# EE 445
# Computer Architecture I

2017-18 Fall

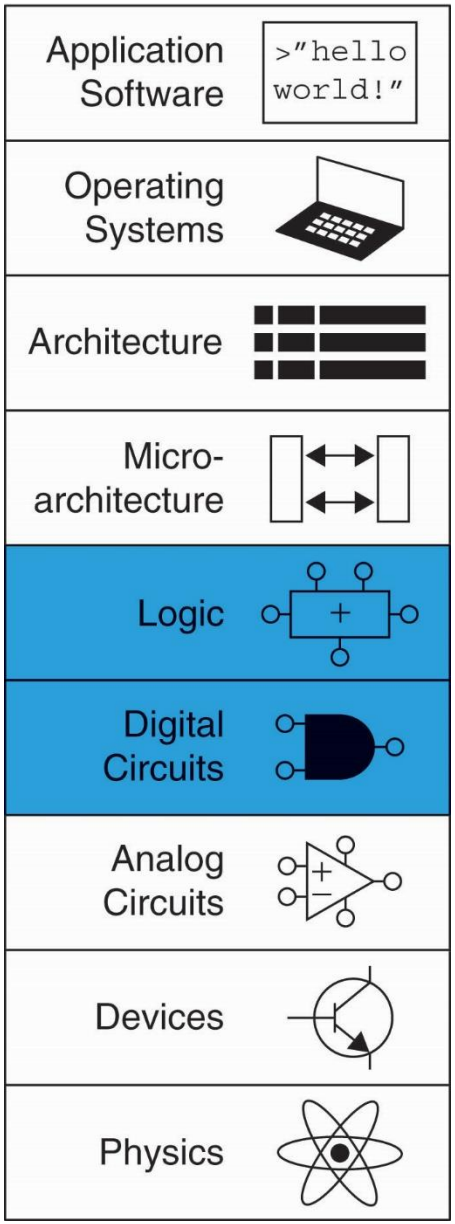
Lecture 2

(Week 3-4)

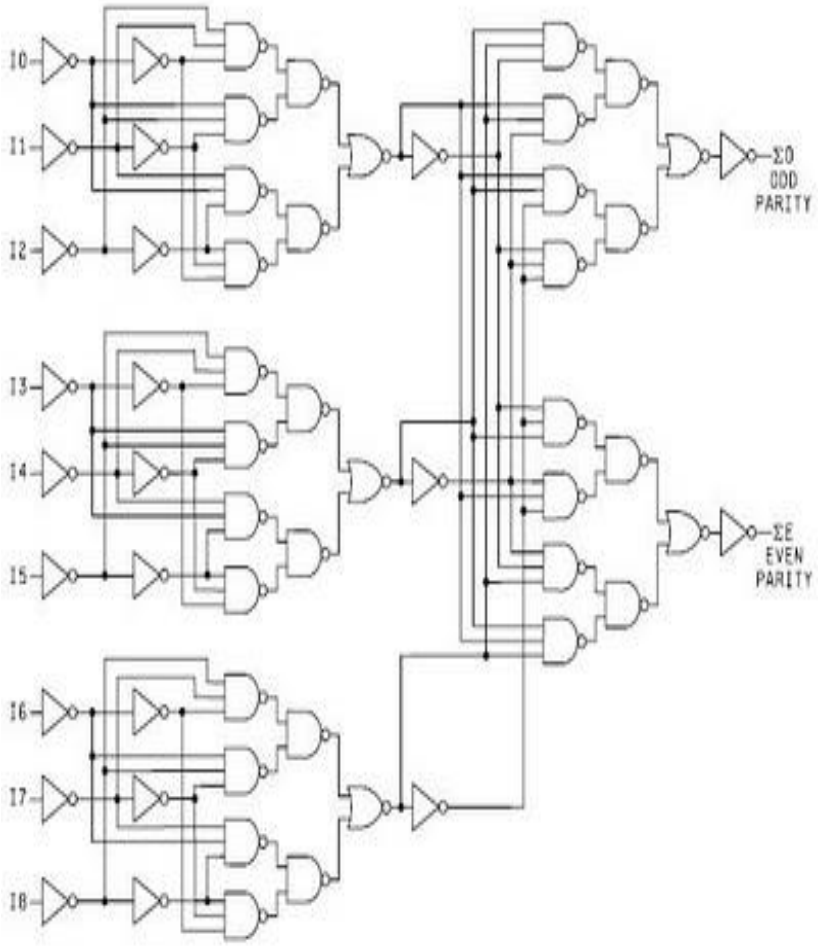
Hardware Description Languages (HDL)

Verilog HDL

# Digital Design



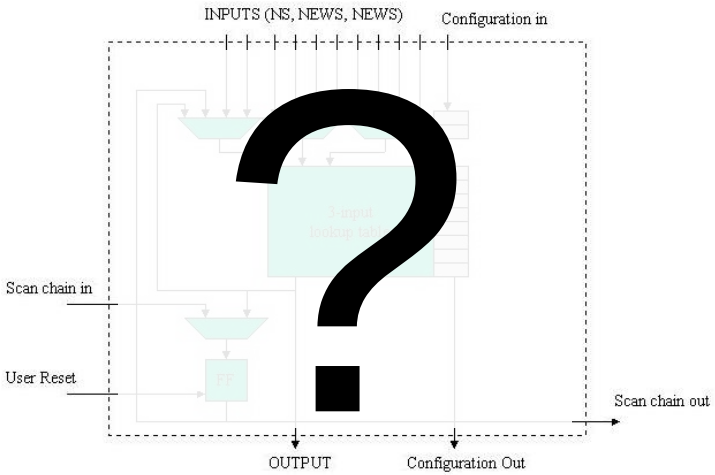
# Conventional Digital Design




- Schematic based description
- Time consuming design cycle
- Not easy to handle large projects
- Hard to modify
- Hard to debug
- Schematic is not self explaining

# Conventional Digital Design Steps

1. **Determine inputs, outputs (and states)**




2. **Build truth table (or state table)**

Full Adder Truth Table

| A | B | Cin | Cout | F |
|---|---|---|---|---|
| 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 |

# Conventional Digital Design Steps

4. Optimize function using a method such as Karnaugh map

| AB\C | $\overline{C}$ | C |
|---|---|---|
| $\overline{A}\,\overline{B}$ | 0 | 0 |
| $\overline{A}B$ | 1 | 1 |
| $AB$ | 0 | 0 |
| $A\overline{B}$ | 0 | 0 |

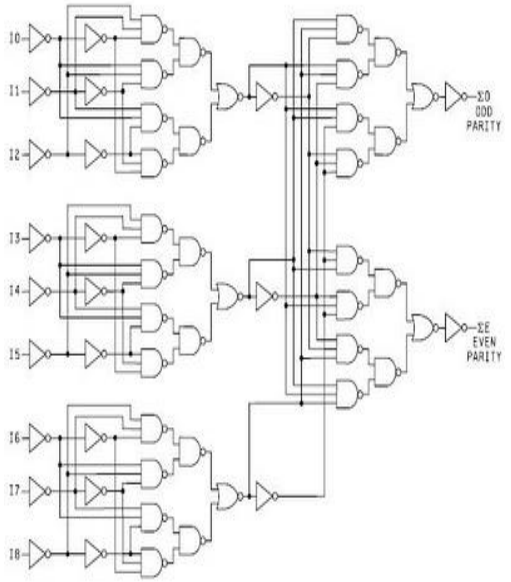
| AB\CD | $\overline{C}\,\overline{D}$ | $\overline{C}D$ | $CD$ | $C\overline{D}$ |
|---|---|---|---|---|
| $\overline{A}\,\overline{B}$ | 0 | 0 | 1 | 1 |
| $\overline{A}B$ | 0 | 0 | 0 | 0 |
| $AB$ | 0 | 0 | 0 | 0 |
| $A\overline{B}$ | 1 | 0 | 0 | 1 |

5. Derive combinational logic (and/or state) equations

$$Y = \overline{A} \cdot B + A \cdot \overline{B} + A \cdot B$$

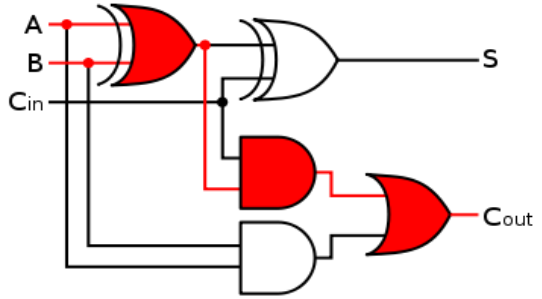
$$F = \overline{\overline{A} \cdot (B + \overline{C})} \cdot (A + \overline{B} + C) \cdot \overline{(\overline{A} \cdot \overline{B} \cdot \overline{C})}$$

6. Draw logic circuit diagram



# FPGA

- **F**ield **P**rogrammable **G**ate **A**rray
- An IC that includes millions of programmable logic gates.
- Can be programmed using :
  - Schematic logic diagram
  - Hardware description languages (VHDL, Verilog)





```
84 reading_x: process(clock)
85 file input_file_x : TEXT is IN "input_file_x_integer.txt";
86 variable in_line_x: line;
87 variable good: boolean;
88 --variable x_tmp: std_logic_vector(7 downto 0);
89 variable x_tmp: integer;
90
91 begin
92 if (clock'event and clock = '1') then
93   readline(input_file_x, in_line_x);
94   read(in_line_x, x_tmp, good);
95   x <= CONV_STD_LOGIC_VECTOR(x_tmp, 8);
96 end if;
97 end process;
```

# *HDLs*

- HDL is NOT an ordinary programming language!
- It is a language to descrine “**hardware**”
- It is used to describe logic circuits in textual forms.
- Verilog and VHDL are the two most popular HDLs.

# *The Purpose of HDLs*

- Simplify design representation so you can concentrate on the overall picture
- HDL allows description in language rather than schematic, speeding up development time
- Allows reuse of components (standard cells i.e. libraries)

# *VHDL or Verilog or SystemVerilog?*

- Hardware description language (HDL)
  - specifies logic function only
  - Computer-aided design (CAD) tool produces or *synthesizes* the optimized gates
- Most commercial designs built using HDLs
- Two leading HDLs:
  - **Verilog**
    - Developed in 1984 by Gateway Design Automation
    - IEEE standard (1364) in 1995
    - Extended in 2005 (IEEE STD 1800-2009) with name ***SystemVerilog***
  - **VHDL 2008**
    - Developed in 1981 by the Department of Defense
    - IEEE standard (1076) in 1987
    - Updated in 2008 (IEEE STD 1076-2008)

# *VHDL or Verilog?*

- Verilog is C-like and liked by electrical and computer engineers since most learn C language in university.
- VHDL (**V**ery High Speed Integrated Circuit **HDL**) is ADA-like and most engineers have no experience with ADA.
- Verilog is used extensively in South-East Asia, India and America while VHDL is used mostly in Europe and Japan.

# *HDL to Gates*

The two major purposes of HDLs are logic simulation and synthesis.

- **Simulation**
  - Inputs applied to circuit
  - Outputs checked for correctness

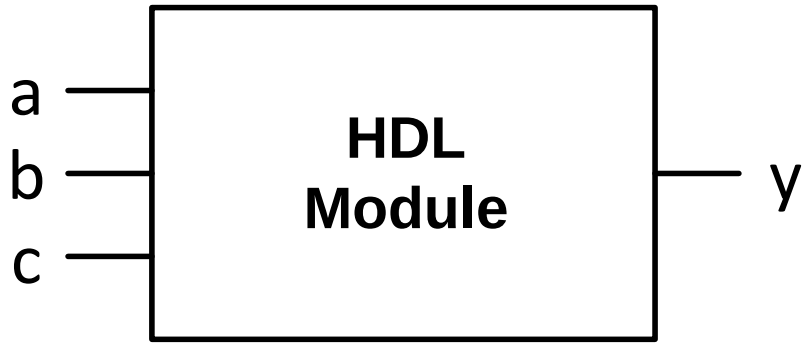
  Millions may be saved by debugging in simulation instead of hardware

- **Synthesis**
  - Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

# *HDL to Gates*

- HDL is not as a computer program but a shorthand for describing digital hardware.
- If you don’t know approximately what hardware your HDL should synthesize into, you probably won’t like what you get.
  - You might create far more hardware than is necessary, or
  - you might write code that simulates correctly but cannot be implemented in hardware.
- Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines.
- Sketch these blocks on paper and show how they are connected before you start writing code.

# HDL Modules

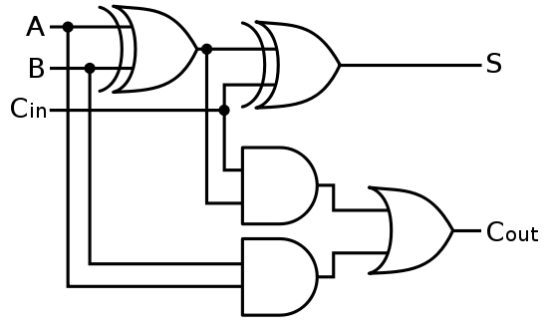



**Three types of Modules:**

- **Structural (Gate):** describe how it is built from simpler modules
- **Data Flow (RTL):** specify the characteristics of a circuit by operations and transfer of data between registers
- **Behavioral (Algorithmic):** describe what a module does as a concurrent algorithm

# Abstraction Levels

- **Structural (Gate) Level:**

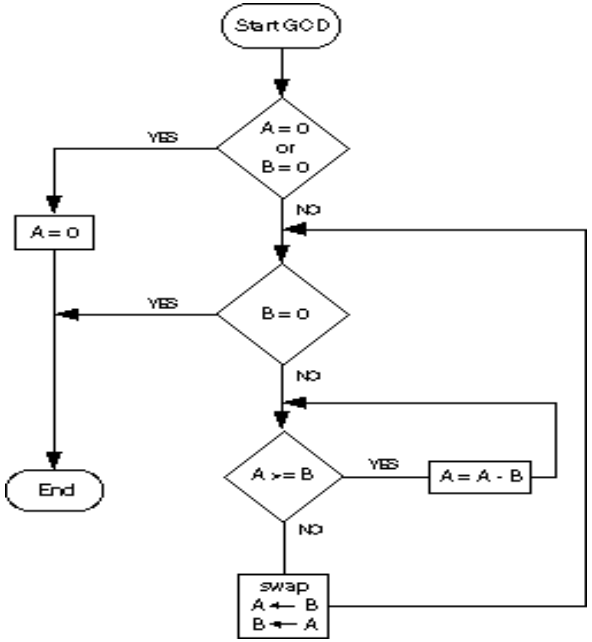



- **Data Flow (RTL) Level:**

$$Y = \overline{A} \cdot B + A \cdot \overline{B} + A \cdot B$$

$$F = \overline{\overline{A} \cdot (B + \overline{C})} \cdot (A + \overline{B} + C) \cdot \overline{(\overline{A} \cdot \overline{B} \cdot \overline{C})}$$

- **Behavioral (Algorithmic) Level:**



# Modeling to Synthesis

## Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

# Modeling to Synthesis

## Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

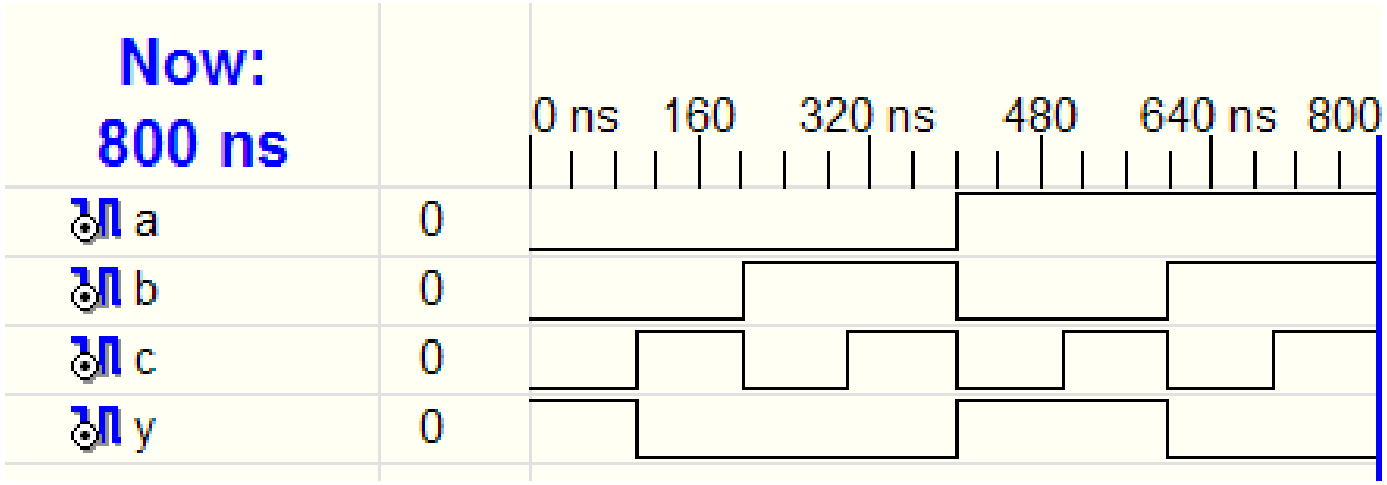
- module/endmodule: required to begin/end module
- example: name of the module
- Operators:
  - ~: NOT
  - &: AND
  - |: OR

# Modeling to Synthesis

## Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

## Simulation:

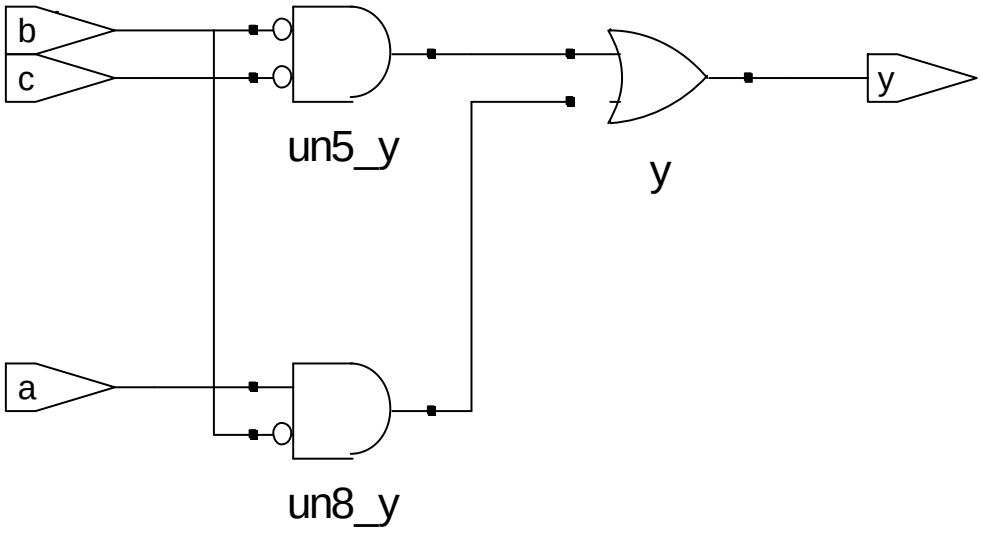


# Modeling to Synthesis

## Verilog:

```
module example(input a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

## Synthesis:

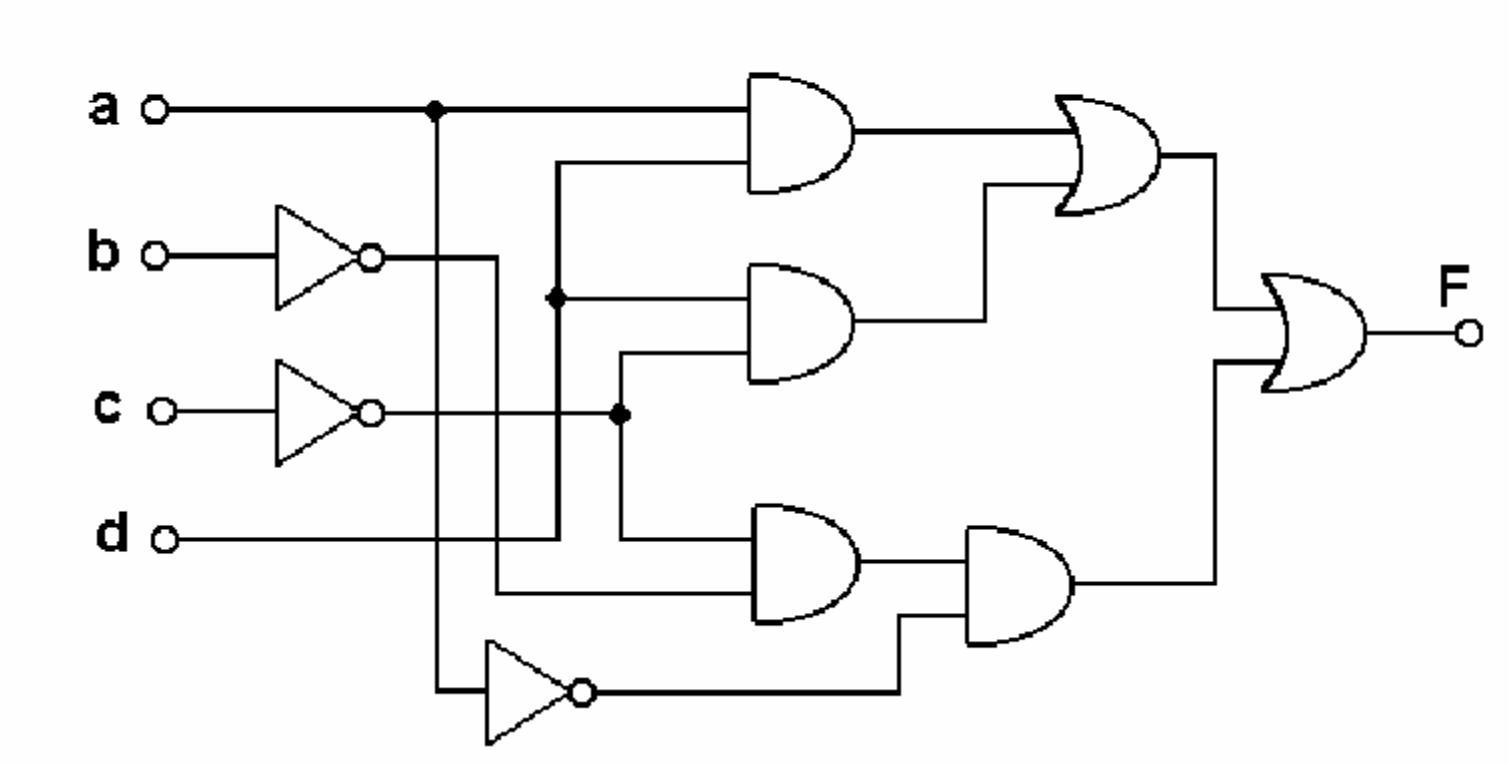


# *Verilog Syntax*

- There are about 100 keywords which are written with lowercase letters.
  - **input**, **output**, **wire**, **reg**, **and**, **or**, **module**, **tri,** etc.
- Case sensitive
  - “Button1” is different than “button1”
- No names start with numbers
  - Example: `2mux` is an invalid name
- Comments:
  - //Single line comment,  /* Block or multiline comment */
- Every line ends with a semicolon ‘’**;”**
- Whitespace ignored.

# Gate Level Description

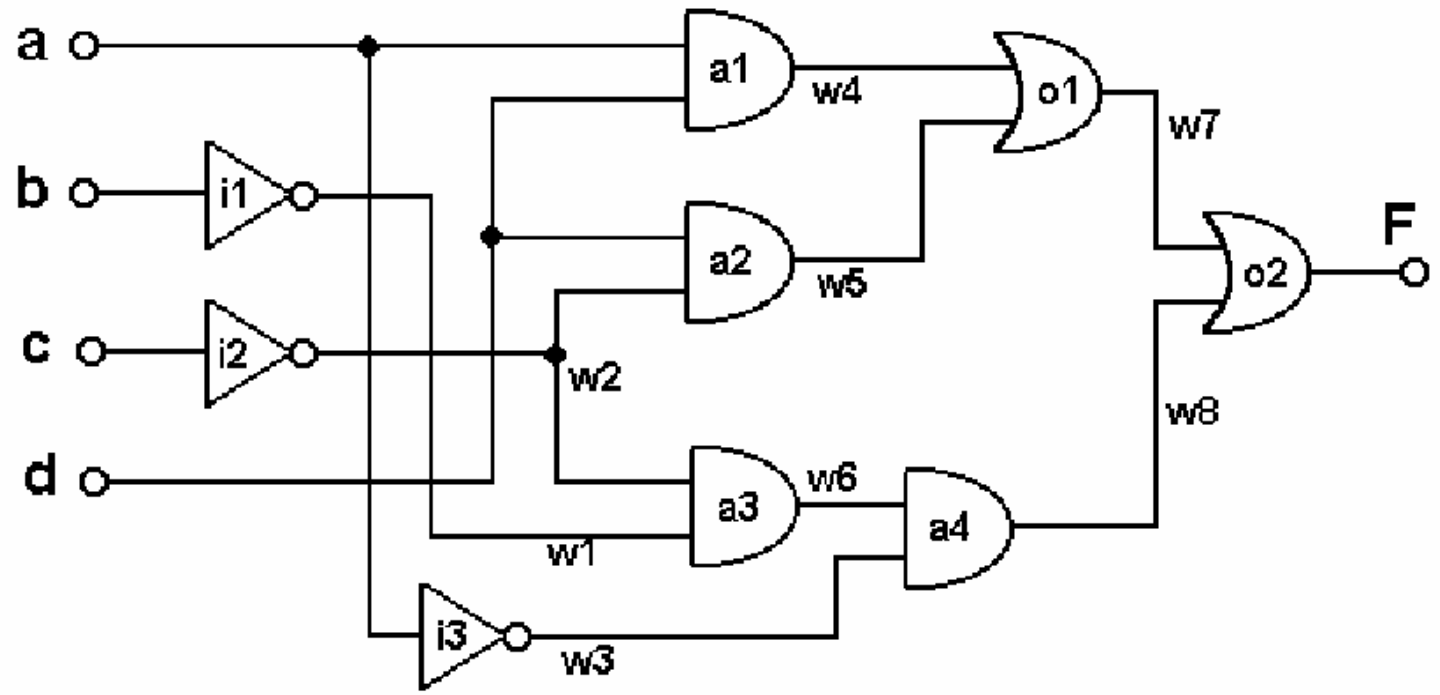
- Describe this circuit…



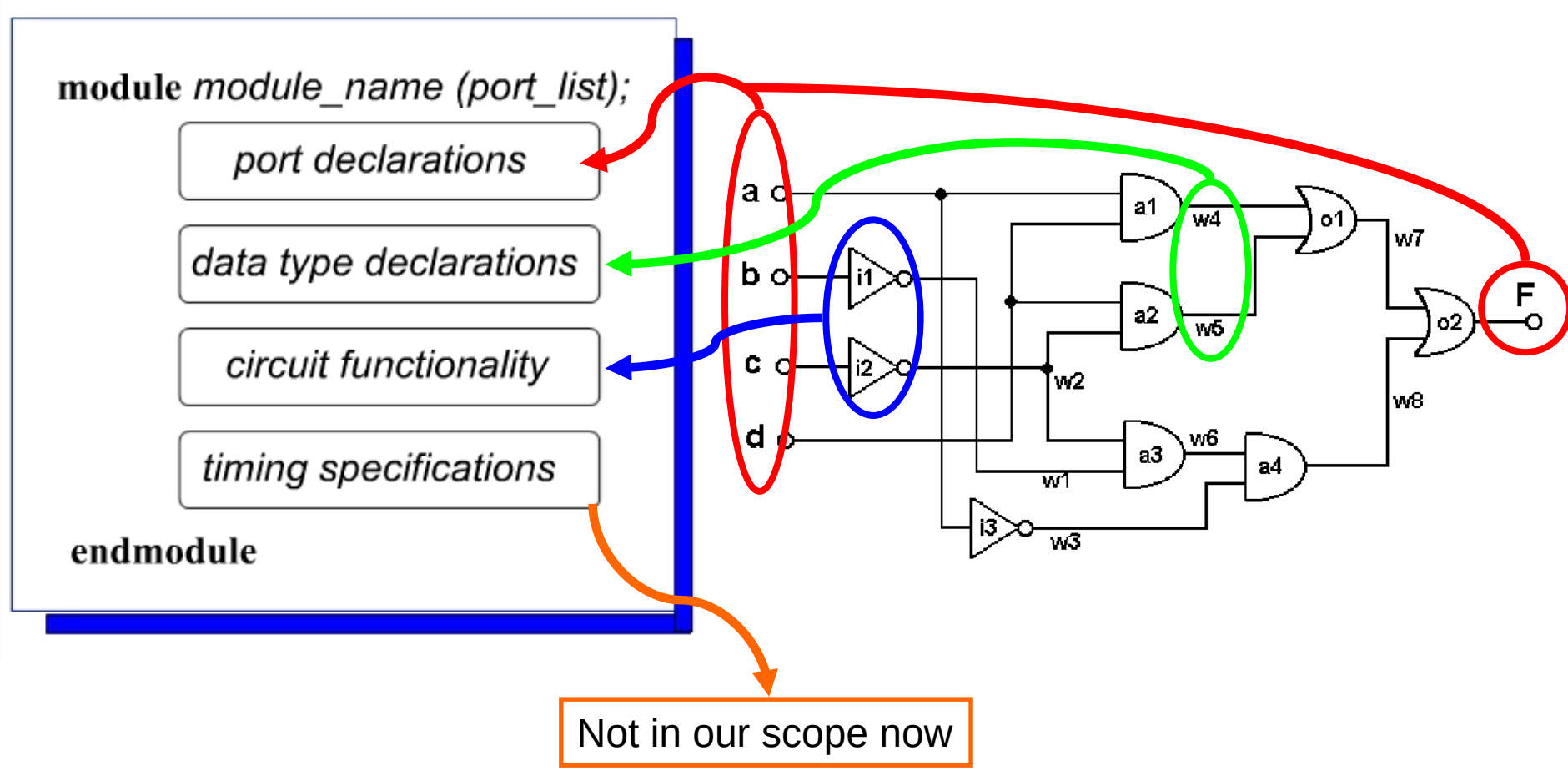
# Gate Level Description

**Underlying Method:** Specify inputs/outputs, gates and how they are connected to each other:

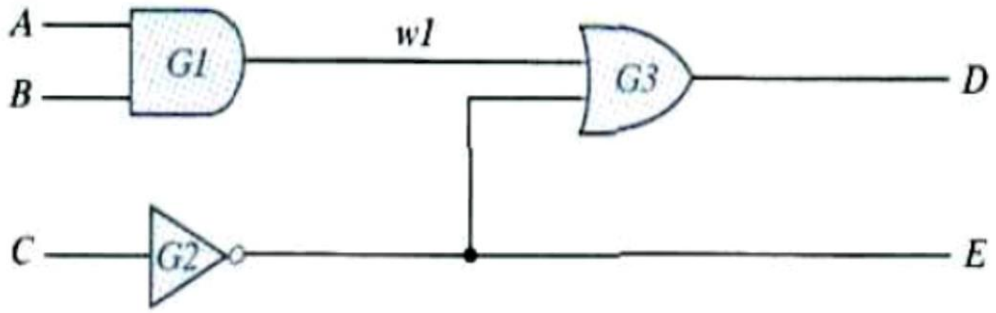
```
and     a1(w4,a,d);
not     i1(w1,b);
…
```



# Verilog - Modules



# Gate Level Description - Example1




**HDL Example** **(Combinational logic modeled with primitives)**

// Verilog model IEEE 1364–1995 Syntax

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

**module** *module_name (port_list);*

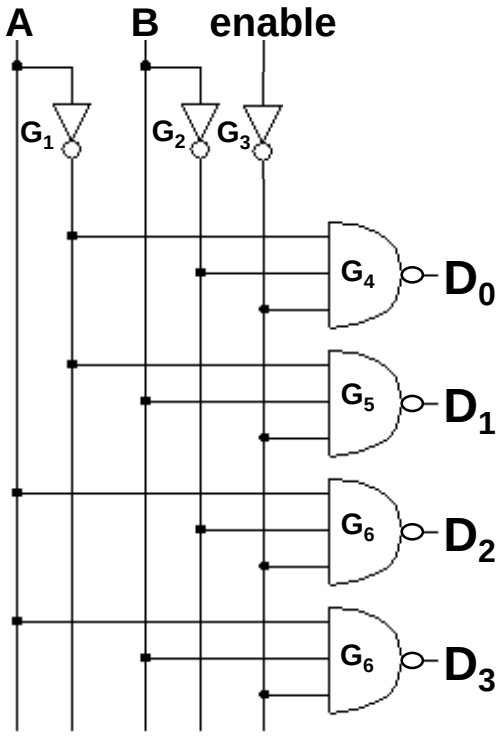
- *port declarations*
- *data type declarations*
- *circuit functionality*
- *timing specifications*

**endmodule**

# Gate Level Description – Example2

**HDL Example**

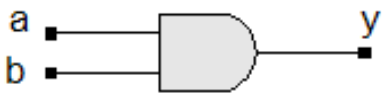
```
// Gate-level description of two-to-four-line decoder

module decoder_2x4_gates (D, A, B, enable);
  output [3:0]          D;
  input                 A, B;
  input                 enable;
  wire                  A_not, B_not, enable_not;

  not
    G1  (A_not, A),
    G2  (B_not, B),
    G3  (enable_not, enable);
  nand
    G4  (D[0], A_not, B_not, enable_not),
    G5  (D[1], A_not, B, enable_not),
    G6  (D[2], A, B_not, enable_not),
    G7  (D[3], A, B, enable_not);
endmodule
```



# Verilog – Data Representation

## 1. Logic:

- **0** represent a logic '0' or false condition
- **1** represent a logic '1' or true condition
- **Z** Output of an undriven tri-state driver – High-Z value
- **x** Models when un-initialized or unknown logic value
  - Initial state of registers
  - Output of a gate with **z** inputs




| and | 0 | 1 | X | Z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | X | X |
| X | 0 | X | X | X |
| Z | 0 | X | X | X |

# Verilog – Data Representation

## 2. Number:

- Format: **<size>'<base_format><number>**
- **<size>** - decimal specification of bits count
- **<base_format>** - **'** followed by arithmetic base of number

  **d** or **D** – decimal (default if no base format given)

  **h** or **H** – hexadecimal

  **o** or **O** – octal

  **b** or **B** – binary

  **Ex:** 8'b1111_0000

  8'hF0

  8'd127

# *Data Types*

Two groups of data types:

- **net data types:**
  - should be driven.
  - the value changes when the driver changes value.
  - e.g. **wire, supply0, supply1, tri, triand, trior, tri0, tri1, ...**

- **variable data types**
  - change value upon assignment and holds its value until another assignment.
  - e.g. **integer**, **real**, **realtime**, **reg**, **time**, ...

# Truth Table of Primitive Gates

*Truth Table for Predefined Primitive Gates*

| and | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

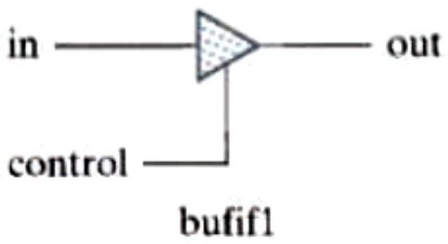
| or | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | x | 1 | x | x |

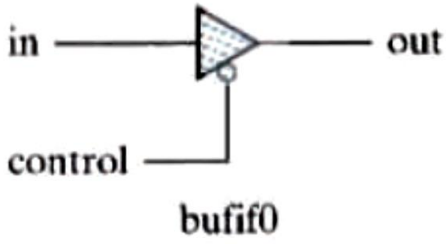
| xor | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 0 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

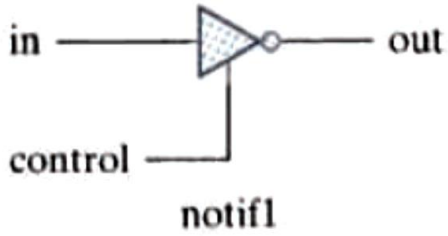
| not | input | output |
|---|---|---|
| | 0 | 1 |
| | 1 | 0 |
| | x | x |
| | z | x |

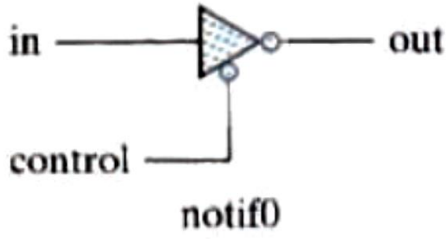
# Three-State Gates

*gate name (output, input, control);*



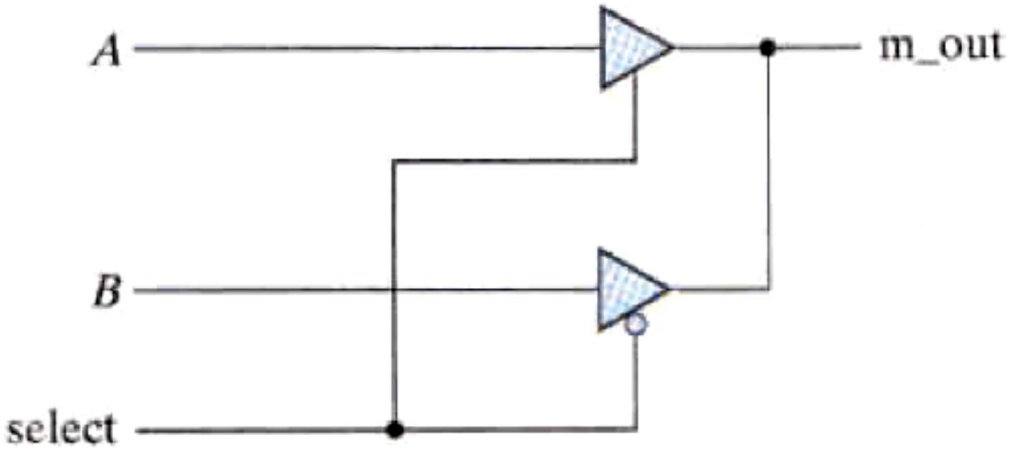




**bufif1** (OUT, A, control); input $A$ is transferred to $OUT$ when $control = 1$. $OUT$ goes to **z** when $control = 0$.

**notif0** (Y, B, enable); output $Y =$ **z** when $enable = 1$ output $Y = B'$ when $enable = 0$.

# Gate Level Description - Example3



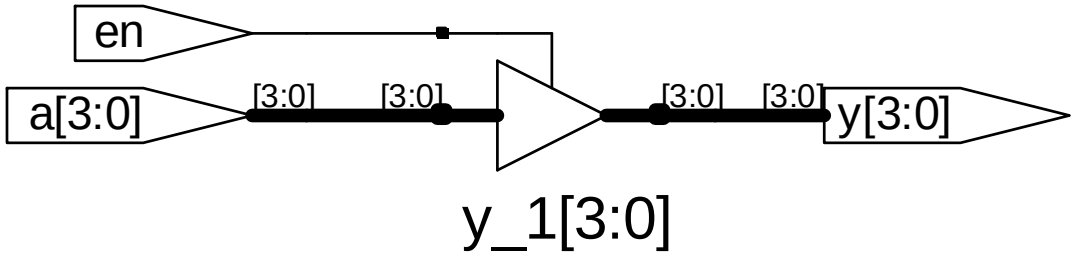

```
// Mux with three-state output
module mux_tri (m_out, A, B, select);
  output m_out;
  input  A, B, select;
  tri    m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

# Z: Floating Output

## Verilog:

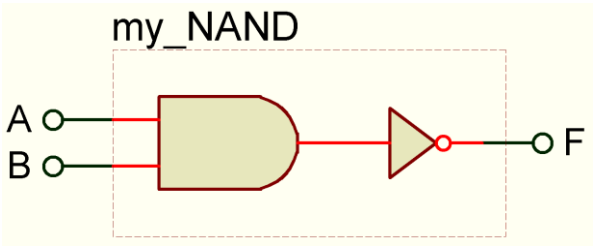
```
module tristate(input [3:0] a,
                input  en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

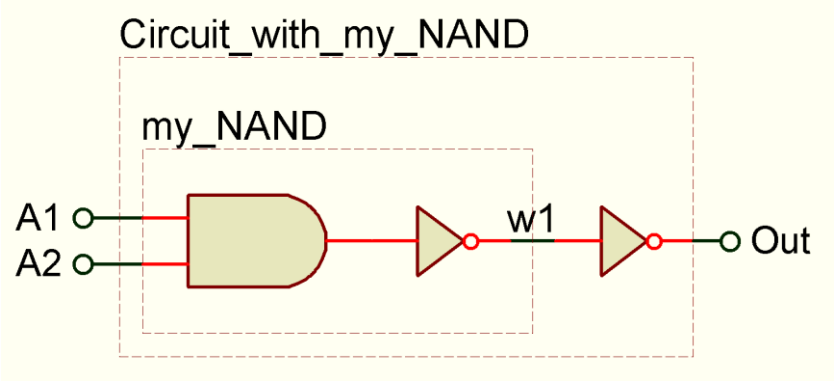
## Synthesis:



# User Defined Primitive (UDP)

```
// An example UDP: “My NAND Gate”
primitive my_NAND(F,A,B);
output  F;
input   A,B;
wire    and_out;

and     A1(and_out,A,B);
not     N1(F,and_out);
endprimitive

// AND from my_NAND:
module Circuit_with_my_NAND(Out,A1,A2);
wire    w1;
my_NAND         (w1,A1,A2);
not             (Out,w1);
endmodule
```



# Data Flow Level Description

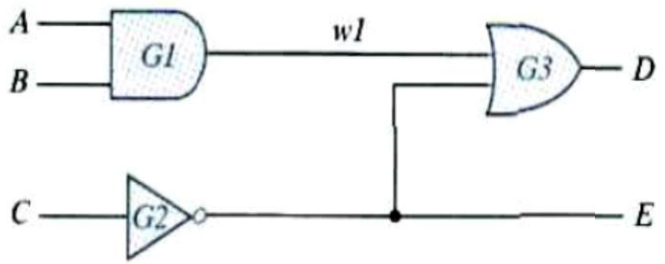
- Data flow modelling is used for describing the Boolean equations of combinational circuits.
- The following operators are used to describe functions:

**Table 4.10**
*Some Verilog HDL Operators*

| Symbol | Operation |
|---|---|
| + | binary addition |
| − | binary subtraction |
| & | bitwise AND |
| \| | bitwise OR |
| ^ | bitwise XOR |
| ~ | bitwise NOT |
| == | equality |
| > | greater than |
| < | less than |
| { } | concatenation |
| ? : | conditional |

# Data Flow Level Description - Example1

```
assign D = (A & B)|~C;
```




$$E = A + BC + B'D$$

$$F = B'C + BC'D'$$

**HDL Example (Combinational logic modeled with Boolean equations)**

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
 output      E, F;
 input       A, B, C, D;

 assign E = A | (B & C) | (~B & D);
 assign F = (~B & C) | (B & ~C & ~D);
endmodule
```

Continuous assignment statements end with a semicolon.

Anytime the inputs on the right side change, the output on the left side is recomputed.

Continuous assignment statements describe combinational logic.

# *Assign*

All statements are concurrent except **initial** or **always** blocks.

**assign** sum=a+b+c;
**assign** sum= a+b;
**assign** sum=sum+c;

sum value becomes **indeterminate**

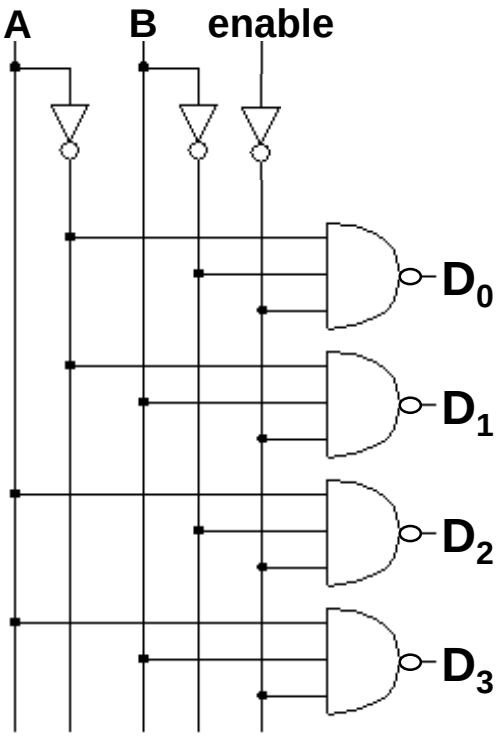
- if, case, for, while loops should appear inside **initial** or **always** blocks .

# Data Flow Level Description – Example2

**HDL Example**

```
// Dataflow description of two-to-four-line decoder

module decoder_2x4_df (                    // Verilog 2001, 2005 syntax
  output      [0: 3]      D,
  input                   A, B,
                          enable
);

  assign      D[0] = ~(~A & ~B & ~enable),
              D[1] = ~(~A & B & ~enable),
              D[2] = ~(A & ~B & ~enable),
              D[3] = ~(A & B & ~enable);
endmodule
```

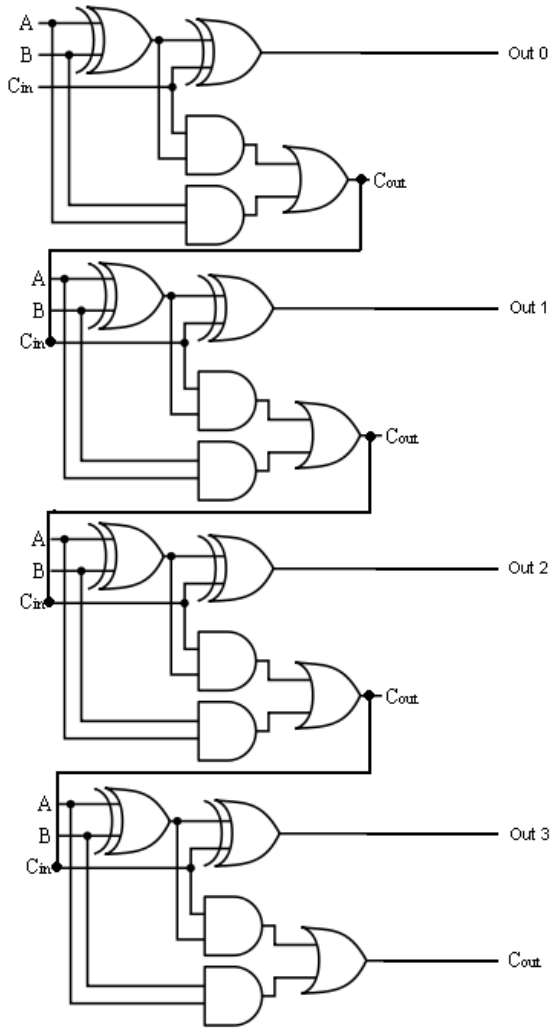
```
not
  G1  (A_not, A),
  G2  (B_not, B),
  G3  (enable_not, enable);
nand
  G4  (D[0], A_not, B_not, enable_not),
  G5  (D[1], A_not, B, enable_not),
  G6  (D[2], A, B_not, enable_not),
  G7  (D[3], A, B, enable_not);
```



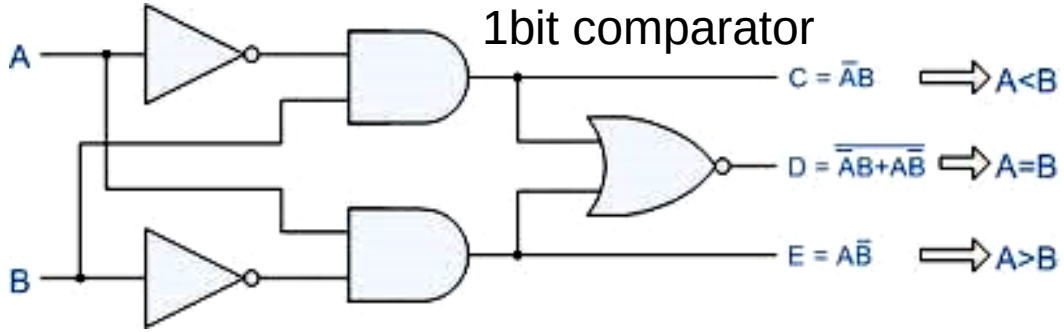
# Data Flow Level Description – Example3

**HDL Example**

```
// Dataflow description of four-bit adder

// Verilog 2001, 2005 module port syntax

module binary_adder (
  output [3: 0]          Sum,
  output                 C_out,
  input  [3: 0]          A, B,
  input                  C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```



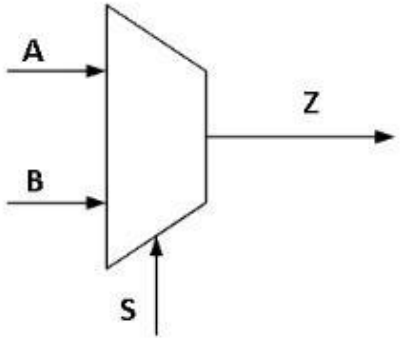
# Data Flow Level Description – Example4

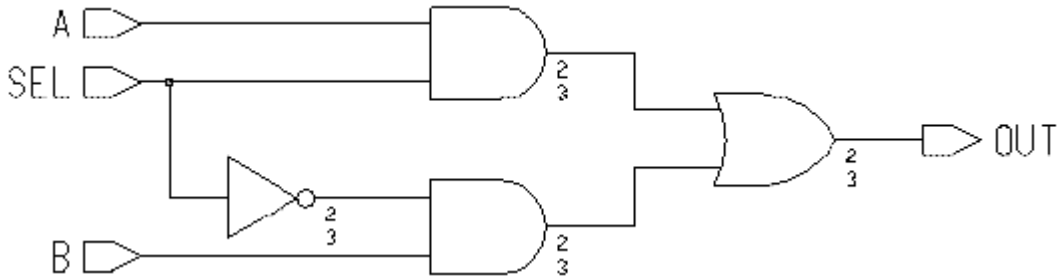



**HDL Example**

```
// Dataflow description of a four-bit comparator      //V2001, 2005 syntax

module mag_compare
( output              A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]        A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```

# Data Flow Level Description – Example5




**HDL Example**

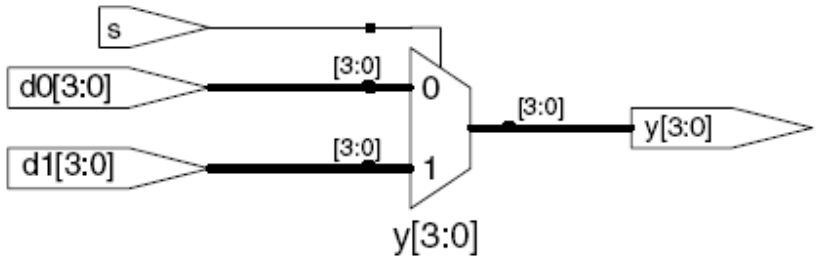
```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output        m_out;
  input         A, B;
  input         select;

  assign m_out = (select)? A : B;
endmodule
```

| ? : | conditional |
|---|---|

# Conditional Assignment

```
module mux2(input[3:0] d0, d1,
            input s,
            output[3:0] y);
   assign y = s ? d1 : d0;
endmodule
```

**Synthesis:**



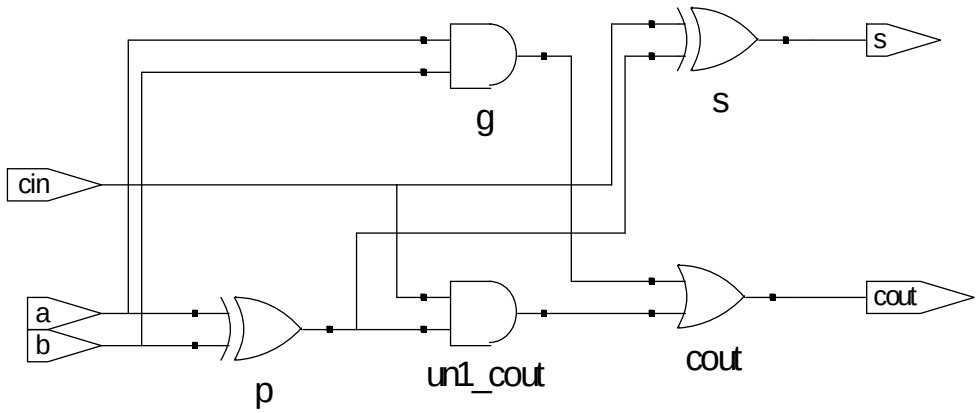

**? :** is also called a *ternary operator* since it operates on 3 inputs: `s`, `d1`, and `d0`.

# Internal Variables

## Verilog:

```
module fulladder(input  a, b, cin,
                 output s, cout);
p, g;    // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

## Synthesis:

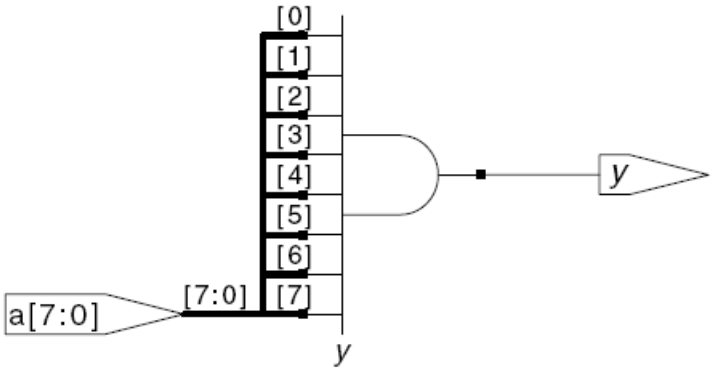


# Reduction Operators

## SystemVerilog:

```
module and8(input [7:0] a,
            output y);
   assign y = &a;

   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];

endmodule
```

## Synthesis:



# Precedence

| | | |
|---|---|---|
| Highest | ~ | NOT |
| | *, /, % | mult, div, mod |
| | +, - | add,sub |
| | <<, >> | shift |
| | <<<, >>> | arithmetic shift |
| | <, <=, >, >= | comparison |
| | ==, != | equal, not equal |
| | &, ~& | AND, NAND |
| | ^, ~^ | XOR, XNOR |
| | \|, ~\| | OR, NOR |
| Lowest | ?: | ternary operator |

# Numbers

## Format: N'Bvalue

**N** = number of bits, **B** = base

**N'B** is optional but recommended (default is decimal)

| Number | # Bits | Base | Decimal Equivalent | Stored |
|---|---|---|---|---|
| 3'b101 | 3 | binary | 5 | 101 |
| 'b11 | unsized | binary | 3 | 00…0011 |
| 8'b11 | 8 | binary | 3 | 00000011 |
| 8'b1010_1011 | 8 | binary | 171 | 10101011 |
| 3'd6 | 3 | decimal | 6 | 110 |
| 6'o42 | 6 | octal | 34 | 100010 |
| 8'hAB | 8 | hexadecimal | 171 | 10101011 |
| 42 | Unsized | decimal | 42 | 00…0101010 |

# Bit Manipulations: Example 1

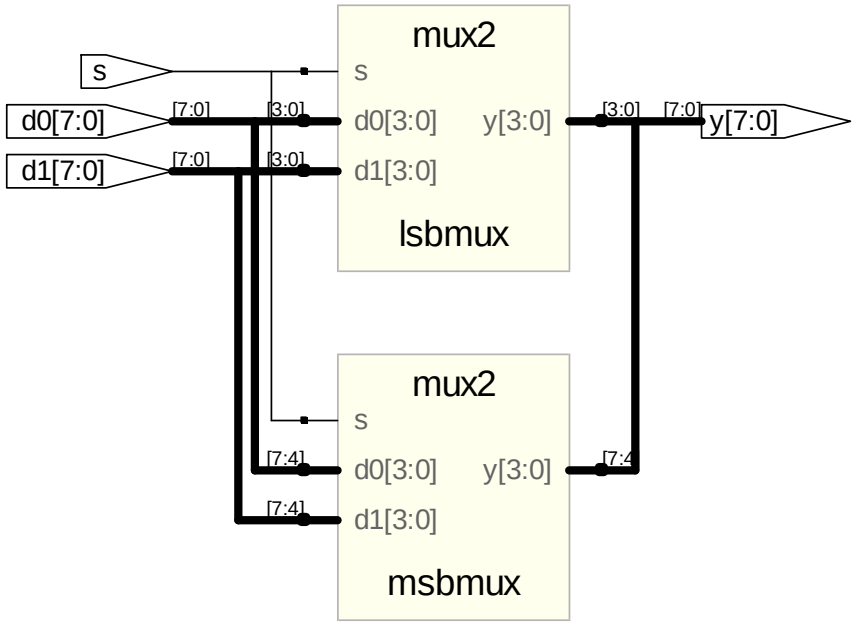
```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// if y is a 12-bit signal, the above statement produces:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

// underscores (_) are used for formatting only to make
// it easier to read. Verilog ignores them.
```

# Bit Manipulations: Example 2

## Verilog:

```
module mux2_8(input[7:0] d0, d1,
              input      s,
              output[7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

## Synthesis:

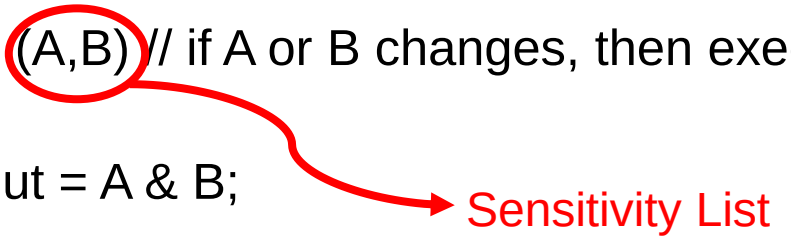


# Behavioural Description &"Always" Blocks

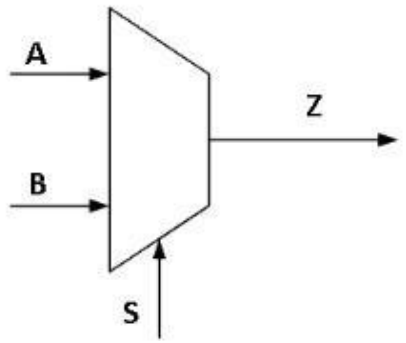
- "always" block executes always.
- The @ symbol indicates that the block will be triggered "at" the condition in parenthesis.

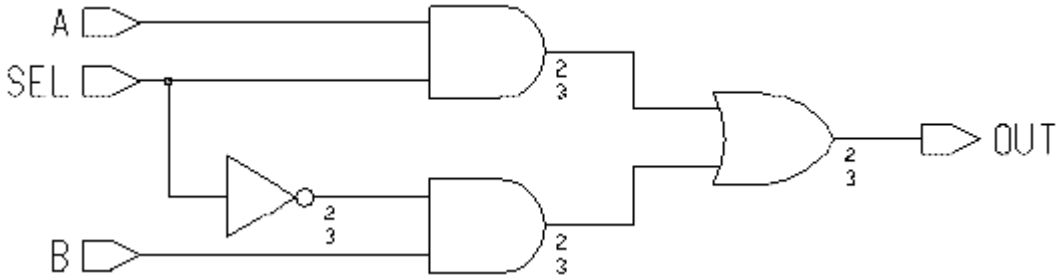
```
// C analogy:
while(1)
{
        if(condition)
                do operation
}
```

```
//Demonstrates the use of "always" block:
module always_and_gate(Out,A,B);

always @ (A,B) // if A or B changes, then execute the following:
begin
        Out = A & B;
end

endmodule
```



# Behavioral Description – Example1




**HDL Example**

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output      m_out;
  input       A, B, select;
  reg         m_out;

  always      @(A or B or select)
    if (select == 1) m_out = A;
    else m_out = B;
  endmodule
```

# Behavioral Description – Example2

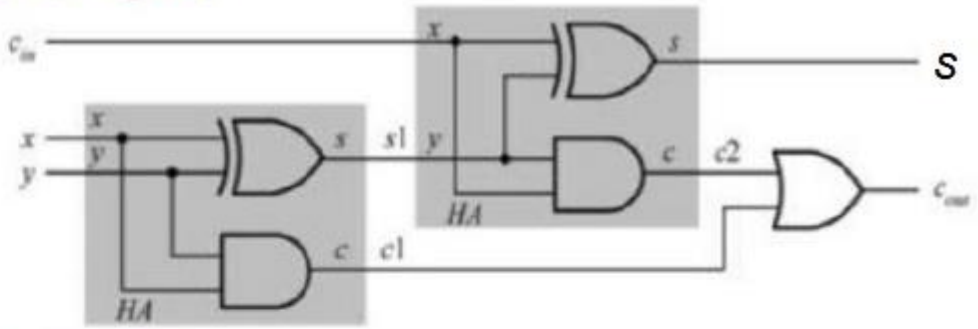
**HDL Example**

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg  m_out,
  input       in_0, in_1, in_2, in_3,
  input [1: 0] select
);
  always @ (in_0, in_1, in_2, in_3, select)        // Verilog 2001, 2005 syntax
    case (select)
      2'b00:              m_out = in_0;
      2'b01:              m_out = in_1;
      2'b10:              m_out = in_2;
      2'b11:              m_out = in_3;
    endcase
  endmodule
```

# Design using different descriptions

## FA using 2 HAs

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input    x, y, c_in;
output   s, c_out;
reg      c_out;
wire     s1, c1, c2;
// structural modeling of the first half adder HA 1.
   xor xor_ha1(s1, x, y);
   and and_ha1(c1, x, y);
// dataflow modeling of the second half adder HA 2.
   assign s = c_in ^ s1;
   assign c2 = c_in & s1;
// behavioral modeling
always @(c1, c2)   // also can use always @(*)
     c_out = c1 | c2;
endmodule
```

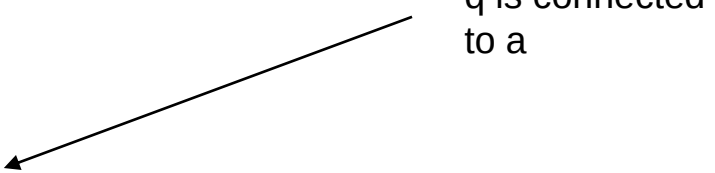


# Hierarchical Design

```
module bottom1(a, b, c);
  input a, b;
  output c;
  reg c;

  always
    begin
      c <= a & b;
    end
endmodule
```

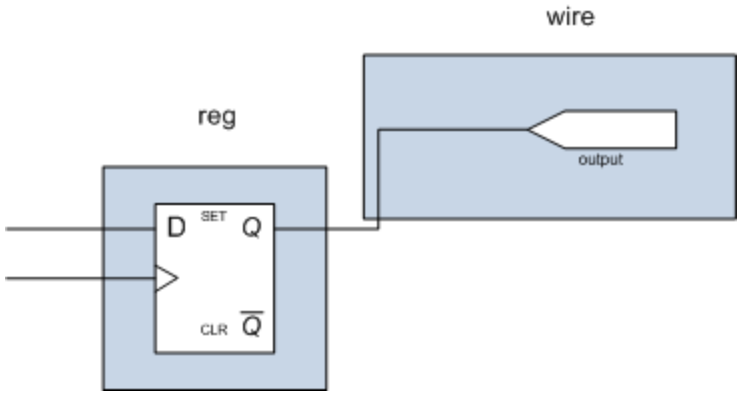
```
module bottom2(l, m, n);
  input l, m;
  output n;
  reg n;

  always
    begin
      n <= l | m;
    end
endmodule
```

```
module top_ver (q, p, r, out);
  input q, p, r;
  output out;
  reg out, intsig;

  bottom1 u1(.a(q), .b(p), .c(intsig));
  bottom2 u2(.l(intsig), .m(r), .n(out));
endmodule
```



# “wire” and “reg”

- **wire**
  - Represents a wire.
  - Cannot store a value.
  - Used for connection of blocks and/or registers.
  - Used to model wires in combinational circuits.
- **reg**
  - Represents a register (DFF).
  - Can store value.
  - Use to model regs in sequential circuits.



# Assignments

- **continuous**
  - Drives values into the nets.
  - **assign** operation

  Continuous assignments to wire
  assign variable = exp;

  **wire** out;
  - **assign** out = n_A & In_B;

  Results in combinational logic

- **procedural**
  - Enable updating registers
  - Can be used only within the structured procedures (always, initial, task, function)
  - Procedural assignment statements:
    - blocking
    - non-blocking.

# *Procedural Assignments*

- Procedural assignment evaluation can be modeled as:
  - Blocking
  - Non-blocking
- Procedural assignment execution can be modeled as:
  - Sequential
  - Concurrent
- Procedural assignment timing controls can be modeled as:
  - Delayed evaluations
  - Delayed assignments

# Blocking vs Non-Blocking

## Blocking

- <variable> **=** <statement>
- Similar to C code
- The next assignment waits until the present one is finished
- Used for combinational logic

## Non-blocking

- <variable> **<=** <statement>
- The inputs are stored once the procedure is triggered
- Statements are executed in parallel
- Used for flip-flops, latches and registers (sequential circuits)

**Do not mix both assignments in one procedure**

# Procedural Assignments

- *Blocking assignment:* evaluation and assignment are immediate

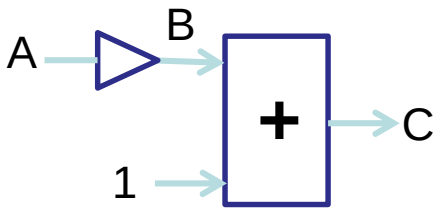
```
always @ (a or b or c)
begin
   x = a | b;          1. Evaluate a | b, assign result to x
   y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
   z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
   x <= a | b;         1. Evaluate a | b but defer assignment of x
   y <= a ^ b ^ c;     2. Evaluate a^b^c  but defer assignment of y
   z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                    4. Assign x, y, and z with their new values
```

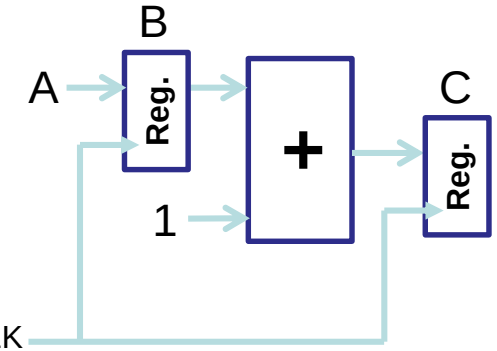
- Sometimes, as above, both produce the same result. Sometimes, not!

# Blocking vs Non-Blocking




Initially assume:

**A = 3;  B = 5;**




```
B = A
C = B + 1
```

```
B <= A
C <= B + 1
```

Finally becomes:

**A = 3;**

**B = 3;**

**C = 4;**

≠

Finally becomes:

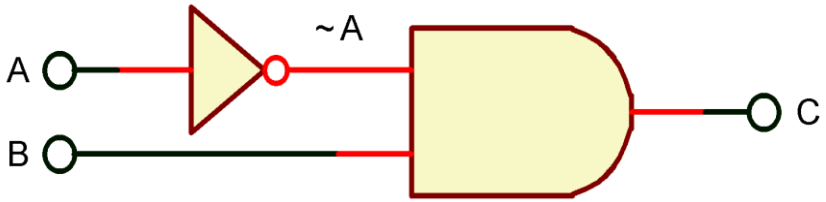
**A = 3;**

**B = 3;**

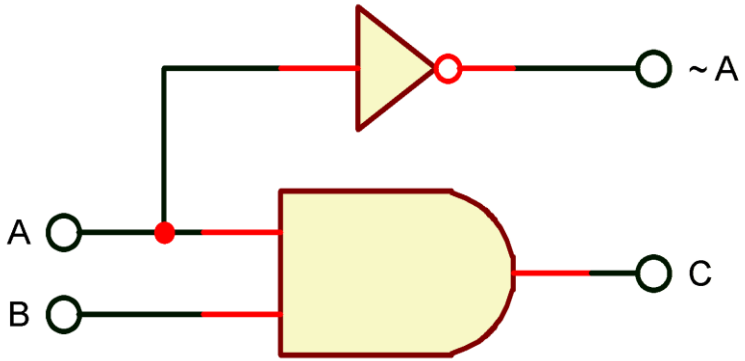
**C = 6;**

# Blocking vs Non-Blocking

```
A = ~A;          // (t=t0)
C = A & B;       // (t=t1)
```

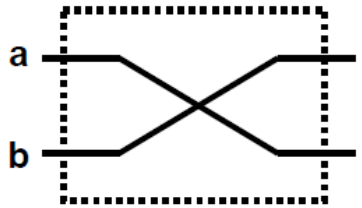
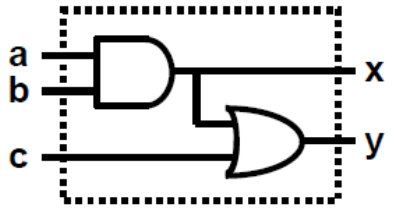



```
A <= ~A;         // (t=t0)
C <= A & B;      // (t=t0)
```



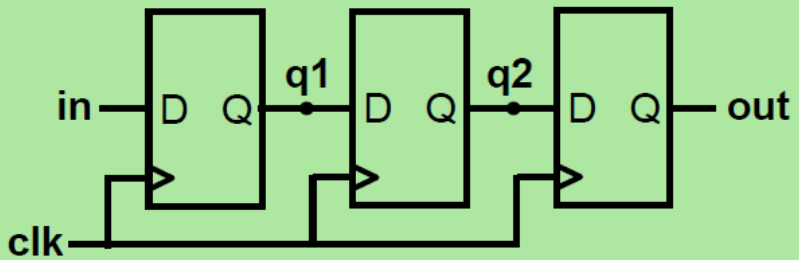
# Why two ways of assigning values?

Conceptual need for two kinds of assignment (in `always` blocks):

| |  |  |
|---|---|---|
| **Blocking**:<br>Evaluation and assignment are immediate | ~~a = b~~<br>~~b = a~~ | x = a & b<br>y = x \| c |
| **Non-Blocking**:<br>Assignment is postponed until all r.h.s. evaluations are done | a <= b<br>b <= a | ~~x <= a & b~~<br>~~y <= x \| c~~ |
| When to use:<br>( only in always blocks! ) | Sequential Circuits | Combinational Circuits |

# Assignment Styles for Sequential Logic

**Flip-Flop Based Digital Delay Line**




- Will nonblocking and blocking assignments both produce the desired result?

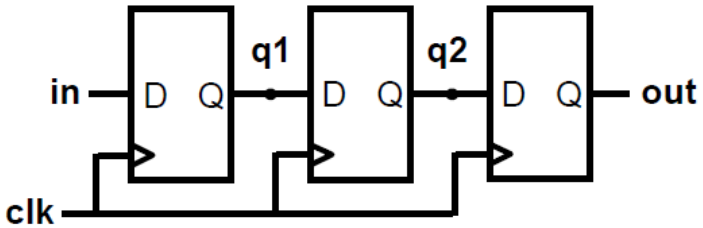
```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

# Use Nonblocking for Sequential Logic

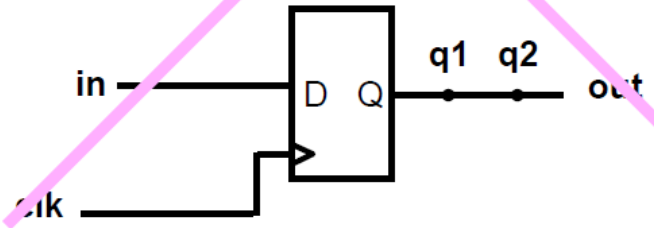
```
always @ (posedge clk)
begin
   q1 <= in;
   q2 <= q1;
   out <= q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."




```
always @ (posedge clk)
begin
   q1 = in;
   q2 = q1;
   out = q2;
end
```

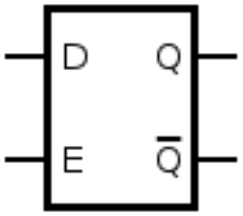
"At each rising clock edge, *q1* = *in*. After that, *q2* = *q1* = *in*; After that, *out* = *q2* = *q1* = *in*; Finally *out* = *in*."




- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential `always` blocks

# Behavioral Description – Example3

## D (Data) Latch:




**Gated D latch truth table**

| E/C | D | Q | $\overline{Q}$ | Comment |
|---|---|---|---|---|
| 0 | X | $Q_{prev}$ | $\overline{Q}_{prev}$ | No change |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 1 | 0 | Set |

**HDL Example**

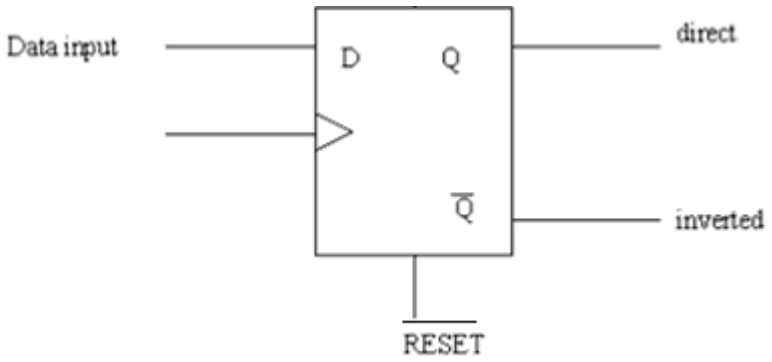
```
// Description of D latch
module D_latch (Q, D, enable);
  output Q;
  input  D, enable;
  reg    Q;
  always @ (enable or D)
    if (enable) Q <= D;                    // Same as: if (enable == 1)
  endmodule
```

# Behavioral Description – Example4

**HDL Example**

```
// D flip-flop without reset
module D_FF (Q, D, Clk);
  output Q;
  input  D, Clk;
  reg    Q;
  always @ (posedge Clk)
    Q <= D;
endmodule

// D flip-flop with asynchronous reset (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
  always @ (posedge Clk, negedge rst)
  if (~rst) Q <= 1'b0;    // Same as: if (rst == 0)
  else Q <= D;
endmodule
```

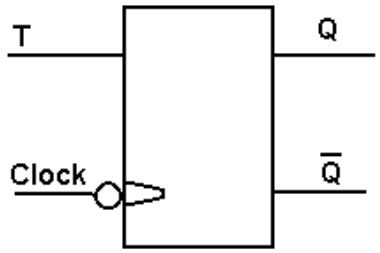
## D (Data) Flip-Flop:




| Clock | D | Q | $Q_{prev}$ |
|---|---|---|---|
| Rising edge | 0 | 0 | X |
| Rising edge | 1 | 1 | X |
| Non-Rising | X | $Q_{prev}$ | |

# Behavioral Description – Example5

## T (Toggle) Flip-Flop:




$$Q_{next} = T \oplus Q = T\overline{Q} + \overline{T}Q$$

| T flip-flop operation[25] | | | | | | | |
|---|---|---|---|---|---|---|---|
| Characteristic table | | | | Excitation table | | | |
| $T$ | $Q$ | $Q_{next}$ | Comment | $Q$ | $Q_{next}$ | $T$ | Comment |
| 0 | 0 | 0 | hold state (no clk) | 0 | 0 | 0 | No change |
| 0 | 1 | 1 | hold state (no clk) | 1 | 1 | 0 | No change |
| 1 | 0 | 1 | toggle | 0 | 1 | 1 | Complement |
| 1 | 1 | 0 | toggle | 1 | 0 | 1 | Complement |

**HDL Example**

```
// T flip-flop from D flip-flop and gates
module TFF (Q, T, Clk, rst);
  output Q;
  input  T, Clk, rst;
  wire   DT;
  assign DT = Q ^ T ;                    // Continuous assignment
// Instantiate the D flip-flop
  DFF TF1 (Q, DT, Clk, rst);
endmodule
```

# Behavioral Description – Example6a

## 4 - Bit Binary Counter:

```
module  Cnt (clk,count);
input clk;
output [3:0] count;
reg [3:0] count;

always @ (posedge clk)
begin
        count <= count + 1;
end
endmodule
```

Count is not initialized
(maybe a problem)

# Behavioral Description – Example6b

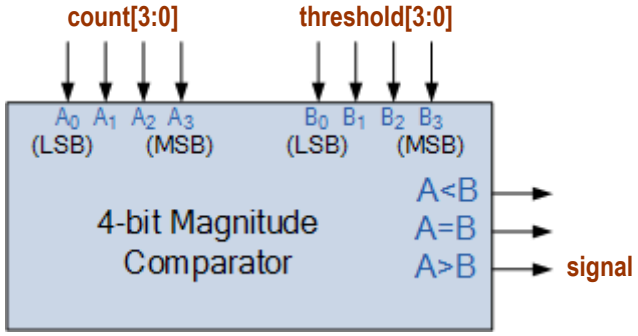
## 4 - Bit Binary Counter with Sync. Reset:

```
module Cnt_r(reset,clk,count);
input reset,clk;
output [3:0] count;
reg [3:0] count;

always @ (posedge clk)
begin
    if(reset)
        count <= 0;
    else
        count <= count + 1;
end
endmodule
```

# Behavioral Description – Example6c

## 4 - Bit Binary Counter with a threshold signal:

```
module Counter(reset,clk,count,threshold,signal);
input reset,clk;
input [3:0] threshold;
output signal;
output [3:0] count;
reg [3:0] count;

assign signal = count > threshold;

always @ (posedge clk)
begin
    if(reset)
        count <= 0;
    else
        count <= count + 1;
end
endmodule
```




4-bit binary counter with synch. reset

# *Testbenches*

- HDL that tests another module: *device under test* (DUT)
- Not synthesizable
- Uses different features of Verilog
- Types:
  - Simple
  - Self-checking
  - Self-checking with testvectors

# *Testbench Example*

- Write Verilog code to implement the following function in hardware:

```
y = bc + ab
```

- Name the module `t_function`

# Testbench Example

- Write Verilog code to implement the following function in hardware:

$$y = bc + ab$$

```
module t_function(input a, b, c,
                  output y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

# Simple Testbench

```
module testbench1();
  reg a, b, c;
  wire y;

  // instantiate device under test
  t_function dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

# Self-checking Testbench

```
module testbench2();
  reg a, b, c;
  wire y;
  t_function dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

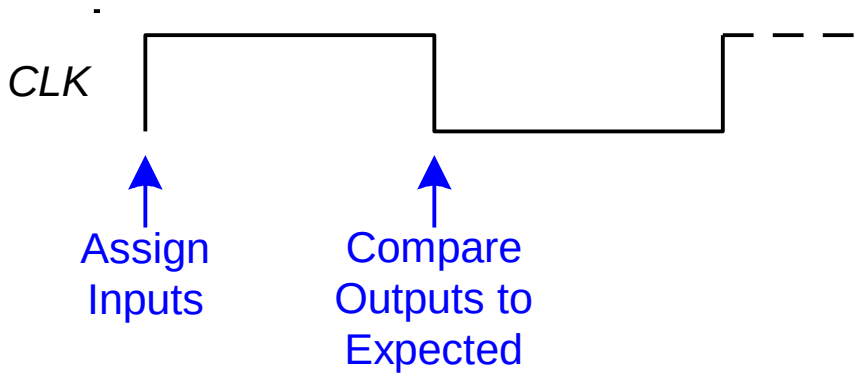
# *Testbench with Testvectors*

- Testvector file: inputs and expected outputs

- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read test vectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs with expected outputs and report errors

# Testbench with Testvectors

- Testbench clock:
  - assign inputs (on rising edge)
  - compare outputs with expected outputs (on falling edge).




- Testbench clock also used as clock for synchronous sequential circuits

# *Testvectors File*

- File: `example.tv`
- contains vectors of abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
Xxx_x
```

# 1. Generate Clock

```
module testbench3();
  reg clk, reset;
  reg a, b, c, yexpected;
  wire y;
 reg [3:0] vectornum, errors;    // bookkeeping variables
 reg [3:0] testvectors[8:0]; // array of testvectors

  // instantiate device under test
  t_function dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

# 2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1; #30; reset = 0;
   end


// Note: $readmemh reads testvector files written in
// hexadecimal
```

# 3. Assign Inputs & Expected Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

# 4. Compare with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```

# 4. Compare with Expected Outputs

```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $stop;
      end
    end
endmodule


// === and !== can compare values that are 1, 0, x, or z.
```

# Testbench3 (complete)

```
module testbench3();
  wire clk, reset;
  wire a, b, c, yexpected;
  wire y;
  wire [31:0] vectornum, errors;    // bookkeeping variables
  wire [3:0]  testvectors[9999:0]; // array of testvectors
  // instantiate device under test
  t_function dut(a, b, c, y);
  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
  // at start of test, load vectors and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #30; reset = 0;
    end
  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end
   // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end
      // increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $stop;
      end
    end
endmodule
```

# HOW TO USE THE QUARTUS TOOL

**Intel Quartus**

Quartus Prime Lite Edition is a free version of the professional-strength Intel® Quartus® Prime Design Software tools.

It allows us to define our digital designs in schematic or Verilog or VHDL hardware description language (HDL).

After modeling the design, one can simulate the corresponding circuit using ModelSim, which is also available with Quartus.

Please see the provided tutorials
- Quartus Prime Introduction Using Verilog Designs.
- Using ModelSim to Simulate Logic Circuits in Verilog Designs

# Exercise

1. **A majority circuit is a combinational circuit whose output is equal to one if the input variables have more 1's than 0's. Design a three input majority circuit.**

a. Design the circuit and draw its schematic diagram. Write a module using the gate level description of the circuit in Verilog.(Define 3 bits using vectors and label your wire names clearly on schematic)

b. Write a module using the data flow description of the circuit.

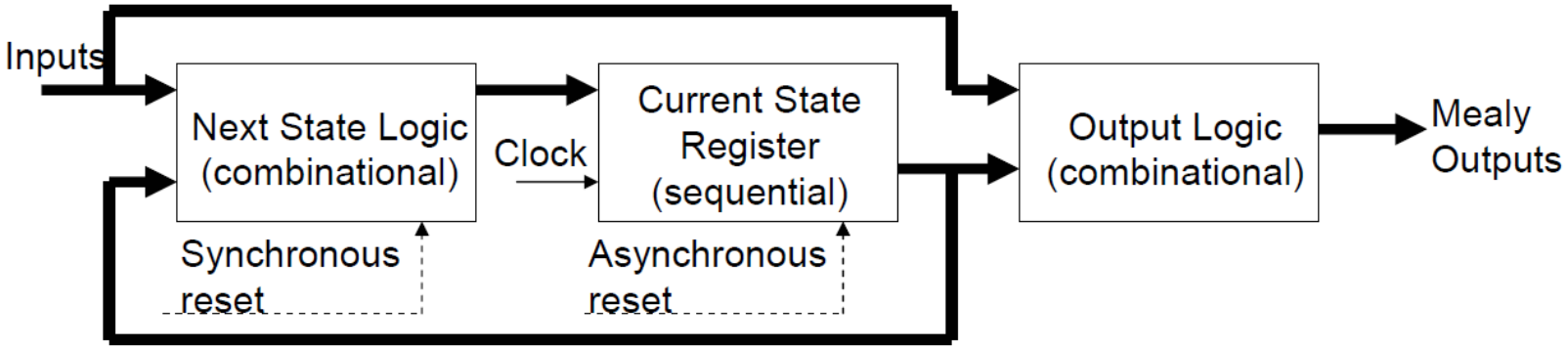
c. Write a module using the behavioral description of the circuit.

2. **Using Quartus and ModelSim** (see the provided tutorials)**.**

a. Generate a test waveform that applies 3 bits input values. The input numbers should be the last four digits(take modulo 8) of your student number. For example, if your student number is 1626218 then apply 6, 2, 1, 0(8≡0 @mod8)to the input of the circuit.

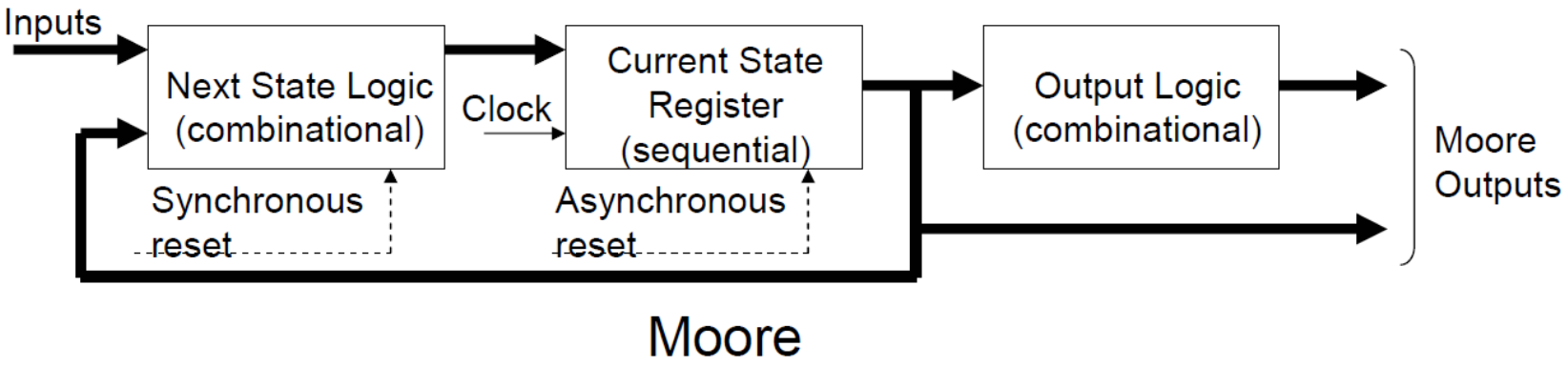
b. Simulate the verilog codes you have written in part1 (each of the gate level, data flow and behavioral descriptions) using the test waveform. Plot the input/output waveforms you obtained.

# FSM Structures (1/3)



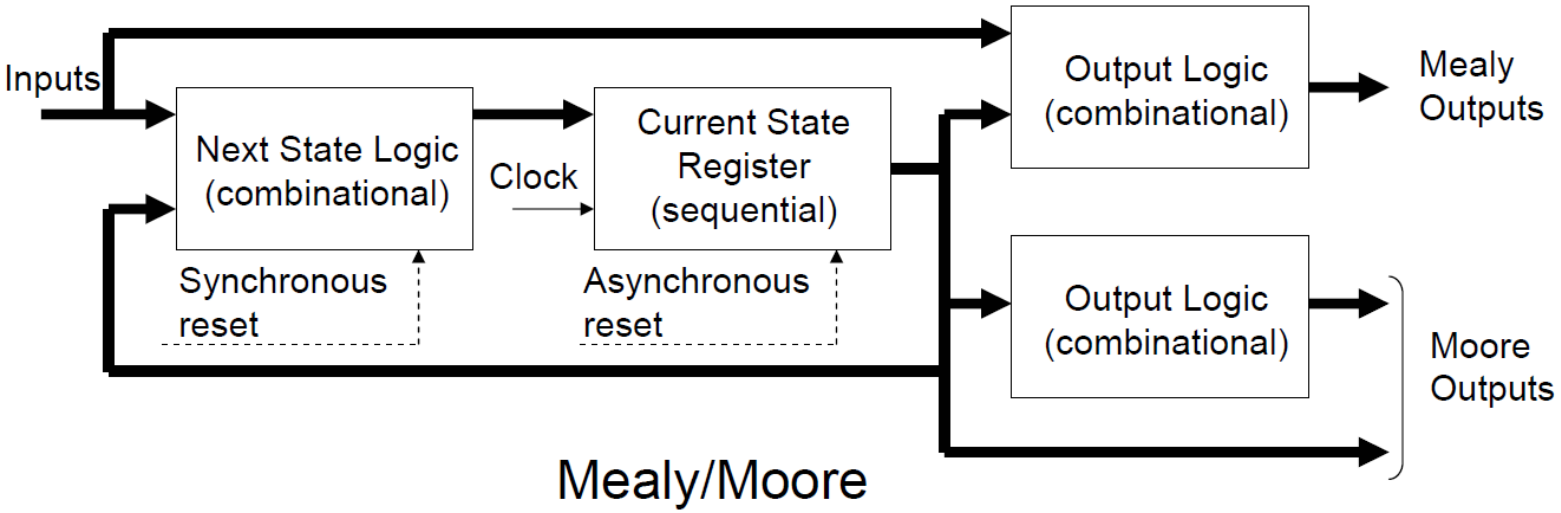

Mealy

# FSM Structures (2/3)



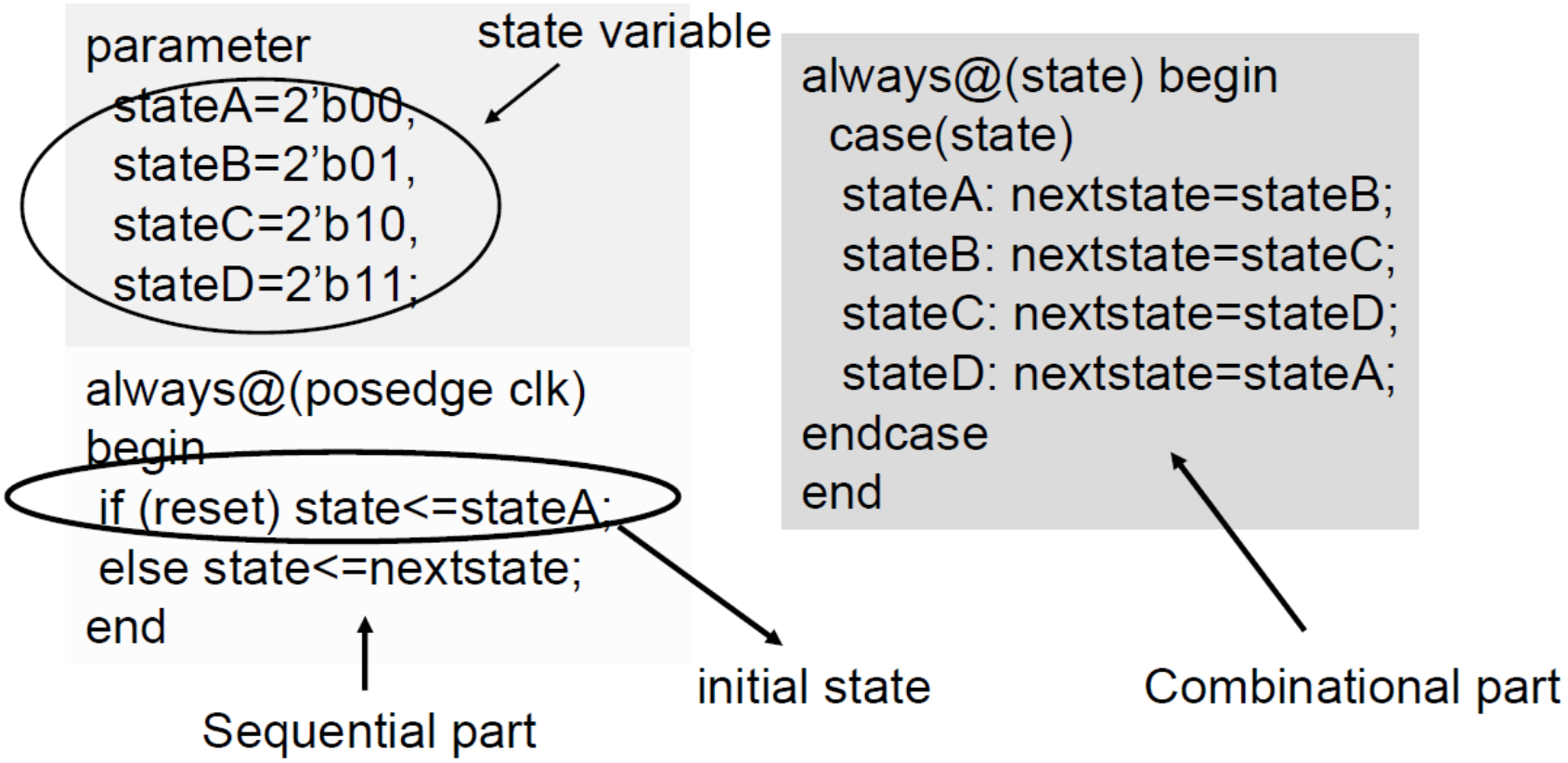

Moore

# FSM Structures (3/3)



# Coding for FSM

- Separate the combination part and the sequential part
- Use parameters to define the state variables



# Blocking v.s Non-Blocking

- Use non-blocking assignments for sequential block
  - Store values until the end of the time slice
  - Avoid simulation race conditions or ambiguity of results
- Use blocking assignments for combinational block
  - Blocking assignments occur immediate in nature

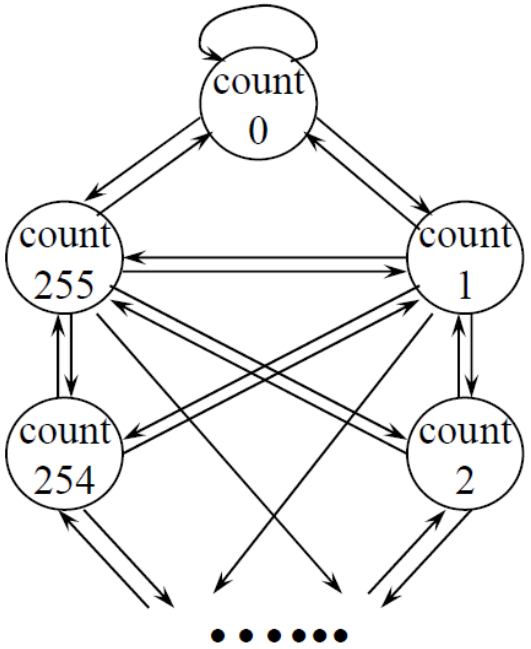
# *Various Coding Styles for FSM*

- There are many different coding styles can be used to describe a FSM
  - HDL is a very flexible language
- Typical coding styles:
  - 1-process FSM
  - 2-process FSM
  - 3-process (or more) FSM

# 1-Process FSM

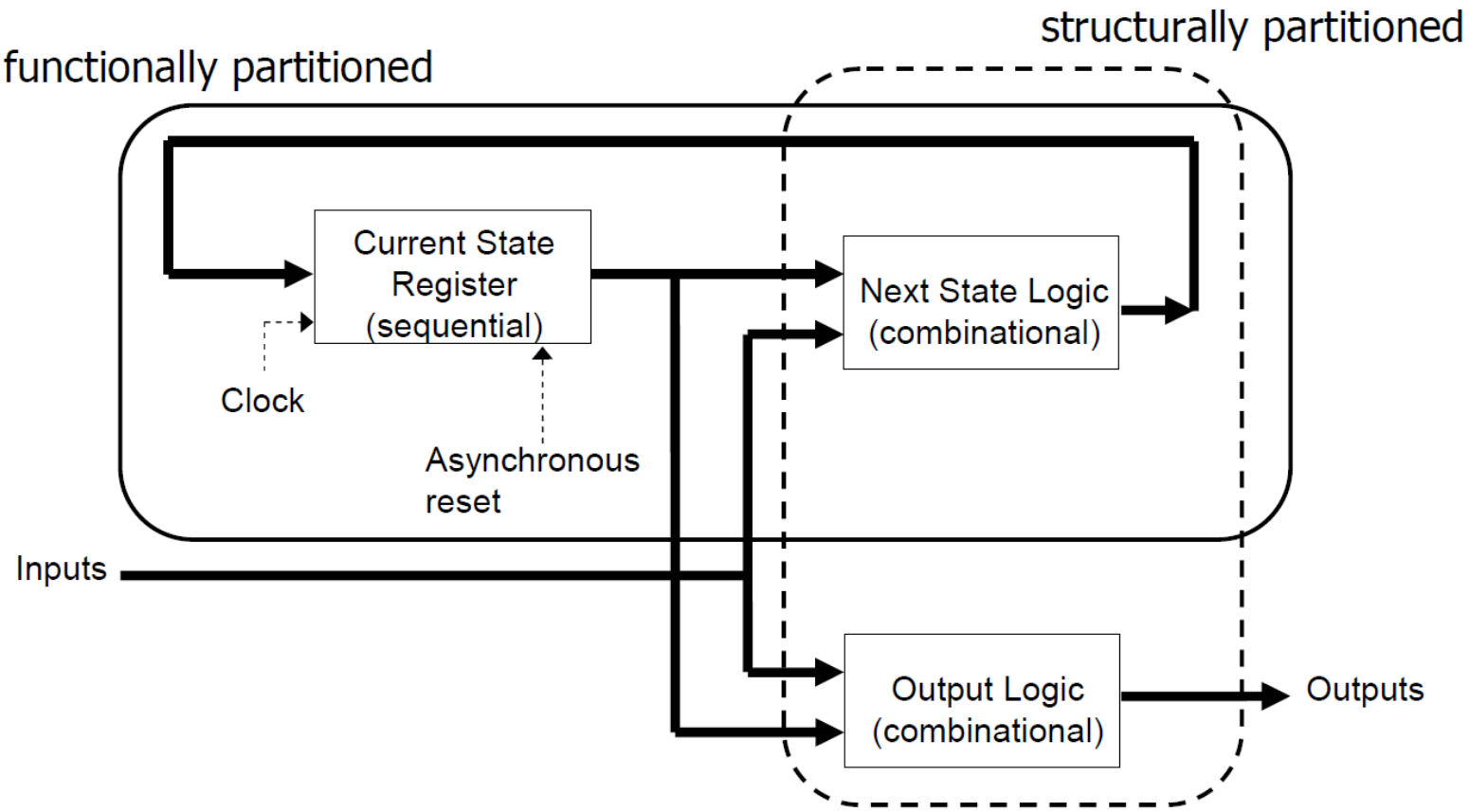
- Lump all descriptions into a single process

```
module counter (clk, rst, load, in, count) ;
input          clk, rst, load ;
input   [7:0] in ;
output [7:0] count ;
reg      [7:0] count ;

always @(posedge clk) begin
    if (rst) count = 0 ;
    else if (load) count = in ;
    else if (count == 255) count = 0 ;
    else count = count + 1 ;
end
endmodule
```




256 states 66047 transitions

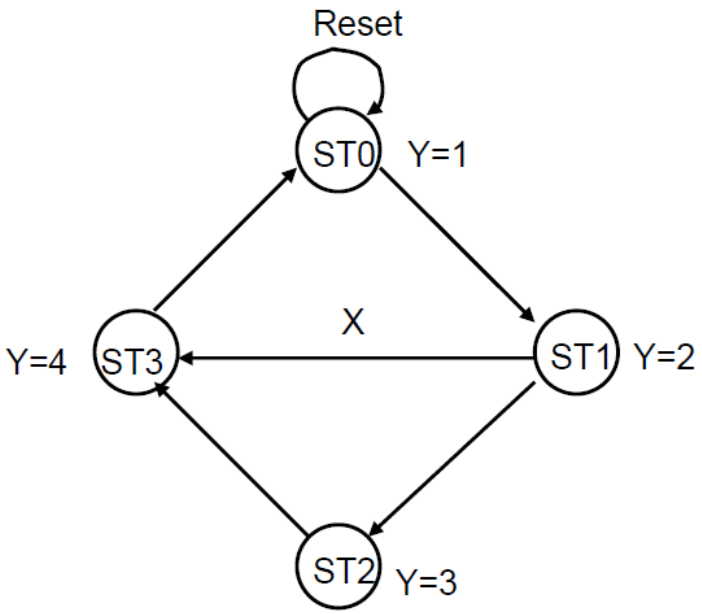
# 2-Process FSM (1/5)



# 2-Process FSM (2/5)

```
module FSM_S2 (Clock, Reset, X, Y);
      input  Clock, Reset, X;
      output  [2:0] Y;
      reg  [2:0] Y;
      reg  [1:0] CS, NS;
      parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;

      always @(X or CS) begin : COMB
            case (CS)
            ST0 : begin
                  Y = 1;
                  NS = ST1;
            end
            ST1 : begin
                  Y = 2;
                  if (X)  NS = ST3;
                  else  NS = ST2;
            end
```

2-process, structurally partitioned



# 2-Process FSM (3/5)

```
    ST2 : begin
            Y = 3;
            NS = ST3;
          end
    ST3 : begin
            Y = 4;
            NS = ST0;
          end
    default : begin
            Y = 1;
            NS = ST0;
          end
  endcase
end
```

```
always @(posedge Clock or posedge Reset)
  begin : SEQ
    if (Reset)
      CS <= ST0;
    else
      CS <= NS;
  end

endmodule
```

# 2-Process FSM (4/5)

- 2-process, functionally partitioning

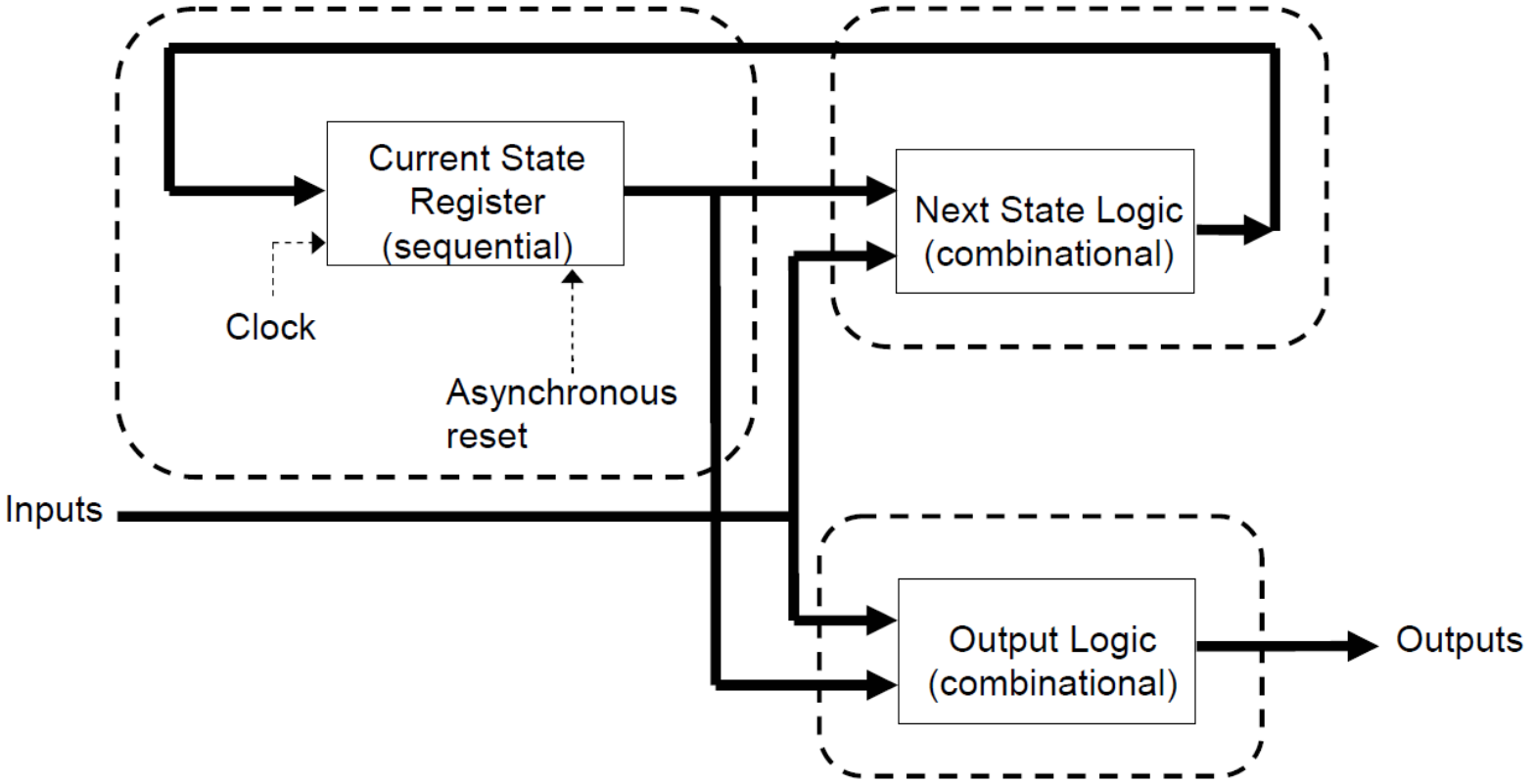
```
module FSM_F2 (Clock, Reset, X, Y);
    input  Clock, Reset, X;
    output  [2:0] Y;
    reg  [2:0] Y;
    reg [1:0] STATE;
    parameter  [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;

  always @(posedge Clock or posedge Reset)
   begin : NEXT_STATE
    if (Reset)
      STATE <= ST0;
    else
```

# 2-Process FSM (5/5)

```
    case (STATE)
      ST0 : STATE <= ST1;
      ST1 : begin
          if (X)
              STATE <= ST3;
          else
              STATE <= ST2;
      end
      ST2 : STATE <= ST3;
      ST3 : STATE <= ST0;
    endcase
end
```

```
always @(STATE)
        begin : OUT
            case (STATE)
              ST0 : Y = 1;
              ST1 : Y = 2;
              ST2 : Y = 3;
              ST3 : Y = 4;
              default : Y = 1;
            endcase
        end

endmodule
```

# 3-Process FSM (1/4)



# 3-Process FSM (2/4)

- 3-process, structurally partitioning

```
module FSM_S3(Clock, Reset, X, Y);
    input  Clock, Reset, X;
    output  [2:0] Y;
     reg  [2:0] Y;
     reg  [1:0] CS, NS;
     parameter  [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;

  always @(X or CS)
    begin : COMB
      NS = ST0;
      case (CS)
```

# 3-Process FSM (3/4)

```
ST0 : begin
        NS = ST1;
      end
ST1 : begin
      if (X)
          NS = ST3;
      else
          NS = ST2;
      end
```

```
      ST2 : begin
              NS = ST3;
            end
      ST3 : begin
              NS = ST0;
            end
    endcase
end
// end process COMB
```

# 3-Process FSM (4/4)

```
always @(posedge Clock
            or posedge Reset)
  begin : SEQ
    if (Reset)
      CS <= ST0;
    else
      CS <= NS;
  end
```

```
always @(CS)
    begin : OUT
      case (CS)
        ST0 : Y = 1;
        ST1 : Y = 2;
        ST2 : Y = 3;
        ST3 : Y = 4;
        default : Y = 1;
      endcase
    end

endmodule
```

# Ex: Binary 4-bit multiplier (1/4)

```
module binary_mult (clk, Reset_b, start, in1, in2, out);
   input clk, Reset_b, start;
   input [3:0] in1, in2;
   output [7:0] out;
   reg [1:0] state, next_state;

   parameter S_idle=2'b00, S_add=2'b01, S_shift=2'b10;

   reg [3:0] A, B, Q;
   reg C;
   wire P_zero;
   reg [2:0] P;
```

# *Ex: Binary 4-bit multiplier (2/4)*

```
assign P_zero=~|P;
assign out={A,Q};

//state reg
always @(posedge clk or posedge Reset_b)
   begin
        if (Reset_b==1)
           state <= S_idle;
        else
           state <= next_state;
   end
```

# Ex: Binary 4-bit multiplier (3/4)

```
//next_state function
always @(start or P_zero or state)
begin
    case (state)
        S_idle:
            if (start==1)
                next_state = S_add;
            else
                next_state = S_idle;
        S_add:
            next_state = S_shift;
        S_shift:
            if (P_zero==1)
                next_state = S_idle;
            else
                next_state = S_add;
    endcase
end
```

# Ex: Binary 4-bit multiplier (4/4)

```
//data path
always @(posedge clk)
      begin
          case (state)
              S_idle:
                   if (start==1)
                      begin
                         A <= 4'b0000;
                         B <= in1;
                         Q <= in2;
                         C <= 1'b0;
                         P <= 2'b00;
                      end
               S_add:
                   begin
                         P <= 2'b01;
                         if (Q[0]==1)
                                {C,A} = A + B;
                   end
               S_shift:
                   begin
                         C <= 1'b0;
                         A <= {C, A[3:1]};
                         Q <= {A[0],Q[3:1]};
                   end
          endcase
      end
endmodule
```

# *Delays in Verilog*

- Represent different physical concepts
- Two types
  - Inertial delay
  - Transport delay
- ***Syntax: #delay***

  It delays execution for a specific amount of time, ***'delay'.***
- #(delay) can not be synthesized.
- #(delay) can be used in testbench files to create delays.

# *Delay Models*

**Inertial Delay**

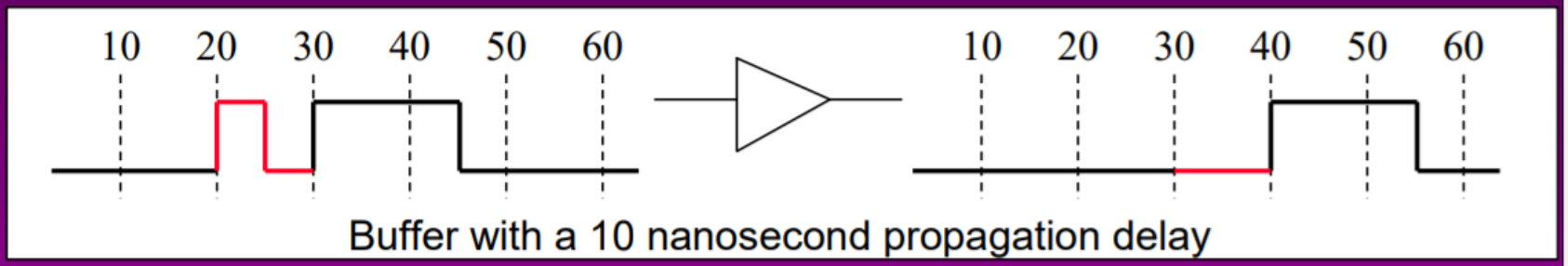
- The inertia of a circuit node to change value
- Abstractly models the RC circuit seen at the node
- Two different types exist
  - Input inertial delay
  - Output inertial delay

**Transport Delay**

- Represents the propagation time of signals from module inputs to its outputs
- Models the internal propagation delays of electrical elements

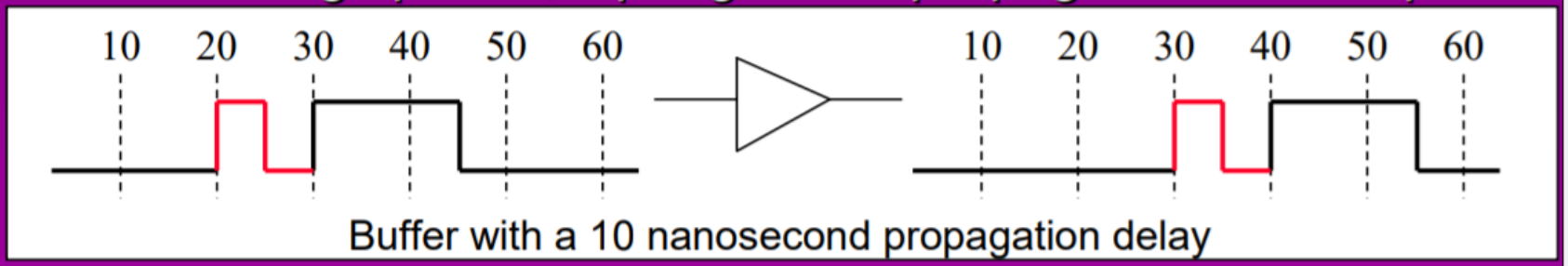
# Delay Models

- Hardware has two primary propagation delay methods:
  - **Inertial delay** models devices with finite switching speeds; input glitches do not propagate to the output




Buffer with a 10 nanosecond propagation delay

  - **Transport delay** models devices with near infinite switching speeds; input glitches propagate to the output




Buffer with a 10 nanosecond propagation delay

# *Delays in Verilog*

**Delayed assignment:**

***#Δt variable = expression;***

*//*“expression” *gets evaluated after the time delay Δt and assigned to the “variable” immediately*

**Intra-assignment delay:**

***variable = #Δt expression;***

// “expression” gets evaluated at time 0 but gets assigned to the “variable” after the time delay Δt

# *Intra-assignment delay*

- The right-hand side is evaluated before the delay
- The left-hand side is assigned after the delay

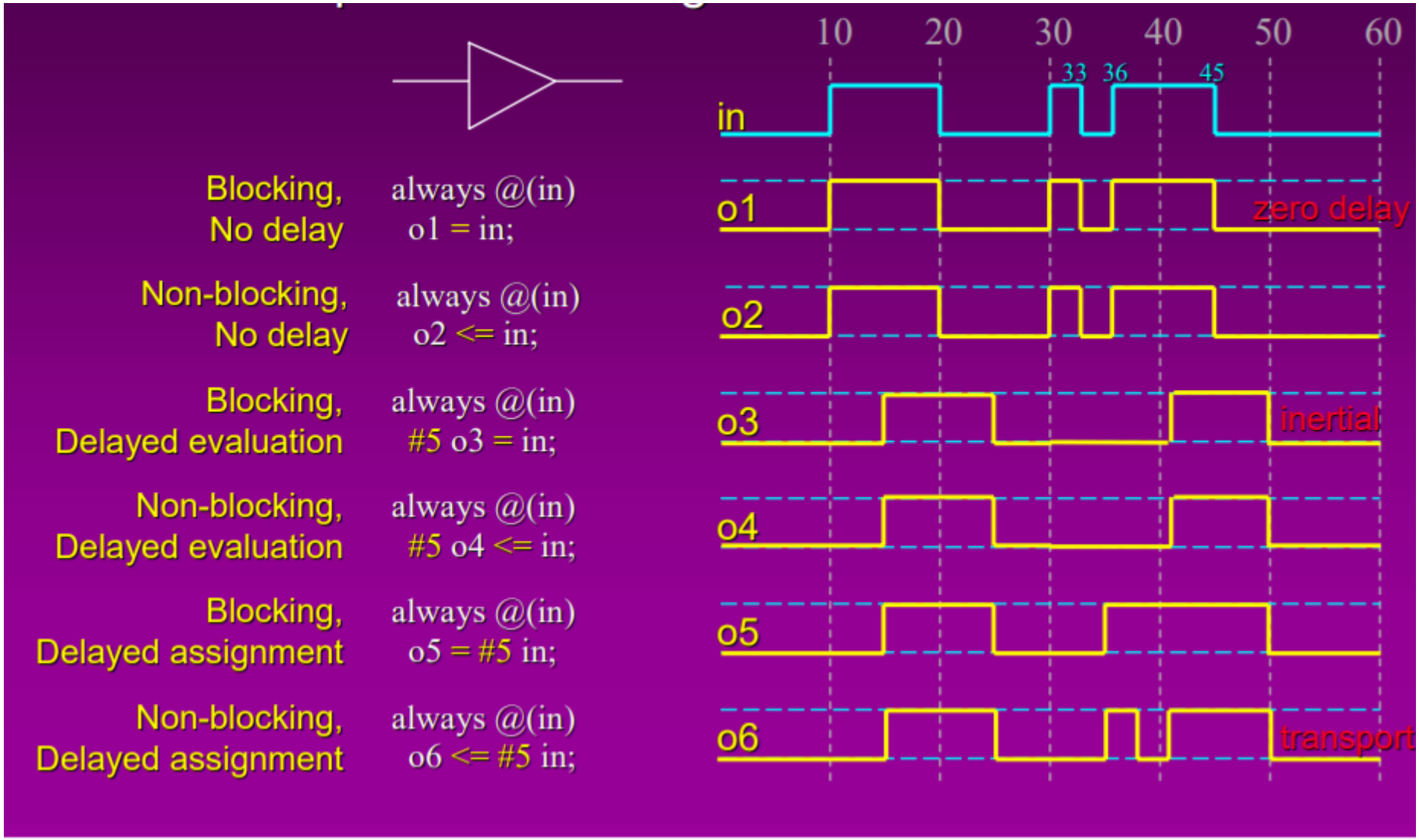
Examples:

```
always @(A)
    B = #5 A;     // A is evaluated at the time it changes but
                  // is not assigned to B until after 5 time units

always @(negedge clk )
    Q <= @(posedge clk) D;    // D is evaluated at the negative edge of CLK,
                              // Q is changed on the positive edge of CLK
```

# Delay example

```
module delay_test(
   input a,  // Assume a=0 initialized at time '0'
   input b,  // Assume b=1 initialized at time '0'
   output reg c,
   output reg d
);

initial
begin
   #20 c = (a | b);    // a | b gets evaluated after 20ns and gets assigned to ‘c’
                       // immediately
   d = #50 (a & b);  // a & b gets evaluated at time 20ns but gets assigned to
                       // ‘c’ at time 70ns (20ns+50ns)
end
endmodule
```

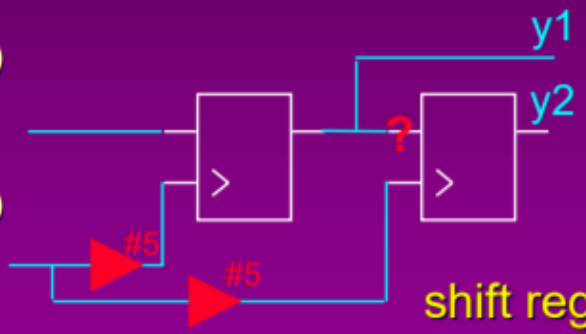
# Buffer with Procedural Assignmens

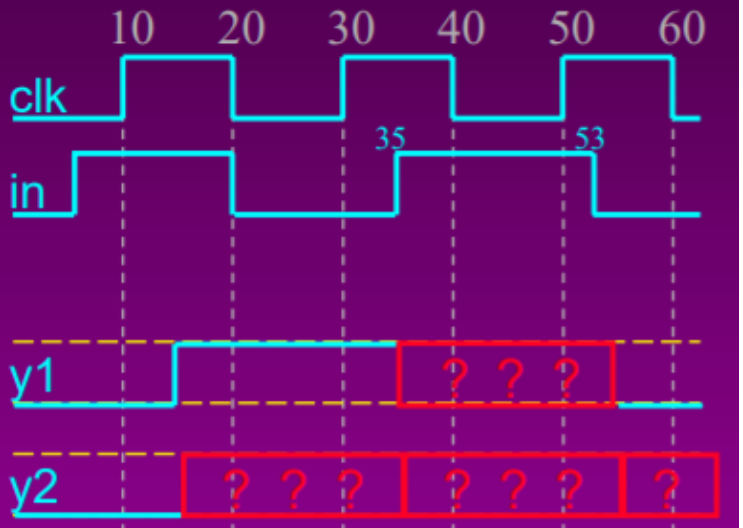


# Shift Register with Race Conditions

- How will these procedural assignments behave?
  - Concurrent assignments
  - Delayed evaluation

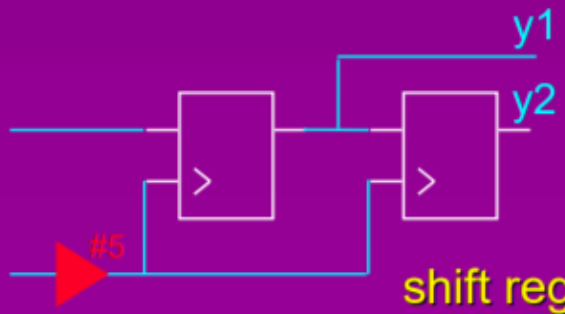
```
always @(posedge clk)
  #5 y1 = in;

always @(posedge clk)
  #5 y2 = y1;
```

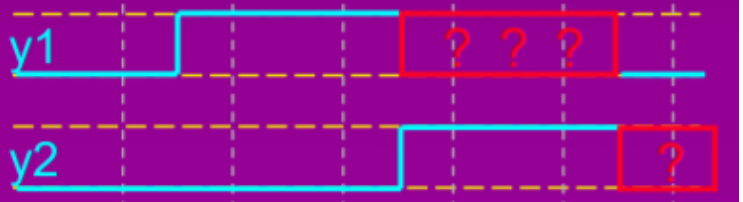



shift register with race condition

```
always @(posedge clk)
  #5 y1 <= in;

always @(posedge clk)
  #5 y2 <= y1;
```




shift register with race condition

# Recommendations

- Combinational Logic:
  - No delays: Use blocking assignments ( a = b; )
  - Inertial delays: Use delayed evaluation blocking assignments ( #5 a = b; )
  - Transport delays: Use delayed assignment non-blocking assignments ( a <= #5 b; )
- Sequential Logic:
  - No delays: Use non-blocking assignments ( q <= d; )
  - With delays: Use delayed assignment non-blocking assignments ( q <= #5 d; )

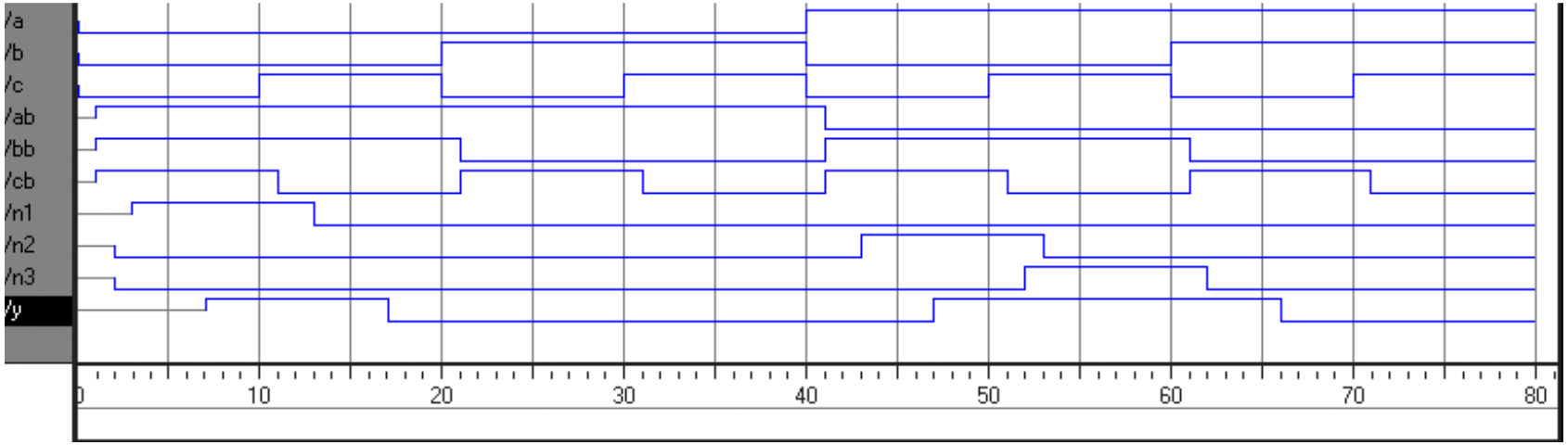
# Combinational Cct. Delay

a compiler directive that sets the simulation's time unit size and precision

```
`timescale 1ns/1ps
```

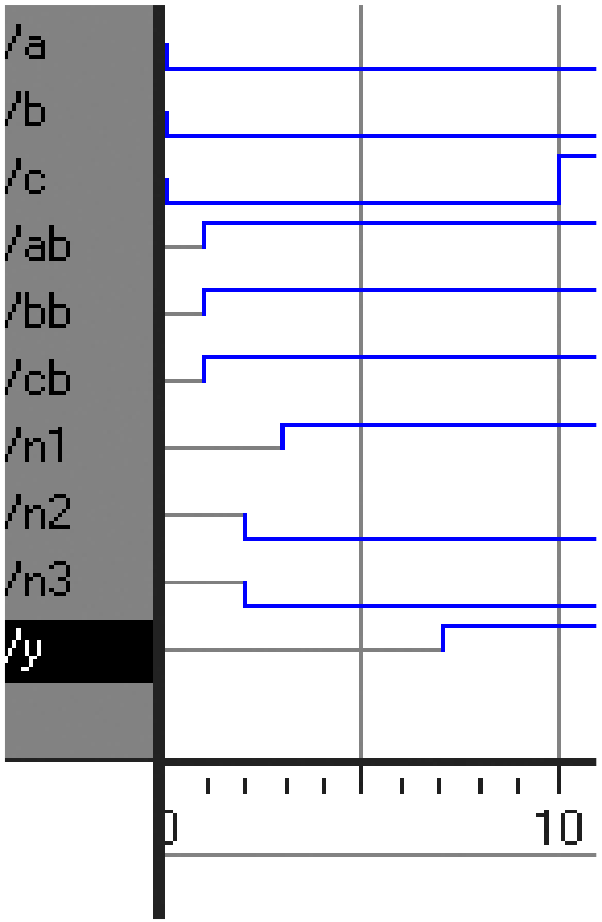
<reference_time_unit>/<time_resolution>

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



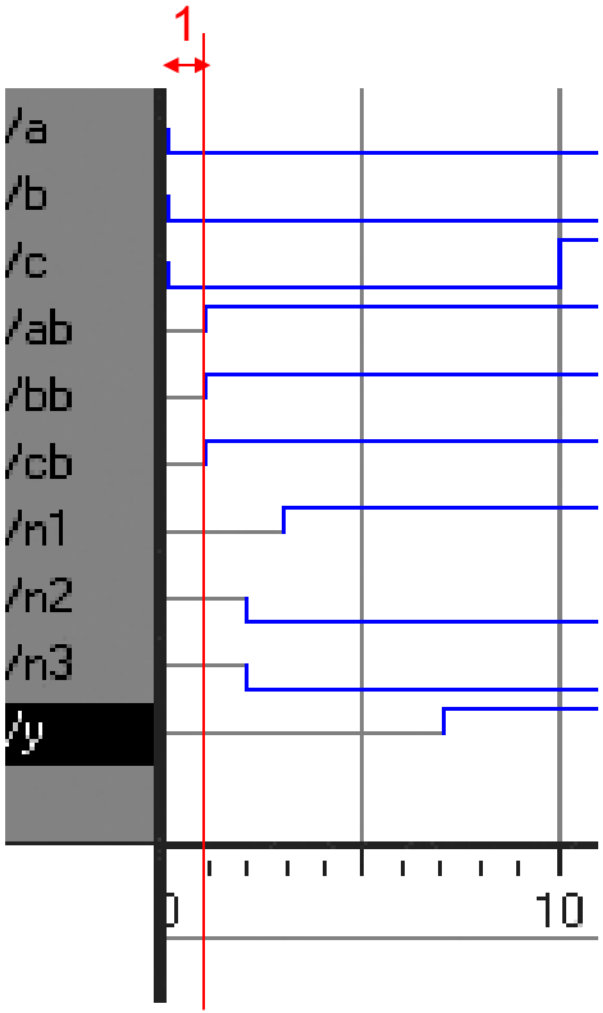
# Combinational Cct. Delay

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



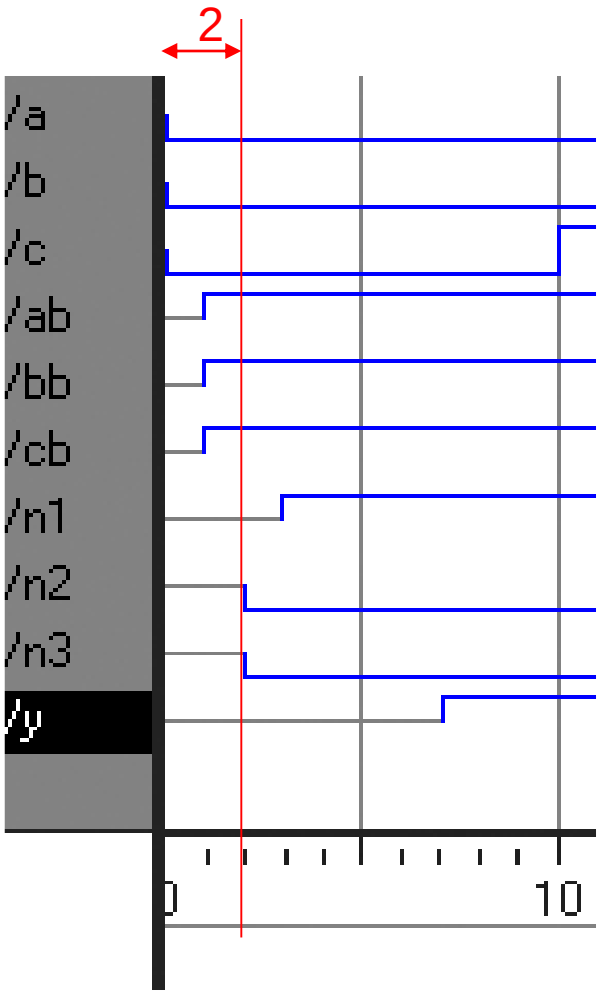
# Combinational Cct. Delay

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



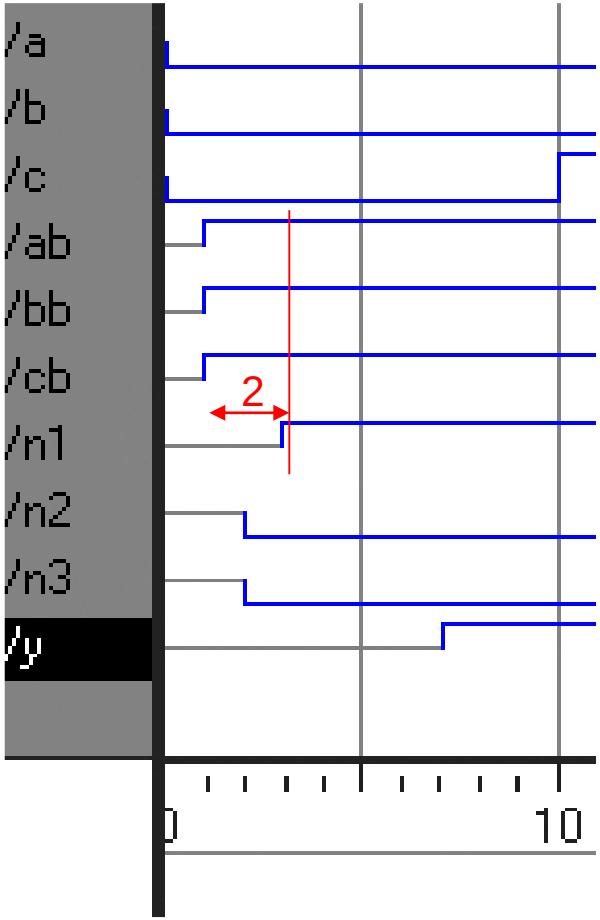
# Combinational Cct. Delay

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



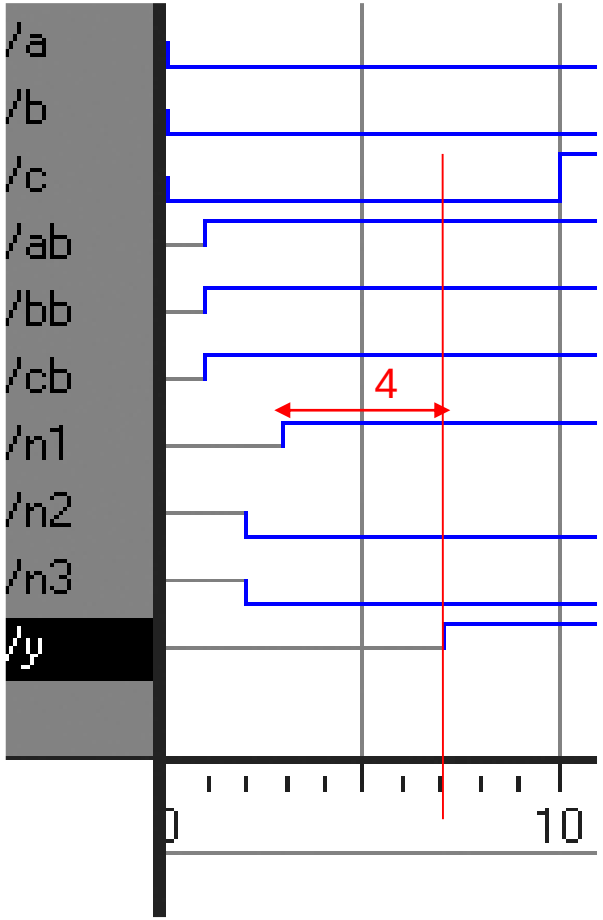
# Combinational Cct. Delay

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



# Combinational Cct. Delay

```
module example(input a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



# Concurrent Procedural Assignments

The order of concurrent evaluation is **indeterminate**

➤ Concurrent blocking assignments have **unpredictable results**

```
always @(posedge clk)
  #5 A = A + 1;

always @(posedge clk)
  #5 B = A + 1;
```

**Unpredictable Result:**
(new value of B could be evaluated before or after A changes)

➤ Concurrent non-blocking assignments have **predictable results**

```
always @(posedge clk)
  #5 A <= A + 1;

always @(posedge clk)
  #5 B <= A + 1;
```

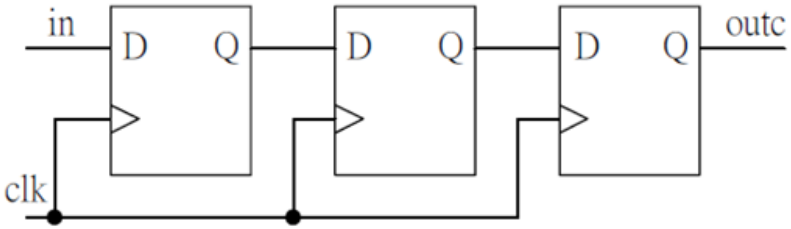
**Predictable Result:**
(new value of B will always be evaluated before A changes)

# Non-Blocking Assignment

```
always @(posedge clk)
begin
  outa<=in;
  outb<=outa;
  outc<=outb;
end
```

≠

```
always @(posedge clk)
  outa=in;
always @(posedge clk)
  outb=outa;
always @(posedge clk)
  outc=outb;
```



# Representing Simulation Time as Queues

➤ Each Verilog simulation time step is divided into 4 queues

**Time 0:**

- Q1 — *(in any order)*
  - Evaluate RHS of all non-blocking assignments
  - Evaluate RHS and change LHS of all blocking assignments
  - Evaluate RHS and change LHS of all continuous assignments
  - Evaluate inputs and change outputs of all primitives
  - Evaluate and print output from $display and $write
- Q2 — (in any order)
  - Change LHS of all non-blocking assignments
- Q3 — (in any order)
  - Evaluate and print output from $monitor and $strobe
  - Call PLI with reason_synchronize
- Q4 :
  - Call PLI with reason_rosynchronize

**Time 1:**

**...**

*Note: this is an abstract view, not how simulation algorithms are implemented.*

# One Possible Execution Order

```
always@(posedge clk)
begin
   out1 = 3;   ❸
end

always@(posedge clk)
begin
   out1  = 0;  ❶
   out1<= 1;   ❺
   out1  = 2;  ❷
end

always@(out1)
begin
   out2 = ~out2;  ❹ ❻
end
```

```
0       clk
bxx     out1
0       out2
$end

#5
1       clk
b00     out1   ❶
b10     out1   ❷
b11     out1   ❸
1       out2   ❹
b01     out1   ❺
0       out2   ❻
```