# Synthesis ASICs cell

Synthesis Result (With Library Cells)




An alternative implementation




*Quartus synthesis result*

# Synthesis: Resource sharing

- If the dataflow within the behavior do not conflict, the resource can be shared between one or more paths

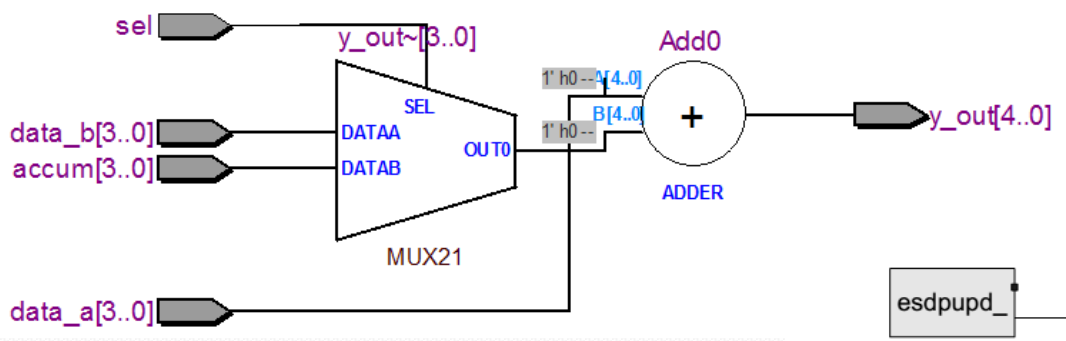
```
assign y_out = sel ? data_a + accum : data_a + data_b;
```

Consequently, the operators can be implemented by a shared adder whose input data-paths are multiplexed. This feature is vendor-dependent. If the tool does not automatically implement resource sharing, the description must be written to force the sharing.

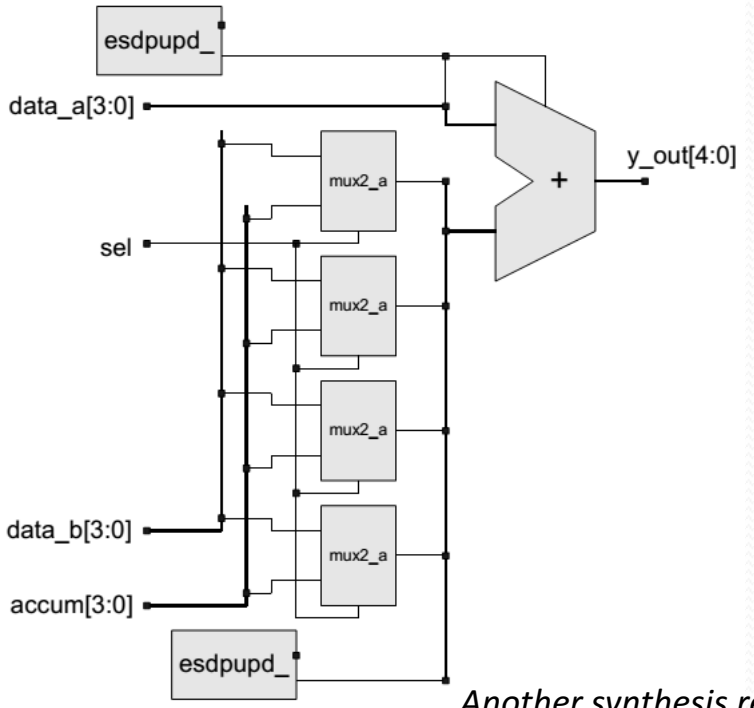
- The use of parentheses in the description in **res_share** forces the synthesis tool to multiplex the datapath

```
module res_share (y_out, sel, data_a, data_b, accum);
  output  [4: 0]    y_out;
  input   [3: 0]    data_a, data_b, accum;
  input             sel;

  assign y_out = data_a + (sel ? accum : data_b);
endmodule
```

# Synthesis: Resource sharing




*Quartus synthesis result*




*Another synthesis result*

# Synthesis of sequential logic with latch

- A set of feedback –free netlist of combinational primitives will synthesize into latch-free combinational logic.
- A set of feedback –free netlist of continuous assignments will synthesize into latch-free combinational logic.
- A continuous assignment with feedback in a conditional operator will synthesize into a latch.

```
assign data_out = (CS_b == 0) ? (WE_b == 0) ? data_in : data_out : 1'bz;
```

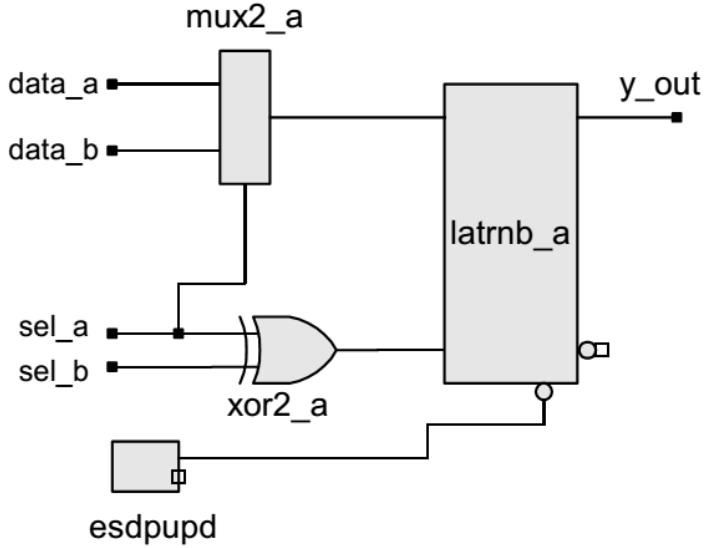
# Synthesis of sequential logic with Latch

- Accidental synthesis of Latch
- Intentional synthesis of Latch

# Accidental synthesis of Latch

- A Verilog description of combinational logic must assign value to the outputs for all possible values of the inputs.
- Incomplete specification infers Latch

```
module mux_latch (y_out, sel_a, sel_b, data_a, data_b);
  output        y_out;
  input         sel_a, sel_b, data_a, data_b;
  reg           y_out;

always @ ( sel_a or sel_b or data_a or data_b)
  case ({sel_a, sel_b})
    2'b10: y_out = data_a;
    2'b01: y_out = data_b;
  endcase
endmodule
```

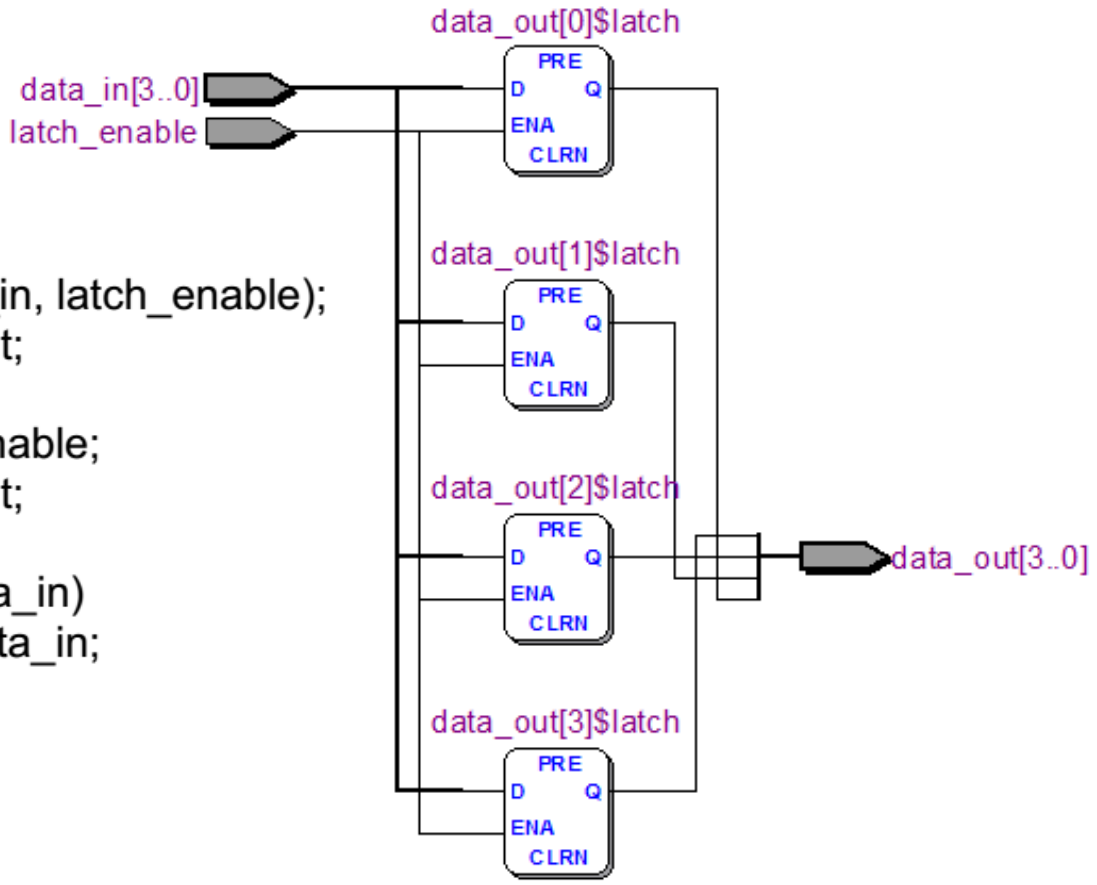



*A synthesis result*

# Intentional synthesis of Latch

**Example:**

```
module latch_if1(data_out, data_in, latch_enable);
  output        [3: 0]      data_out;
  input         [3: 0]      data_in;
  input                     latch_enable;
  reg           [3: 0]      data_out;

  always @  (latch_enable or data_in)
    if (latch_enable) data_out = data_in;
      else data_out = data_out;
endmodule
```



# Synthesis of sequential logic with flip-flops

❖ Flip-flop or Latch

- Memory inferred for an edge-sensitive cyclic behavior will be synthesized as a flip flop
- Memory inferred for a level-sensitive cyclic behavior or a continuous assignment with feedback will be synthesized as a latch
- Note: an incomplete conditional statement (***if...else*** ) or a ***case*** statement in a level-sensitive cyclic behavior will synthesize to a latch.
- Note: an incomplete conditional statement statement (***if...else*** ) or a ***case*** statement in an edge-sensitive cyclic behavior will synthesize to a flip-flop with circuitry effectively implementing clock enable.

# Synthesis of sequential logic with flip-flops

❖ Synchronizing signal

- The order in which signals appear in the event control expression of an edge-sensitive cyclic behavior does not determine which signal is the synchronizing signal

- The sequence in which signals are decoded in the statement that follows the event control expression of an edge-sensitive cyclic behavior determines which of the edge-sensitive signals are control signals and which is the clock (i.e., the synchronizing signal).

Note: If the event control expression is sensitive to the edge of more than one signal, an ***if*** statement must be the first statement in the behavior.

# Synthesis of sequential logic with flip-flops

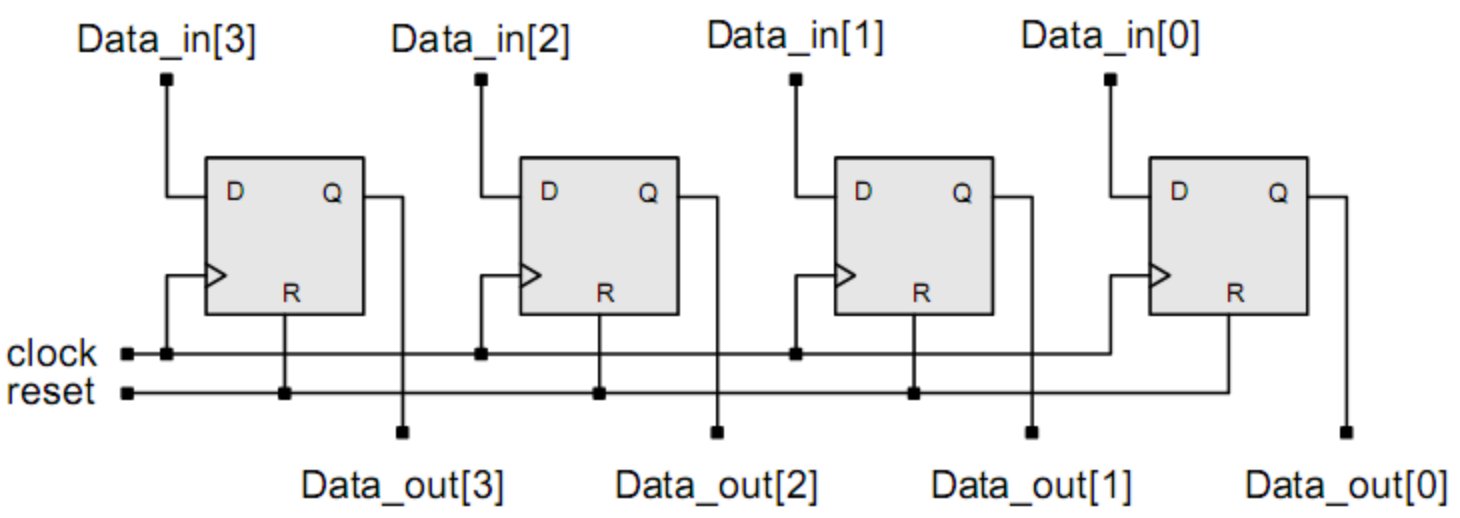
***Example:*** *Synthesis of a 4-bit Parallel Load Data Register*

```
module D_reg4_a  (Data_out, clock, reset, Data_in);
  output      [3: 0]     Data_out;
  input       [3: 0]     Data_in;
  input                  clock, reset;
  reg         [3: 0]     Data_out;

  always @  (posedge clock or posedge reset)
    begin
      if (reset == 1'b1) Data_out <= 4'b0;
        else Data_out <= Data_in;
      end
endmodule
```

- The positive edge of *reset* appears in the event control expression and appears in the first clause of the *if* statement
- The positive edge of *clock* also appears in the event control expression, but is not explicitly decoded by the branch statement that follows the event control expression.

# Synthesis of sequential logic with flip-flops

***Example:*** *Synthesis of a 4-bit Parallel Load Data Register*

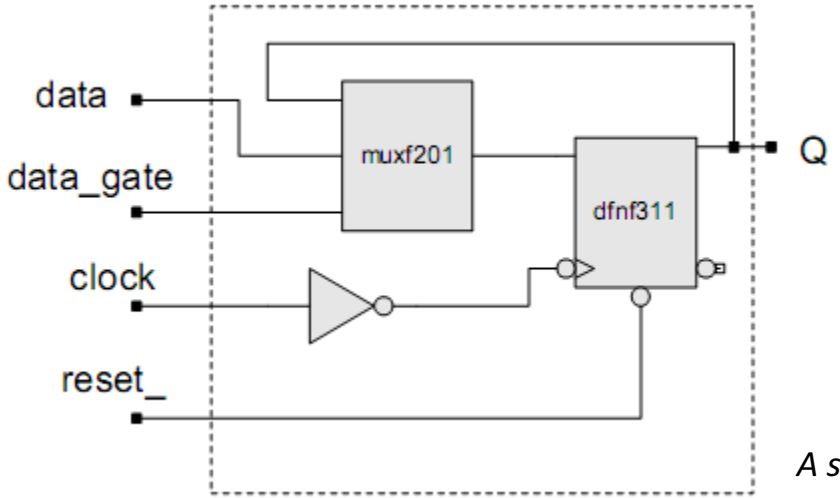



*A synthesis result*

# Synthesis of sequential logic with flip-flops

***Another example:***

```
module best_gated_clock (clock, reset_, data_gate, data, Q);
  input       clock, reset_, data, data_gate;
  output      Q;
  reg         Q;

  always @  (posedge clock or negedge reset_)
    if (reset_ == 0) Q <= 0; else if (data_gate) Q <= data;
endmodule
```



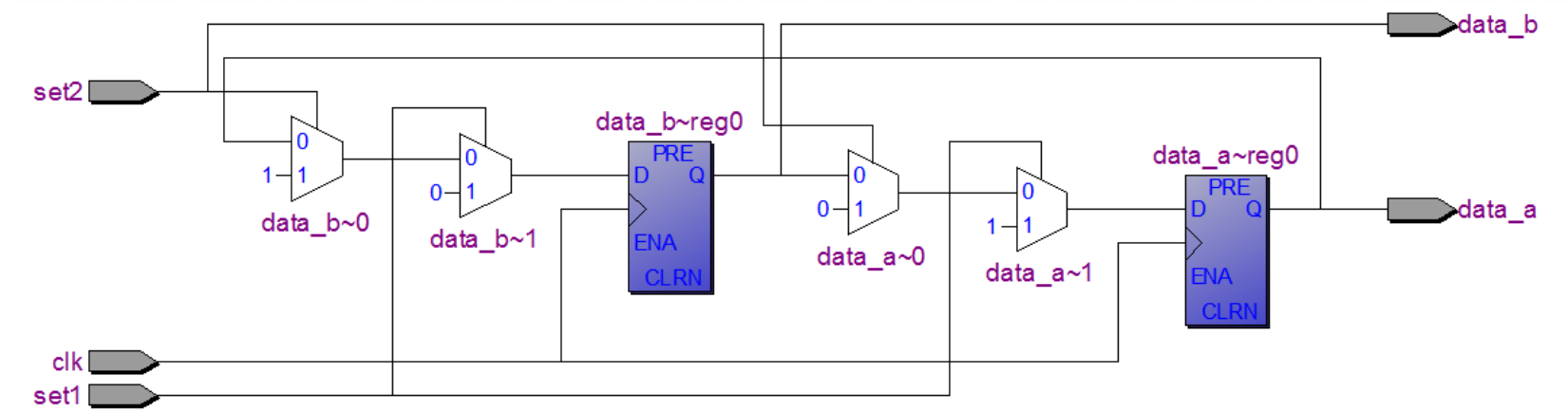

*A synthesis result*

# Synthesis tip

A variable that is referenced within an edge-sensitive behavior before it is assigned value by the behavior will be synthesized as the output of a flip-flop.

```
module swap_synch (data_a, data_b, set1, set2, clk); // Typo at text
  output      data_a, data_b;                        // Typo at text
  input       clk, set1, set2;
  reg         data_a, data_b;

  always @  (posedge clk)
    begin
      if (set1) begin data_a <= 1; data_b <= 0; end else
        if (set2) begin data_a <= 0; data_b <= 1; end else
          else
            begin
              data_b <= data_a;
              data_a <= data_b;
            end
    end
endmodule
```

# Synthesis tip




*A synthesis result*

# Synthesis tip

A variable that is assigned value by a cyclic behavior before it is referenced within the behavior, but is not referenced outside the behavior, will be eliminated by the synthesis process.

```
module or4_behav (y, x_in);

  parameter   word_length = 4;
  output                    y;
  input       [word_length - 1: 0] x_in;
  reg                       y;
  integer                   k;    // Eliminated in synthesis

  always @ x_in
    begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1) begin
            y = 1;
            disable check_for_1;
        end
    end
endmodule
```

# Synthesis tip

*D_out* is not referenced outside the scope of the behavior

A synthesis tool will eliminate *D_out*. output of a flip-flop.

```
module empty_circuit (D_in, clk);
  input   D_in;
  input   clk;
  reg     D_out;

  always @ (posedge clk)  begin
    D_out <= D_in;
  end
endmodule
```

Note: If *empty_circuit* is modified to declare *D_out* as an output port, *D_out* will be synthesized as the output of a flip-flop.

# Synthesis of Data types

- The identity of nets that are primary inputs or outputs are retained in synthesis
- Internal nets may be eliminated by synthesis
- Integers are stored as 32-bit words (minimum per IEEE 1364)
- Used sized parameters rather than integer parameters

# Synthesis of Operators

Some operators may map directly into library cells (e.g. +, 1, <, >, =)
Synthesis might impose restrictions on an operator

- Shift operators is allowed only if the shift index is a constant
- Reduction, bitwise, and logical operators synthesize to equivalent gates
- Conditional operator synthesizes to a mux structure
- If both operands of * are constant the tool produces the constant result
- If one operand of * is a power of 2 the synthesis tool forms the result by left-shifting the other operand
- If one operand of / is a power of 2 the result will be formed by right-shifting the other operand

# Synthesis of Operators

Use parentheses within expressions to influence the outcome of synthesis

```
module operator_group (sum1, sum2, a, b, c, d);
  output [4: 0]     sum1, sum2;
  input  [3: 0]     a, b, c, d;


  assign sum1 = a + b + c + d;
  assign sum2 = (a + b) + (c + d);

endmodule
```

Synthesis results: