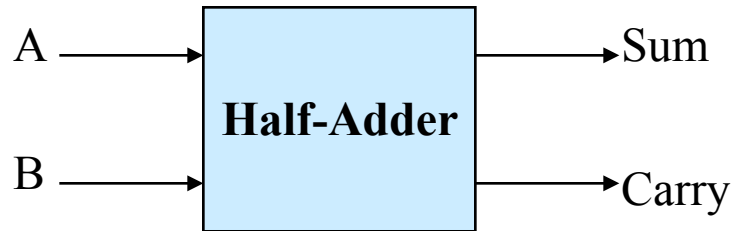


# Topic 7

## Introduction to Verilog HDL

# Descriptions of a Half Adder



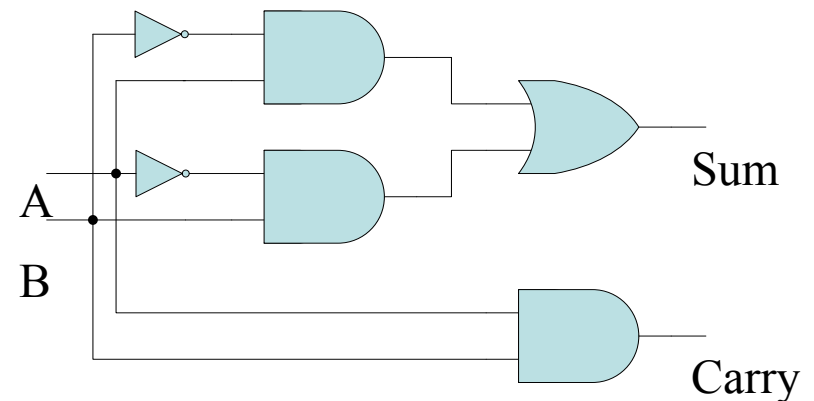
- Derive Boolean functions (sum-of-minterms) from the truth table for both outputs

$$\begin{aligned}\text{Sum} &= A'B + AB' = m_1 + m_2 = \Sigma (1, 2) \\ &= (A+B)(A'+B') = M_0 \cdot M_3 = \Pi (0, 3)\end{aligned}$$

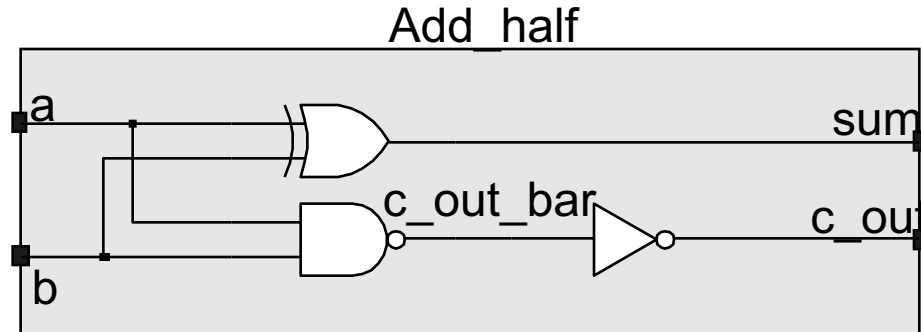
- Addition of two single bits A, B
- Based on the operations it performs, a truth table can be built

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$\begin{aligned}\text{Carry} &= AB = m_3 \\ &= (A+B)(A+B')(A'+B) \\ &= M_0 \cdot M_1 \cdot M_2 \\ &= \Pi (0, 1, 2)\end{aligned}$$



# Alternative Description of a Half Adder Hardware Description Language (HDL)



module name      module ports

```
module Add_half (sum, c_out, a, b);  
  input      a, b;  
  output     sum, c_out;      ← declaration of port modes
```

```
  wire       c_out_bar;      ← declaration of internal signal
```

```
  xor         (sum, a, b);  
  nand        (c_out_bar, a, b);  
  not         (c_out, c_out_bar);  
endmodule
```

instantiation of pre-defined primitive gates

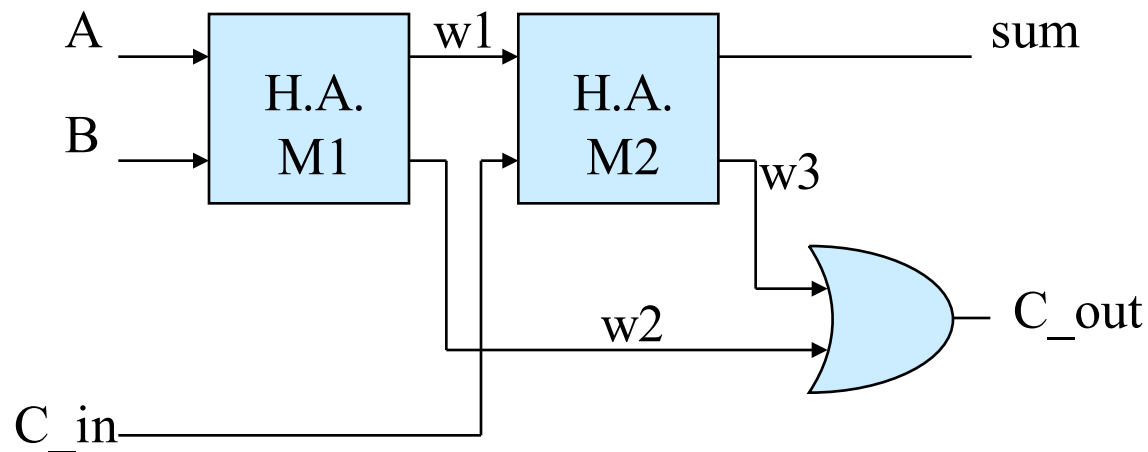
Same variable indicates connection

# Full Adder Implemented with Half Adder

```
module Add_full (sum, c_out, a, b, c_in); // parent module
  input    a, b, c_in;
  output   c_out, sum;
  wire     w1, w2, w3;

  Add_half M1 (w1, w2, a, b);           // child module
  Add_half M2 (sum, w3, w1, c_in);       // child module
  or (c_out, w2, w3);                   // primitive instantiation
endmodule
```

Module instance name



# 4-bit Carry-Ripple Adder

```
module Add_rca_4 (sum, c_out, a, b, c_in);
```

```
  output [3: 0] sum;
```

```
  output c_out;
```

```
  input [3: 0] a, b;
```

```
  input c_in;
```

```
  wire c_in2, c_in3, c_in4;
```

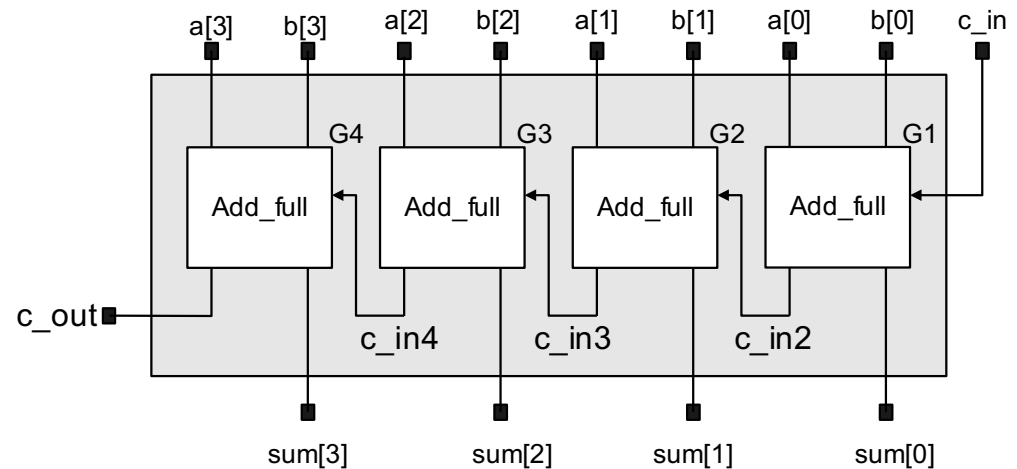
```
  Add_full M1 (sum[0], c_in2, a[0], b[0], c_in);
```

```
  Add_full M2 (sum[1], c_in3, a[1], b[1], c_in2);
```

```
  Add_full M3 (sum[2], c_in4, a[2], b[2], c_in3);
```

```
  Add_full M4 (sum[3], c_out, a[3], b[3], c_in4);
```

```
endmodule
```



# 16-bit Carry-Ripple Adder (Top)

```
module Add_rca_16 (sum, c_out, a, b, c_in);
```

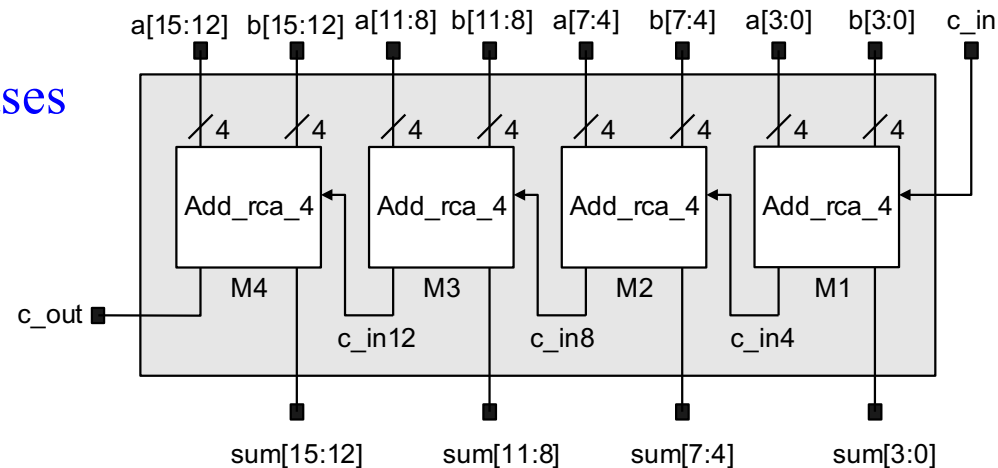
```
  output [15:0] sum;
```

```
  output c_out;
```

```
  input [15:0] a, b;
```

```
  input c_in;
```

16 bit buses



```
wire c_in4, c_in8, c_in12, c_out;
```

```
Add_rca_4 M1 (sum[3:0], c_in4, a[3:0], b[3:0], c_in);
```

```
Add_rca_4 M2 (sum[7:4], c_in8, a[7:4], b[7:4], c_in4);
```

```
Add_rca_4 M3 (sum[11:8], c_in12, a[11:8], b[11:8], c_in8);
```

```
Add_rca_4 M4 (sum[15:12], c_out, a[15:12], b[15:12], c_in12);
```

```
endmodule
```

# 2-bit Comparator



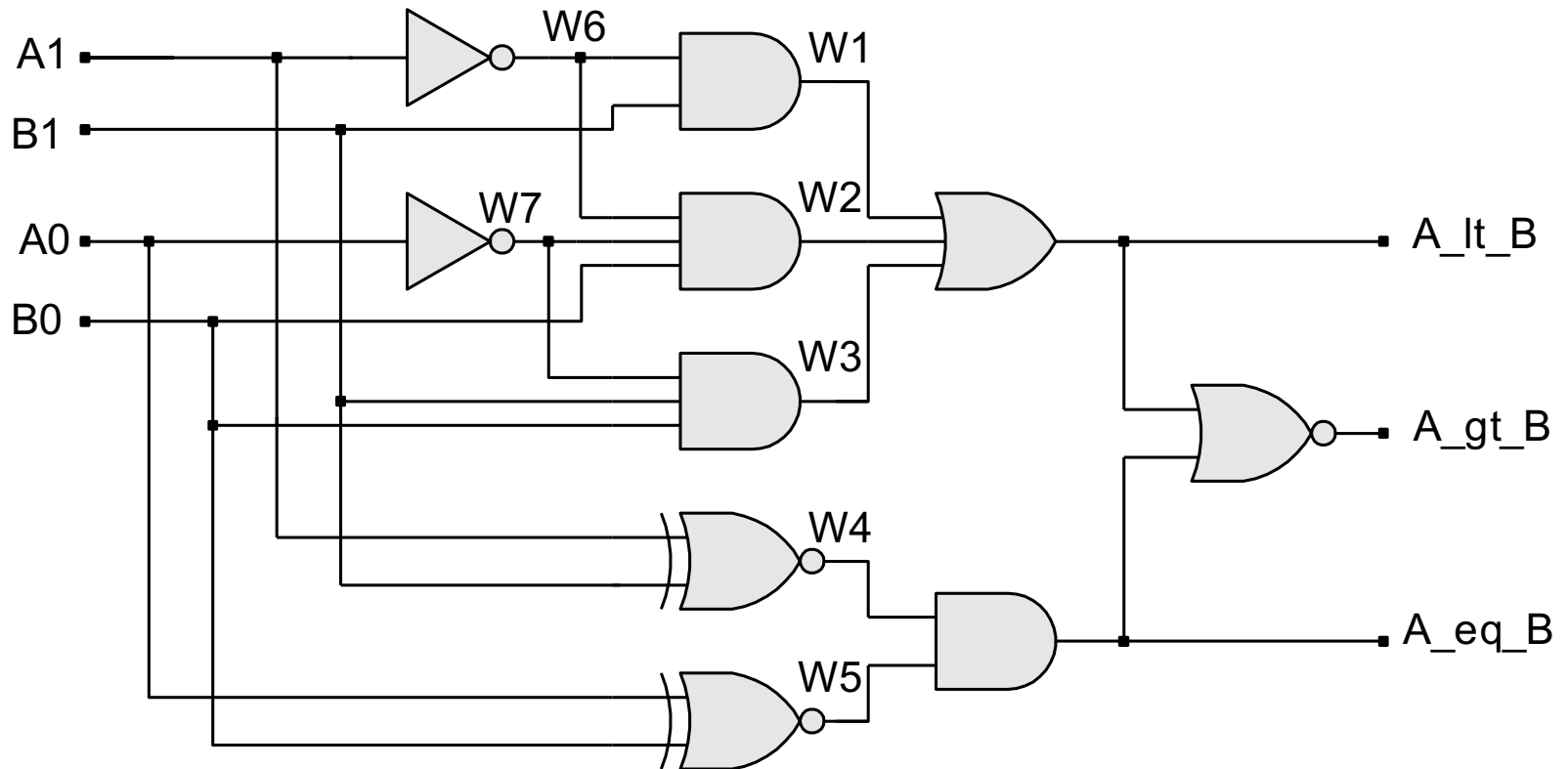
- Boolean equations:

$$A\_lt\_B = A1' B1 + A1' A0' B0 + A0' B1 B0$$

$$A\_gt\_B = A1 B1' + A0 B1' B0' + A1 A0 B0'$$

$$A\_eq\_B = A1' A0' B1' B0' + A1' A0 B1' B0 + A1 A0 B1 B0 + A1 A0' B1 B0'$$

# Gate-level Schematic

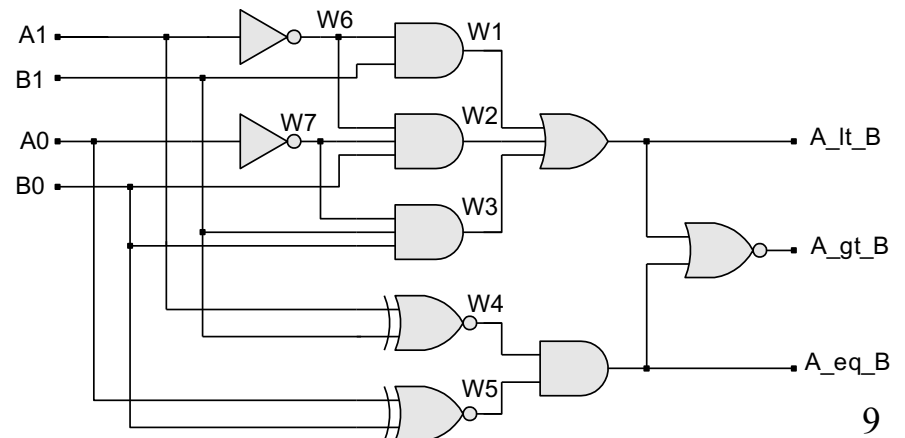




# Comparator – Structural Model

```
module compare_2_str (A_lt_B, A_gt_B, A_eq_B, A0, A1, B0, B1);  
  input      A0, A1, B0, B1;  
  output     A_lt_B, A_gt_B, A_eq_B;  
  
  wire       w1, w2, w3, w4, w5, w6, w7;  
  
  or    (A_lt_B, w1, w2, w3);  
  nor   (A_gt_B, A_lt_B, A_eq_B);  
  and   (A_eq_B, w4, w5);  
  and   (w1, w6, B1);  
  and   (w2, w6, w7, B0);  
  and   (w3, w7, B1, B0);  
  not   (w6, A1);  
  not   (w7, A0);  
  xnor  (w4, A1, B1);  
  xnor  (w5, A0, B0);  
endmodule
```

May be implicitly declared



# Comparator – RTL Model

```
module compare_2 (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);  
  input          A1, A0, B1, B0;  
  output         A_lt_B, A_gt_B, A_eq_B;  
  
  assign A_lt_B = (~A1) & B1 | (~A1) & (~A0) & B0 | (~A0) & B1 & B0;  
  assign A_gt_B = A1 & (~B1) | A0 & (~B1) & (~B0) | A1 & A0 & (~B0);  
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) | (~A1) & A0 & (~B1) & B0  
                | A1 & A0 & B1 & B0 | A1 & (~A0) & B1 & (~B0);  
  
endmodule
```

- 
- Continuous assignment statements
  - All concurrently executed



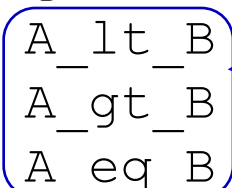
# Comparator – Alternative RTL Model

```
module compare_2_logic (A_lt_B, A_gt_B, A_eq_B,  
                        A1, A0, B1, B0);  
  
  input      A1, A0, B1, B0;  
  output     A_lt_B, A_gt_B, A_eq_B;  
  
  assign     A_lt_B = ({A1, A0} < {B1, B0});  
  assign     A_gt_B = ({A1, A0} > {B1, B0});  
  assign     A_eq_B = ({A1, A0} == {B1, B0});  
endmodule
```



Concatenation of A1 and A0

# Comparator – Behavioral Model

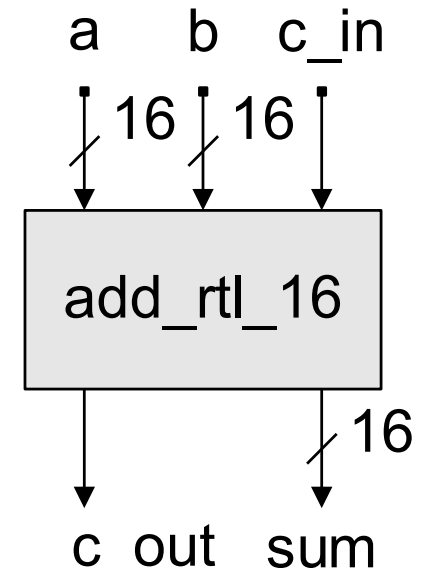
```
module compare_2_algo (A_lt_B, A_gt_B, A_eq_B, A,B);  
  input      [1:0] A, B;  2-bit bus  
  output    A_lt_B, A_gt_B, A_eq_B;  
  
  reg      A_lt_B, A_gt_B, A_eq_B;  
  
  always @(A or B)  Cyclic statement triggered upon @condition  
  begin  
      
    A_lt_B = 0;  
    A_gt_B = 0;  
    A_eq_B = 0;  
    if (A==B) A_eq_B = 1;  
    else if (A>B) A_gt_B = 1;  
    else A_lt_B = 1;  
  end  
endmodule
```

Destination variables inside always must be register (reg)

# RTL Alternative of 16-bit Adder

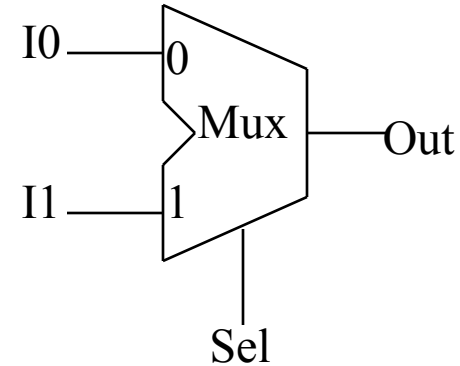
```
module add_rtl_16 (sum, c_out, a, b, c_in);  
  input  [15:0]      a, b;  
  input                c_in;  
  output [15:0]      sum;  
  output                c_out;  
  
  assign {c_out, sum} = a + b + c_in;  
endmodule
```

Concatenation of c\_out and sum



# 2-to-1 MUX

```
module MUX_2_1 (Out, I0, I1, Sel);  
  input I0, I1, Sel;  
  output Out;  
  reg Out;  
  
  always @(I0, I1, Sel) begin  
    case (Sel)  
      1'b0: Out = I0;  
      1'b1: Out = I1;  
      default Out = 0;  
    endcase  
  end  
endmodule
```



Like **switch...case...** in C/C++

# Flip-Flop – Modeling Clock Behavior

```
module D_ff (q, data_in, clk);  
  input data_in, clk;  
  output q;
```

```
  reg q;
```

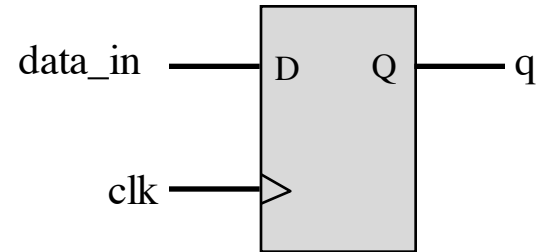
```
  always @ (posedge clk)
```

```
  begin
```

```
    q <= data_in;
```

```
  end
```

```
endmodule
```



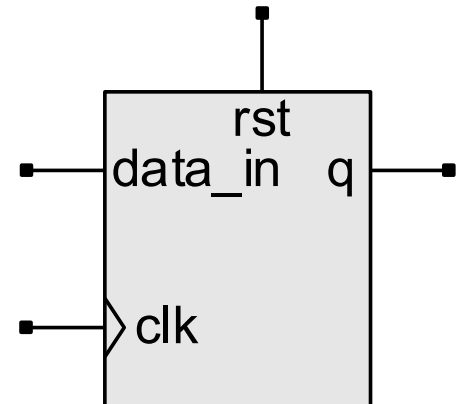
Rising edge of

Non-blocking assignment statement

# Synchronous Control Input

```
module D_ff (q, data_in, clk, syn_rst);  
  input data_in, clk, syn_rst;  
  output q;  
  
  reg q;  
  
  always @ (posedge clk)  
  begin  
    if (syn_rst == 1) q <= 0;  
    else q <= data_in;  
  end  
endmodule
```

Synchronous reset





# Asynchronous Control Input

```
module D_ff (q, data_in, clk, asyn_rst);  
  input data_in, clk, asyn_rst;  
  output q;
```

```
  reg q;
```

```
  always @ (posedge clk or posedge asyn_rst)
```

```
  begin
```

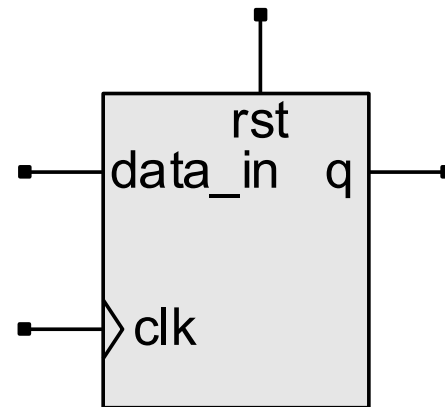
```
    if (asyn_rst == 1) q <= 0;
```

```
    else q <= data_in;
```

```
  end
```

```
endmodule
```

Asynchronous reset

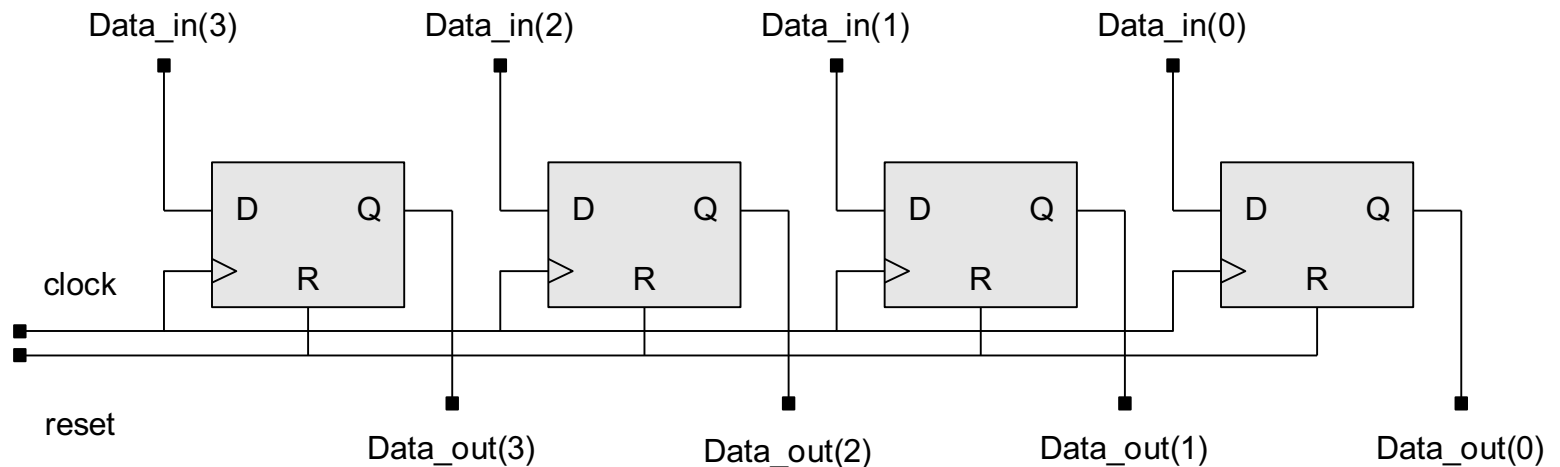


# Registers

- GENERAL RULE: A variable will be synthesized as a flip-flop when its value is assigned synchronously with an edge of a signal

```
module D_reg4 (Data_in, clock, reset, Data_out);  
  input      [3:0] Data_in;  
  input      clock, reset;  
  output     [3:0] Data_out;  
  reg       [3:0] Data_out;  
  always @ (posedge reset or posedge clock)  
    if (reset == 1'b1) Data_out <= 4'b0;  
    else                Data_out <= Data_in;  
endmodule
```

4 bits of 0



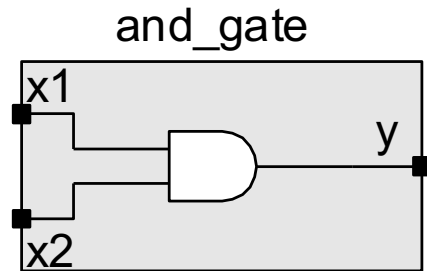
# Hardware Description Language (HDL)

- An HDL is a language that describes the hardware of digital systems in a textual form
  - Can describe digital system specified at different levels of abstraction
- There are many HDLs,
  - two most popular IEEE standards: VHDL and Verilog HDL;
  - other IEEE standards: SystemC, SystemVerilog, HandleC...

# Different HDLs

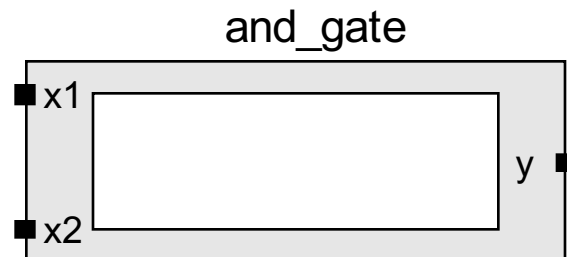
- VERILOG

```
module and_gate (y, x1, x2);  
    input    x1, x2;  
    output   y;  
  
    and (y , x1, x2);  
endmodule
```



- VHDL

```
entity and_gate is  
    port (x1, x2: in bit;  
          y: out bit);  
end and_gate;  
architecture data_flow of  
    and_gate is  
begin  
    y <= x1 and x2;  
end data_flow;
```

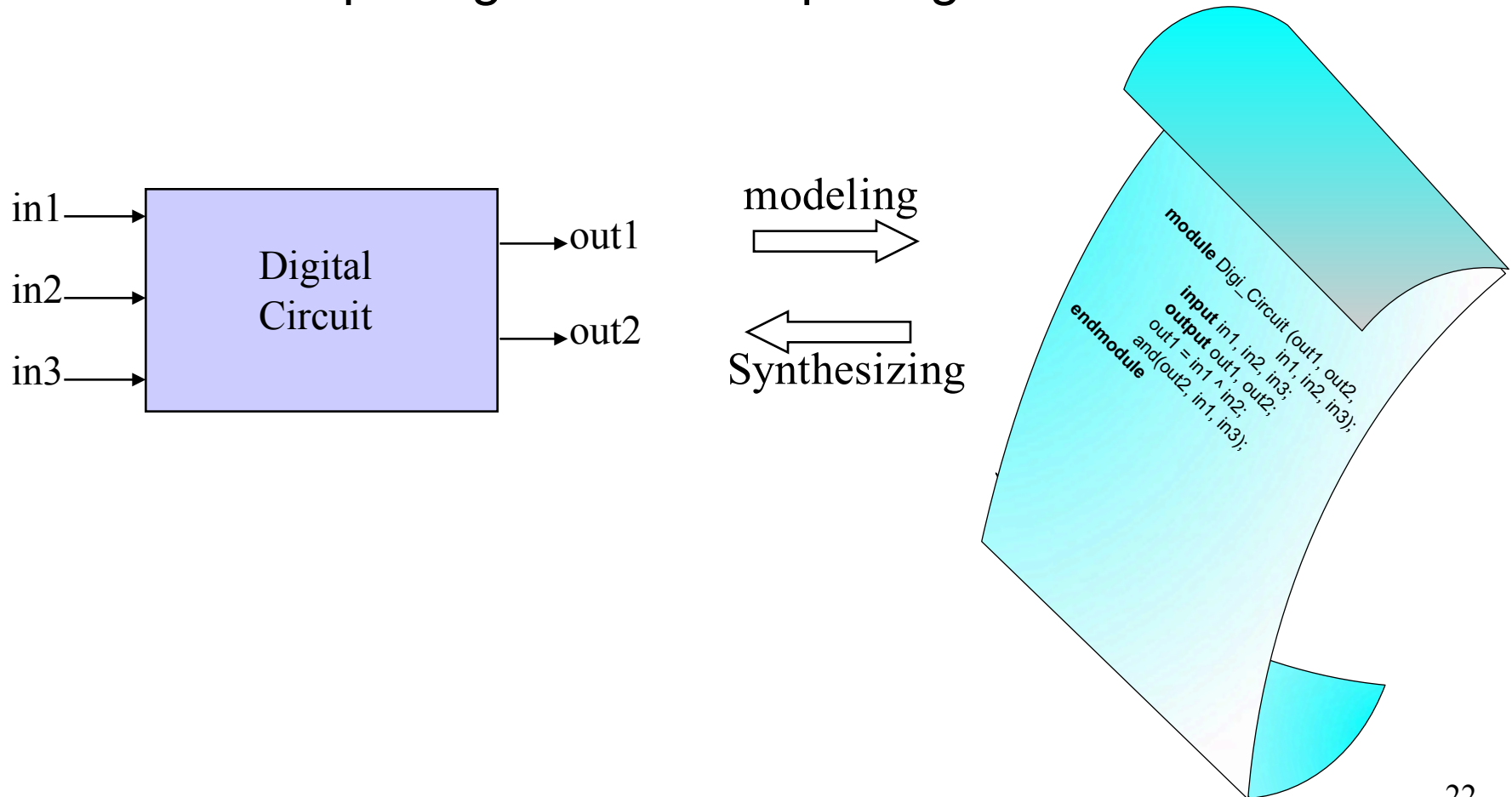


# Hardware Description Language (HDL)

- What is HDL used for?
  - Another way to describe digital circuit
  - Quick functional verification – simulation
  - Quick virtual to real circuit conversion – synthesis
- Advantages in digital design
  - Most ‘reliable’ design process, with minimum cost and time-to-market (TTM)
  - Reduce fault penalty!

# HDL Modeling

- The HDL model specifies a relationship (scheduling rule) between input signals and output signals



# Summary: Verilog Module Structure

**module** the\_design ( ... );

declarations: ports, constants, variables, events

declarations: tasks and functions

} Declaration

instantiations of primitives and assignments

continuous assignment: assign y = ...

behavioral statements (**initial**, **always**) {

procedural (blocking) assignment

procedural nonblocking assignment

procedural-continuous assign

event trigger

task calls

function calls

} Implementation

}

**endmodule**

# Parameterized Module

```
module Param_Examp (y_out, a, b);  
    parameter size = 8, delay = 15;  
    output [size-1:0] y_out;  
    input    [size-1:0] a, b;  
    wire     [size-1:0] #delay y_out; // net transport delay  
    // Other declarations, instantiations,  
    // and behavior statements go here.  
endmodule
```

- Verilog allows parameters to be overridden on an instance basis and by hierarchical dereferencing.



# Parameter Annotation

```
module modXnor (y_out, a, b);  
    parameter size = 8, delay = 15;  
    output [size-1:0] y_out;  
    input    [size-1:0] a, b;  
    wire     [size-1:0] #delay y_out = a~^b; //bitwise xnor  
endmodule
```

```
module Param;  
    modXnor G1 (y1_out, b1, c1); //Instantiation with  
                                //default parameters  
    modXnor #(4,5) G2 (y2_out, b2, c2); //Uses size = 4,  
                                        //delay = 5  
endmodule
```

# Procedural Assignments

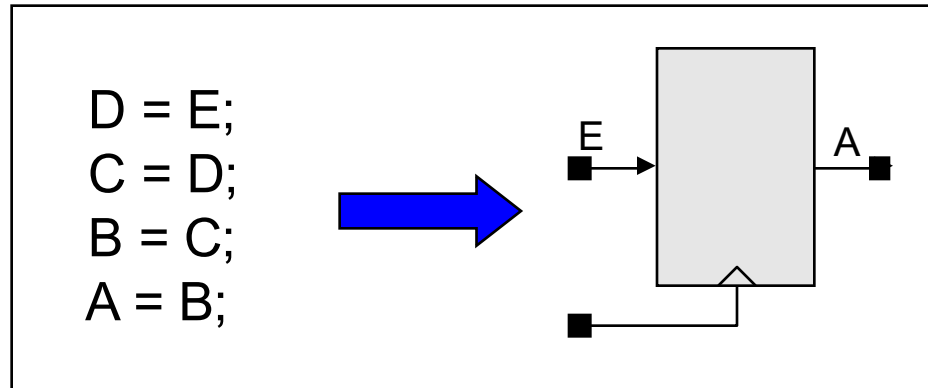
- Blocking procedural assignment (=)
- Non-blocking Procedural Assignment (<=)
- Procedural Continuous Assignment
  - `assign ... deassign`
  - `force ... release`
- LHS must be reg data type

# Nonblocking Procedural Assignment

- Evaluation and schedule of the RHS of an assignment is not blocked by the activity of preceding statements in a sequential activity flow
  - All nonblocking procedural assignments evaluate their RHS at the same time
  - Evaluated values are scheduled to assigned to LHS concurrently
- Assignment operator:  $\leq$
- Syntax:  $\langle \text{lvalue} \rangle \leq [\text{timing control}] \langle \text{expression} \rangle;$
- The outcome of executing a sequential list of nonblocking assignments is independent of the order of the list.

# Blocking Vs. Nonblocking Assignment

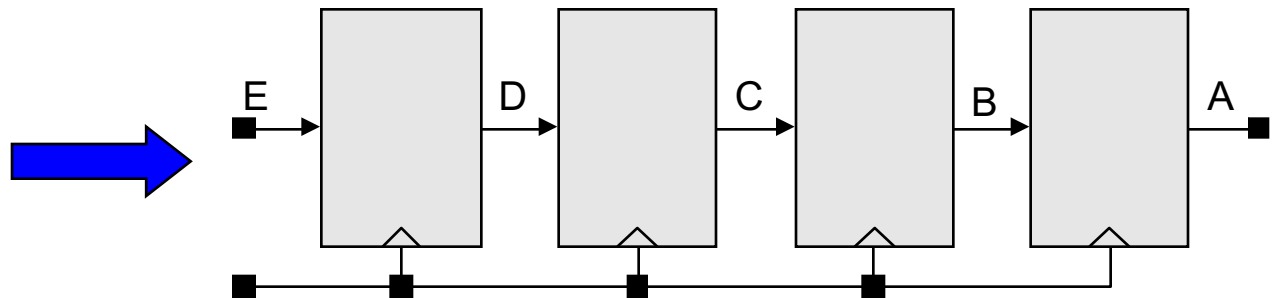
- The listed order affects the outcome of blocking assignments



D <= E;  
C <= D;  
B <= C;  
A <= B;

or

A <= B;  
B <= C;  
C <= D;  
D <= E;



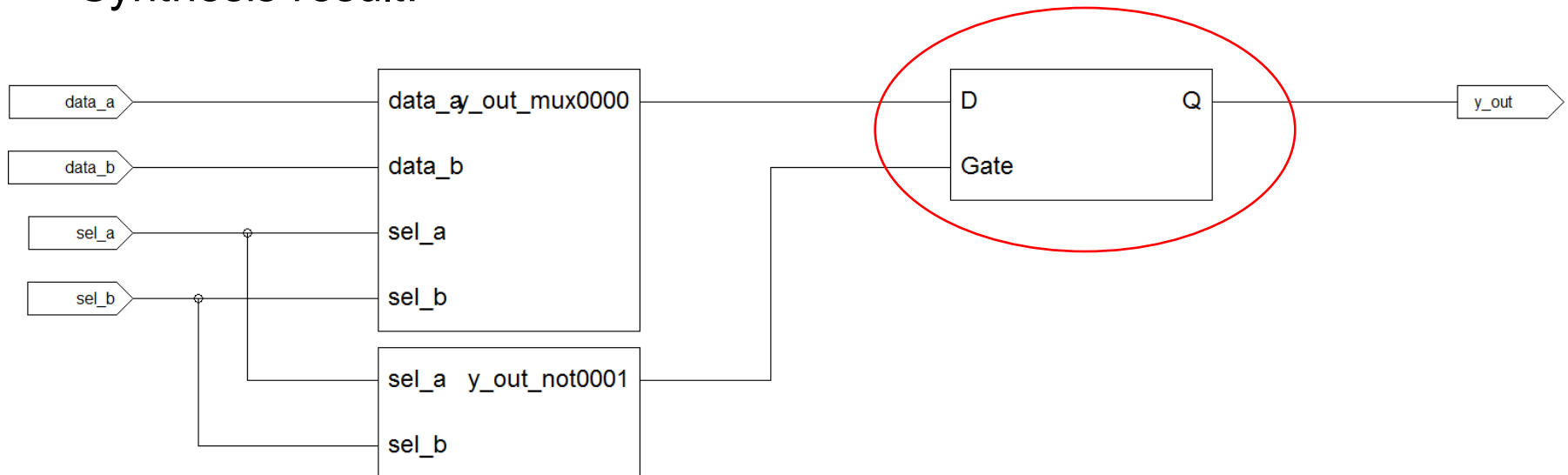
# Unwanted Latch

- Incomplete case statement or conditional branch results in latches, even for combinational circuit

- Example:

```
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
```

- Synthesis result:

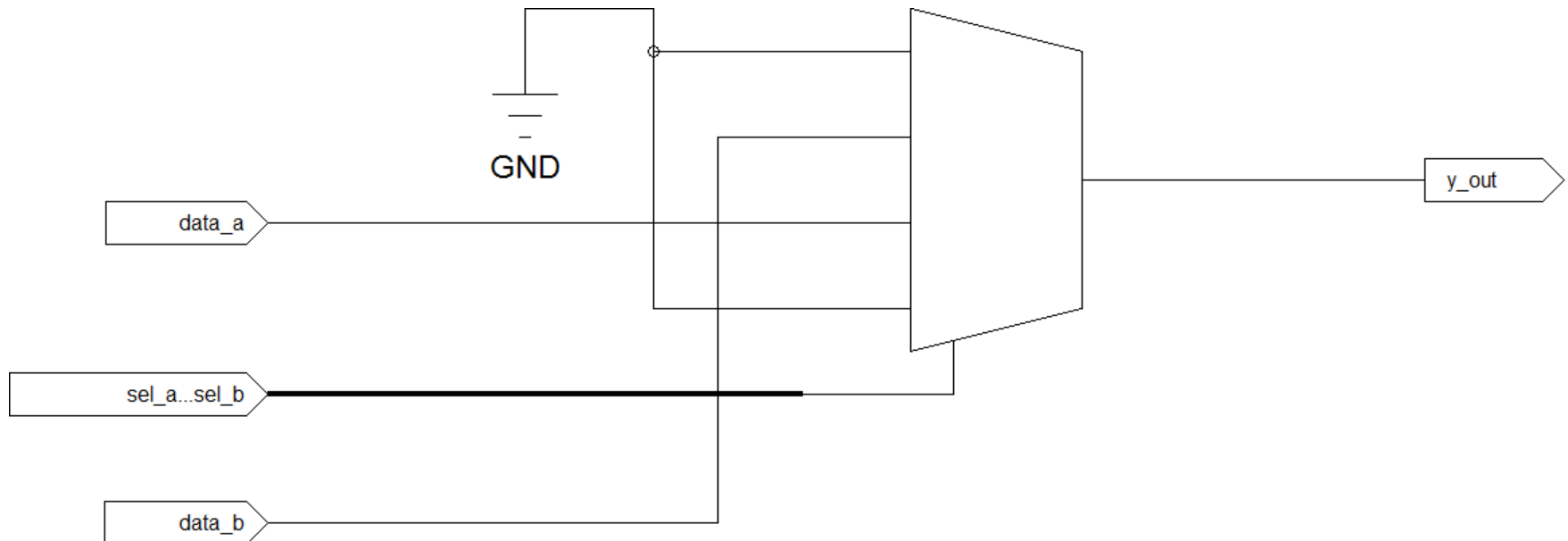


# Unwanted Latch – Fixed

- Fix

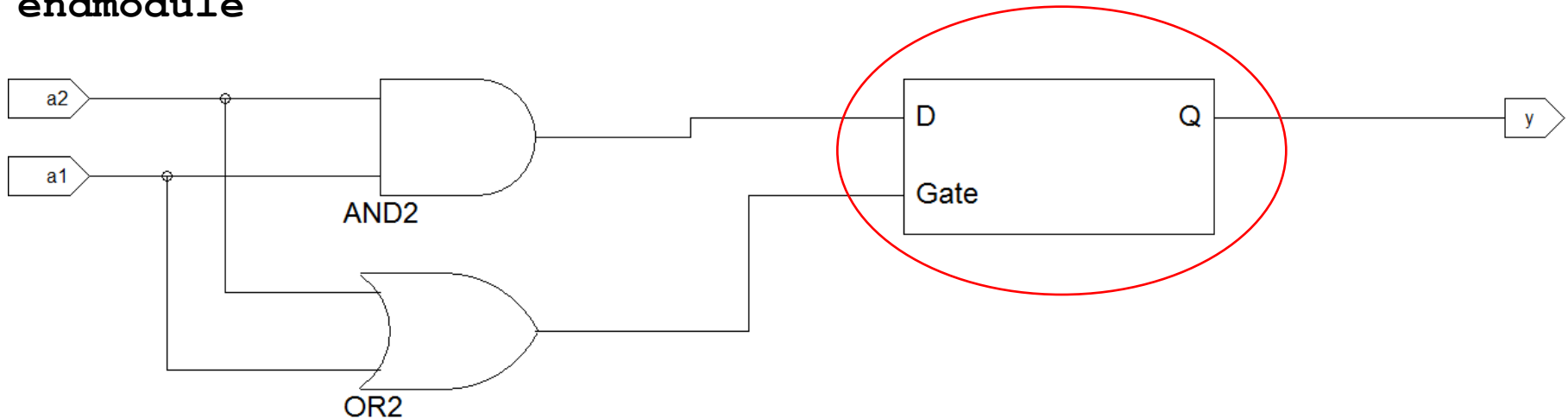
```
always @( sel_a or sel_b or data_a or data_b)  
  case ({sel_a, sel_b})  
    2'b10: y_out = data_a;  
    2'b01: y_out = data_b;  
    default y_out = 0;  
  endcase
```

- Synthesis result:



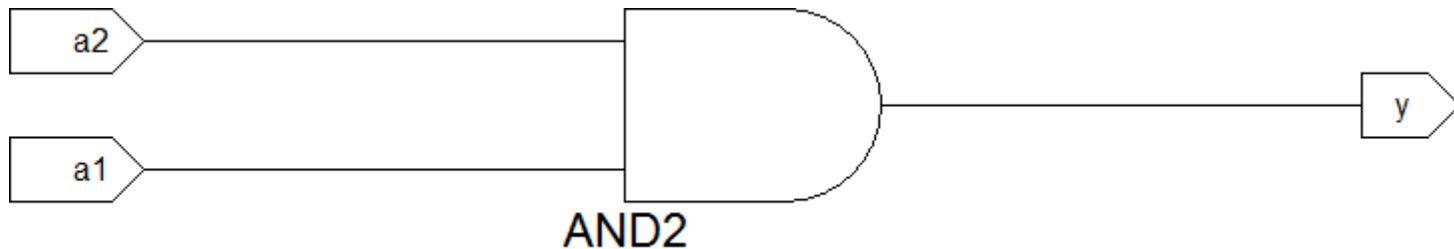
# Unwanted Latch

```
module incomplete_and (y, a1, a2);  
  input a1, a2;  
  output y;  
  reg y;  
  always @(a1 or a2)  
    if ({a2, a1} == 2'b11)      y = 1; else  
    if ({a2, a1} == 2'b01)      y = 0; else  
    if ({a2, a1} == 2'b10)      y = 0;  
endmodule
```



# Unwanted Latch – Fixed

```
module incomplete_and (y, a1, a2);  
  input a1, a2;  
  output y;  
  reg y;  
  always @(a1 or a2) begin  
    y = 0;  
    if ({a2, a1} == 2'b11)      y = 1; else  
    if ({a2, a1} == 2'b01)      y = 0; else  
    if ({a2, a1} == 2'b10)      y = 0;  
  end  
endmodule
```





# Reference

- Advanced Digital Design with Verilog HDL, 2/e, Michael Ciletti, 2010, ISBN: 978-0136019282
- IEEE Standard for Verilog HDL, [www.ieee.org](http://www.ieee.org)