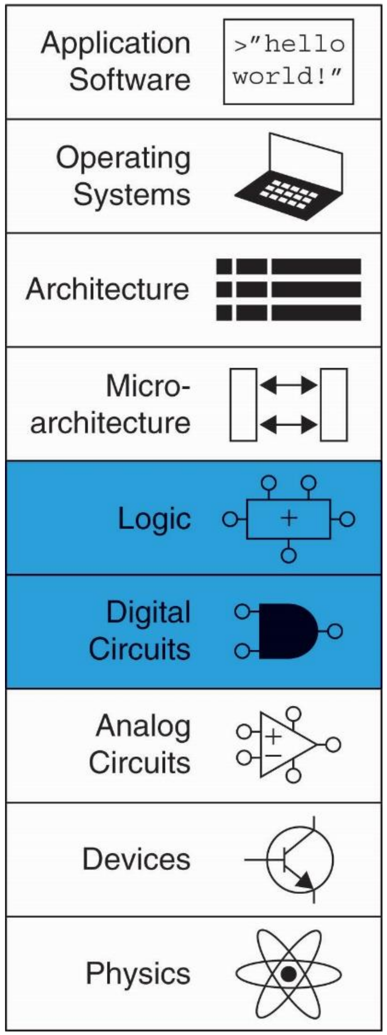
数字逻辑设计与微处理器系统
第五讲
Hardware Description Languages

马 上
通信抗干扰技术国家级重点实验室
mashang@uestc.edu.cn
Office：B3-510, Main Building
Tel（O）：61830321

2021年4月

Contents


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
Digital Circuits
Analog Circuits
Devices
Physics

1.1 Verilog Introduction

- **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:**
  - **SystemVerilog/Verilog**
    - **developed in 1984 by Gateway Design Automation**
    - **IEEE standard (1364) in 1995**
    - **Extended in 2005 (IEEE STD 1800-2009)**
  - **VHDL 2008**
    - **Developed in 1981 by the Department of Defense**
    - **IEEE standard (1076) in 1987**
    - **Updated in 2008 (IEEE STD 1076-2008)**

1.1 Verilog Itroduction

- What & Why HDL?
- Verilog HDL
  - Modual Concept and structure
  - Identifier, Data Representation,Data Type
  - Mathmatical symbol
- Abstraction Levels
  - Gates
  - Dataflow
  - Procedural

1.1 What & Why HDL?

- Hardware Description Language(HDL)
  - A software programming language used to model the intended operation of a piece of hardware
- Why HDL?
  - Text-based design rather than Schematic design
    - ASIC complexity increase
    - faster time-to-market
  - Simulation
  - Logic Synthesis
  - Words are better than pictures
  - PLD, CPLD and FPGA became inexpensive and commonplace

1.1 What is need for HDL ?

- Model, Represent, And Simulate Digital Hardware
  - Hardware Concurrency
  - Parallel Activity Flow
  - Semantics for Signal Value And Time
- Special Constructs And Semantics
  - Edge Transitions
  - Propagation Delays
  - Timing Checks

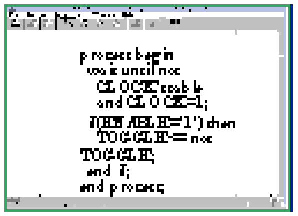
1.1 History of HDL

- First HDL
  - PALASM, 1980s--Logic equation
- Then more complex (minimization, if-then-else statement)
  - CUPL (Complier Universal for Programmable Logic)
  - ABEL (Advanced Boolean Equation Language)
- Verilog HDL
- VHDL represents another high level language for digital system design.
  - VHDL = VHSIC HDL
  - VHSIC = Very High Speed Integrated Circuit

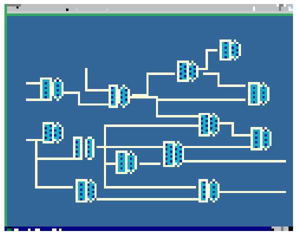
1.1 HDLs

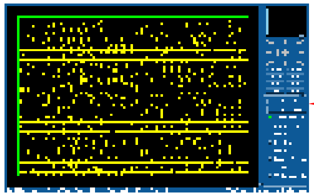
- Software: Assembler →high-level language, like C, C++, Java
- Hardware: block diagram & schematic →HDL
- Efficient →Large, complex, abstraction
- HDL Tool Suits
  - Text editor
  - Complier
  - Synthesizer
  - Simulator
  - Template generator
  - Schematic viewer
  - Translator
  - Timing analyzer
  - Back annotator

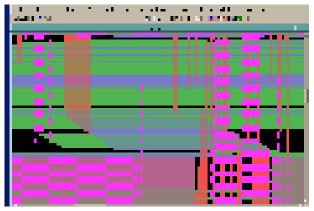
1.1 Design flow

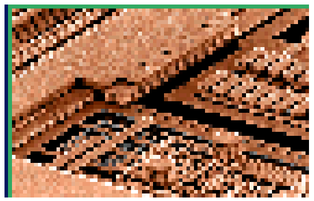


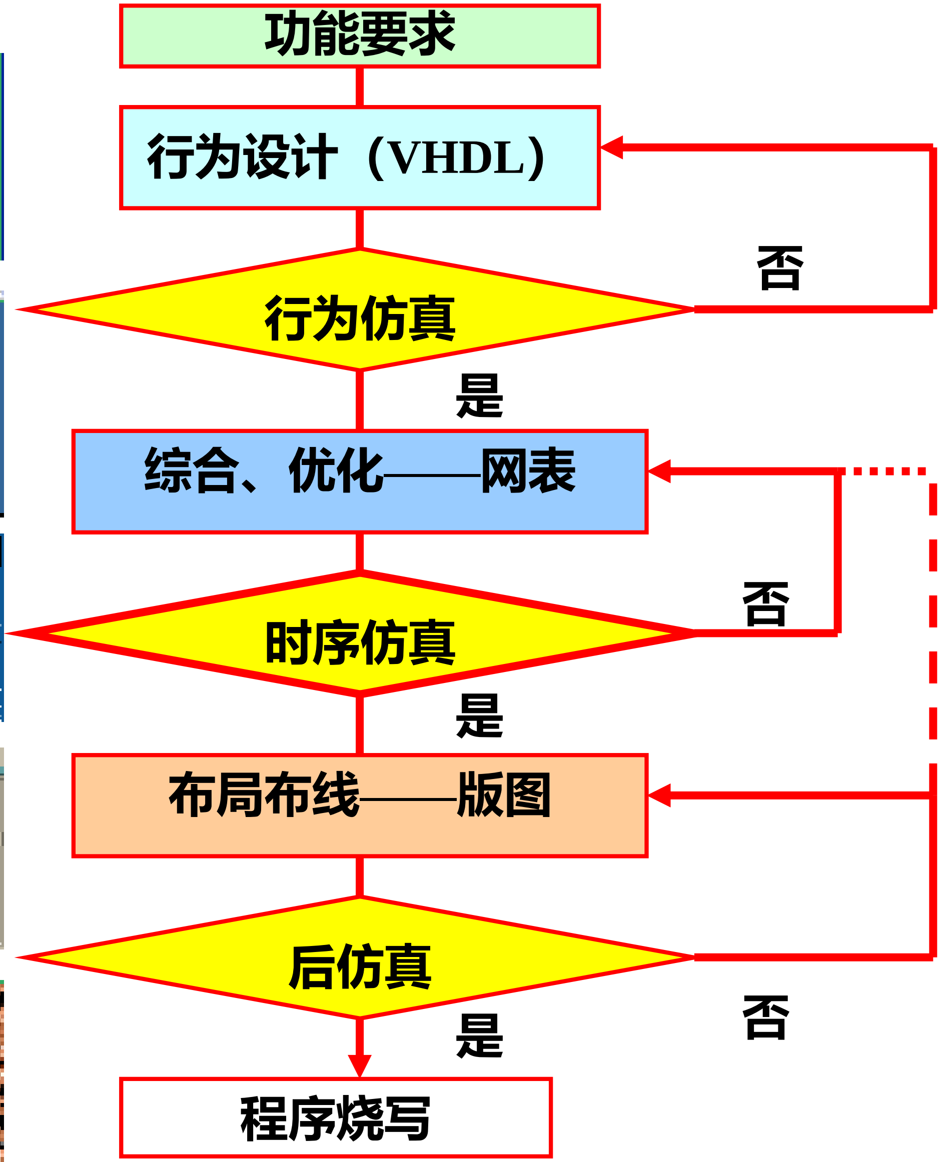
```
process begin
  wait until not
    CLOCK'stable
    and CLOCK=1;
   if(ENABLE='1') then
    TOGGLE<= not
TOGGLE;
  end if;
end process
```











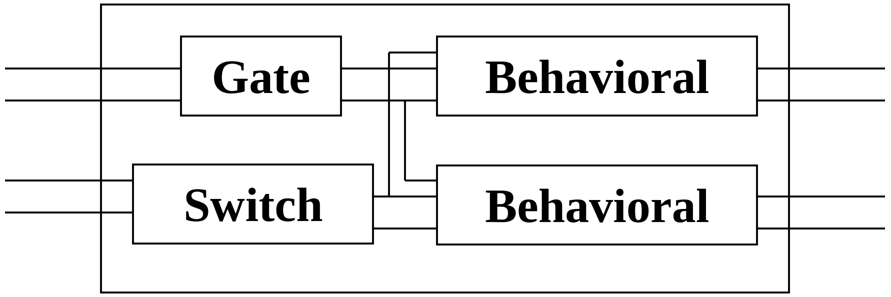
设计创意

+

仿真验证

1.1 Introduction to Verilog HDL

- Originally designed in 1983/1984 as a proprietary verification /simulation
- IEEE standard 1364 in 1995
- Similar to C language
- 4 value logic (0, 1, x, z)
- HDL which provides a wide range of levels of abstraction
  - Architectural, Algorithmic, RTL, Gate, Switch
- Mixed level modeling and simulation
- Description of digital systems only
- Interactive usage
- Hierarchical specification



1.1 History

- 1981
  - Gateway Design Automation was founded
  - Phil Mooby make GHDL(GenRad's HDL) and HILO simulator
- 1983
  - Gateway released the Verilog HDL and simulator
- 1985
  - The language and simulator were enhanced and renamed to "Verilog-XL"
- 1988 synopsys make Verilog synthesis tool
- 1989
  - Cadence bought Gateway
- 1995
  - The Verilog HDL was adopted as IEEE standard 1364
  - i.e. Verilog-1995
- 1997, Verilog-2001

1.1 Verilog Spans a Wide Range of Modeling Levels

Increasing level of abstraction

- Architectural

```
always #($dis_poisson(seed,32))
begin
    if $q_full(qid)
            $q_remove (qid, job, job_id, status);
    else
            ➔ fill_queue;
end
```

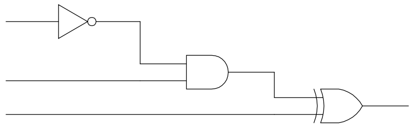
- Algorithmic

```
always @(fetch_done)
begin
        casez (IR[7:6])
                2'b00 : LDA (acc, IR[5:0]);
                2'b01 : STR (acc,IR[5:0]);
                2'b10 : JMP (IR[5:0]);
                2'b11 : ;    // NOP
        endcase
end
```

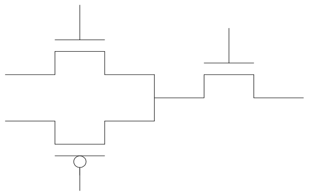
- RTL

```
assign rt1 = (11&buserr) | zero;
assign sub = rt1 ^| op;
assign out1 = i1 & i2 | op;
```

- Gate



- Switch



behavioral (non-structural)

gate/switch (structural)

1.1 HDL to Gates

- **Simulation**
  - **Inputs applied to circuit**
  - **Outputs checked for correctness**
  - **Millions of dollars 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)**

IMPORTANT:

When using an HDL, think of the hardware the HDL should produce

1.2 Verilog Syntax

- **Case sensitive**
  - **Example: reset and Reset are not the same signal.**
- **Formed from {[A-Z], [a-z], [0-9], _, $}, but can't begin with $ or [0-9]**
  - **myidentifier** ☺
  - **m_y_identifier** ☺
  - **3myidentifier** ☻ ✗
  - **$my_identifier** ☻ ✗
  - **_myidentifier$** ☺
- **Whitespace ignored**

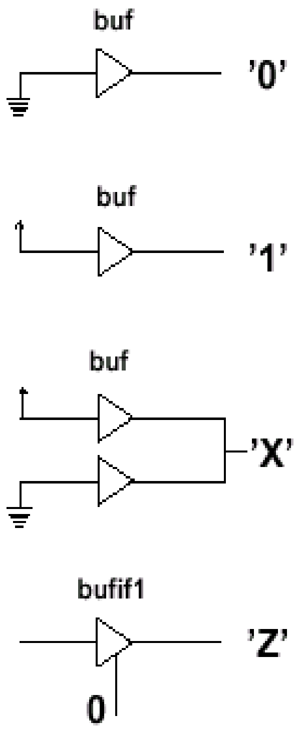
1.2 Verilog Syntax

- **Comments:**

```
// single line comment
/* multiline
        comment */
```

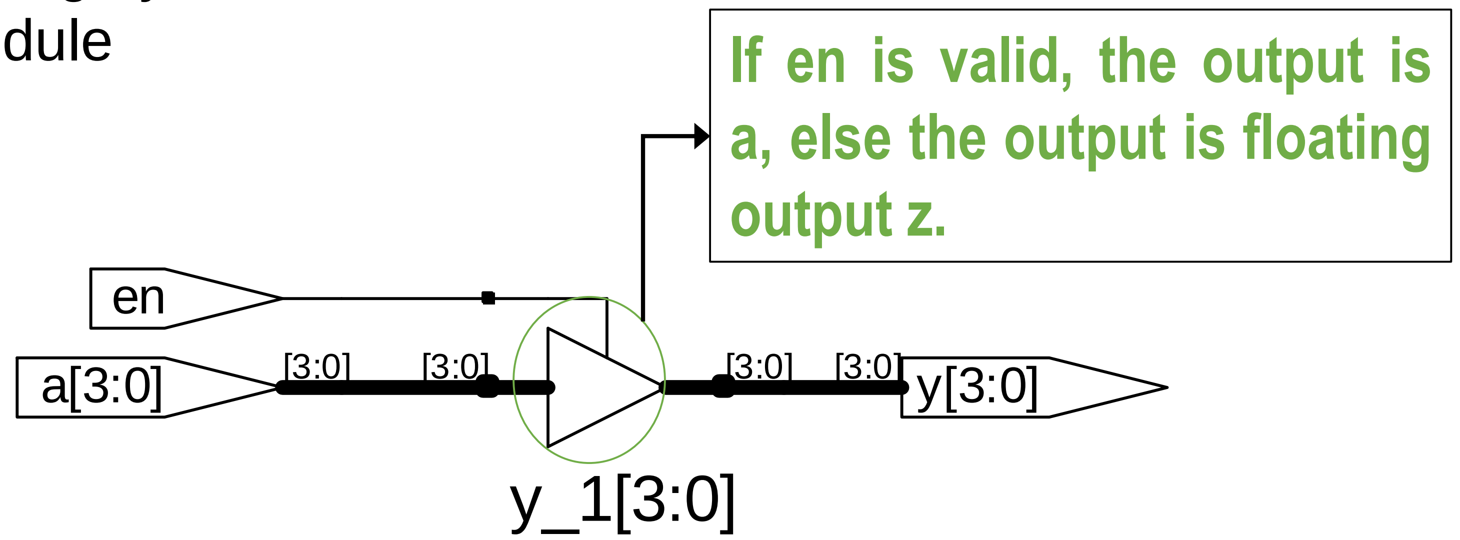
- **Verilog Value Set**

  - **0 represents low logic level or false condition**
  - **1 represents high logic level or true condition**
  - **x represents unknown logic level**
  - **z represents high impedance logic level**



1.2 Z: Floating Output

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


If en is valid, the output is a, else the output is floating output z.
en
a[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
y[3:0]
y_1[3:0]

1.2 Numbers in Verilog

<size>’<radix> <value>

No of bits

Binary	$\rightarrow$ b or B
Octal	$\rightarrow$ o or O
Decimal	$\rightarrow$ d or D
Hexadecimal	$\rightarrow$ h or H

Consecutive chars
0-f, x, z

- **8’h ax = 1010xxxx**
- **12’o 3zx7 = 011zzzxxx111**

- You can insert “_” for readability
  - 12’b 000_111_010_100
  - 12’b 000111010100
  - 12’o 07_24

1.2 Numbers in Verilog

- Bit extension
  - MS bit = 0, x or z ⇒ extend this
    - 4’b x1 = 4’b xx_x1
  - MS bit = 1 ⇒ zero extension
    - 4’b 1x = 4’b 00_1x
- If *size* is ommitted it
  - is inferred from the *value* or
  - takes the simulation specific number of bits or
  - takes the machine specific number of bits
- If *radix* is ommitted too .. decimal is assumed
  - 15 = <size>’d 15

1.2 Data Types

- **Nets**
  - **Connects between hardware elements**
  - **Must be continuously driven by**
    - **Continuous assignment (assign)**
    - **Module or gate instantiation (output ports)**
  - **Default initial value for a *wire* is “Z” (and for a *trireg* is “x”)**
- **Registers**
  - **Represent data storage elements**
  - **Retain value until another value is placed on to them**
  - **Similar to “variables” in other high level language**
  - **Different to edge-triggered flip-flop in real circuits**
  - **Do not need clock**
  - **Default initial value for a *reg* is “X”**

1.2 Nets

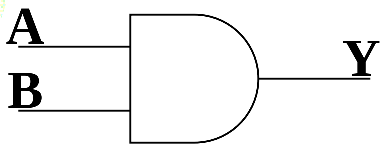
- Can be thought as hardware wires driven by logic
- Equal *z* when unconnected
- Various types of nets

wire, tri	: standard
wor, trior	: wired OR
wand, triand	: wired AND
trireg	: capacitive
tri1	: pull up
tri0	: pull down
supply1	: power
supply0	: ground

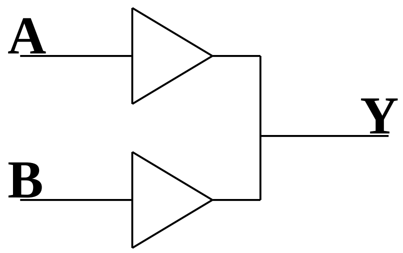
综合编译器不支持的net类型

1.2 Nets

In following examples: Y is evaluated, *automatically*, every time A or B changes



```
wire Y;  // declaration
assign Y = A & B;
```

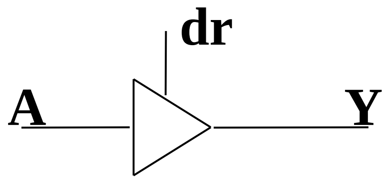


```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

Y		A	
		0	1
B	0	0	0
	1	0	1

```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

Y		A	
		0	1
B	0	0	1
	1	1	1



```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

1.2 Registers

- **Variables that store values**
- **Do not represent real hardware but ..**
- **.. real hardware can be implemented with registers**
- **Only one type: `reg`**

```
reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0
```

- **Register values are updated explicitly!!**

1.2 Vectors

- **Represent buses**

```
wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;
```

- **Left number is MS bit**
- **Slice management**

```
busC[1] = busA[2];
busC[0] = busA[1];
```

- **Vector assignment (*by position!!*)**

```
busB[1] = busA[3];
busB[2] = busA[2];
busB[3] = busA[1];
busB[4] = busA[0];
```

1.2 Register Types

- Integer
  - integer a,b; // declaration
  - 32-bit signed (2's complement)
- real
  - real c, d; //declaration
  - 64-bit real number
  - Defaults to an initial value of 0.0
- Time
  - 64-bit unsigned, behaves like a 64-bit reg
  - $display("At %t, value=%d", $time, val_now)

1.2 Integer & Real Data Types

- **Declaration**

```
integer i, k;
real r;
```

- **Use as registers (inside procedures)**

```
i = 1; //assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
```

- **Integers are not initialized!!**
- **Reals are initialized to *0.0***

1.2 Time Data Type

- **Special data type for simulation time measuring**
- **Declaration**

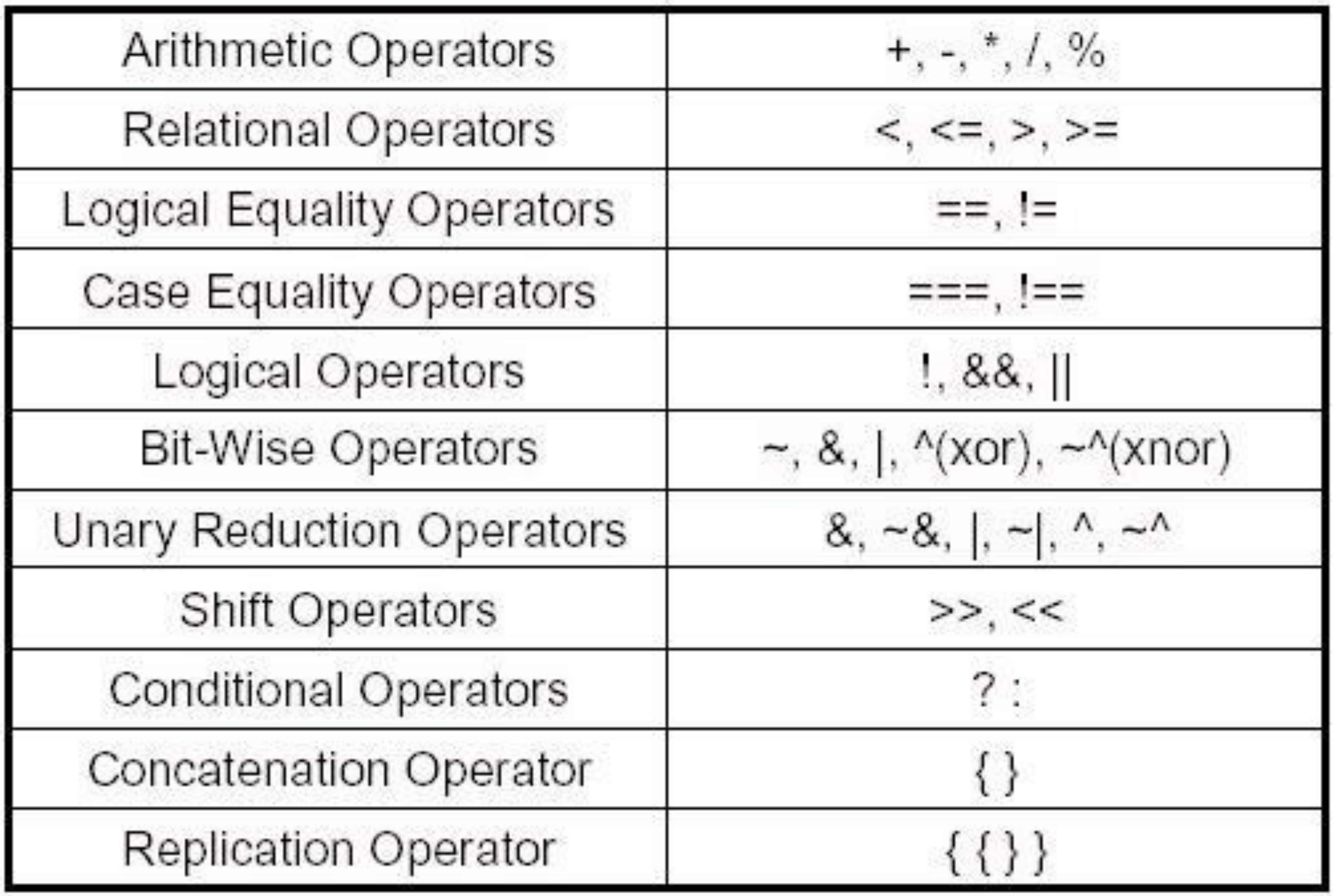
```
time my_time;
```

- **Use inside procedure**

```
my_time = $time; // get current sim time
```

- **Simulation runs at simulation time, not real time**

1.2 Operators



Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

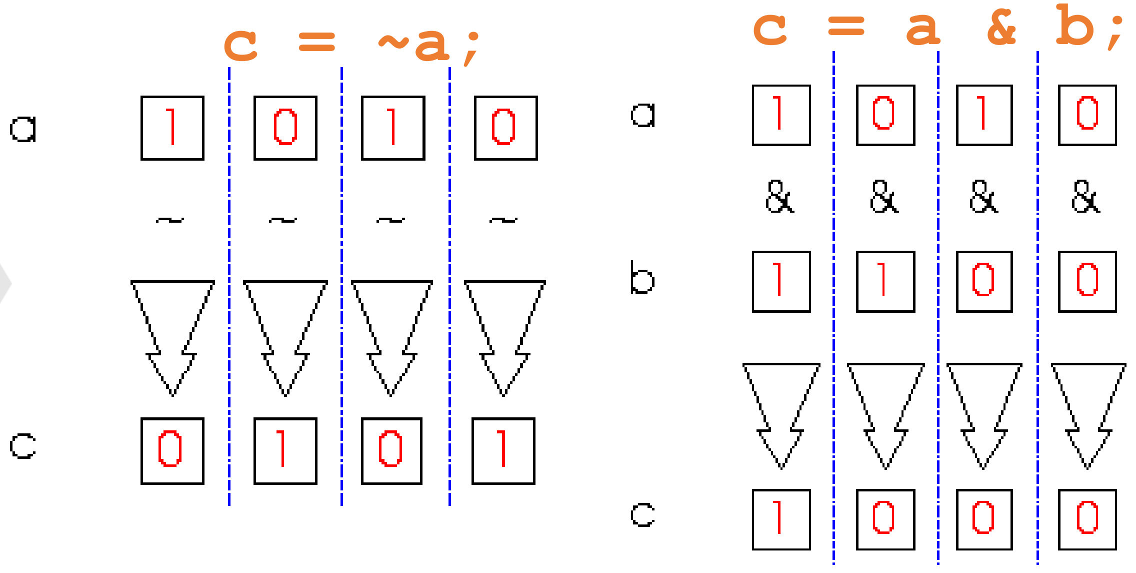
1.2 Bitwise Operators

- & → **bitwise AND**
- | → **bitwise OR**
- ~ → **bitwise NOT**
- ^ → **bitwise XOR**
- ~^ ^~ → **bitwise XNOR**

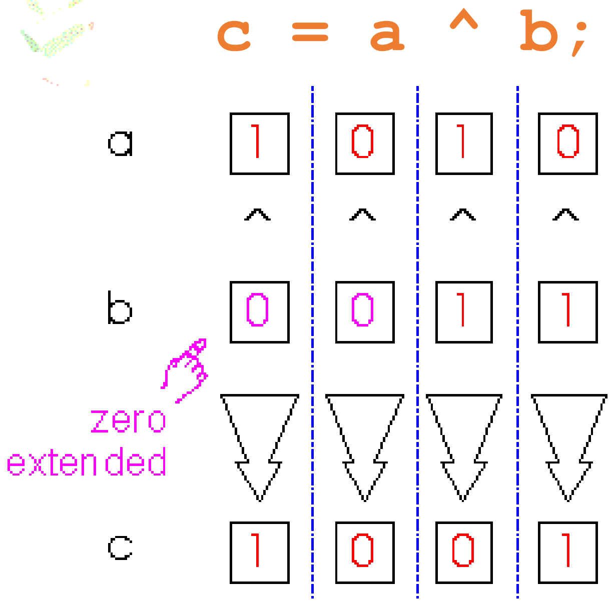
- **Operation on bit by bit basis**

1.2 Bitwise Operators

- **a = 4'b1010;**
  b = 4'b1100;

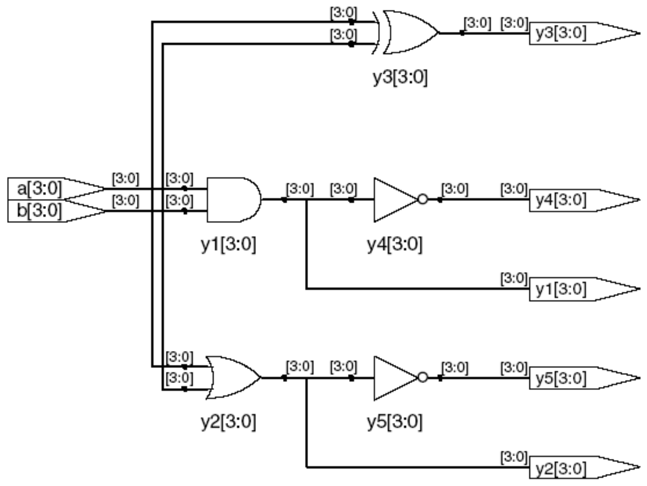


- **a = 4'b1010;**
  b = 2'b11;



1.2 Bitwise Operators

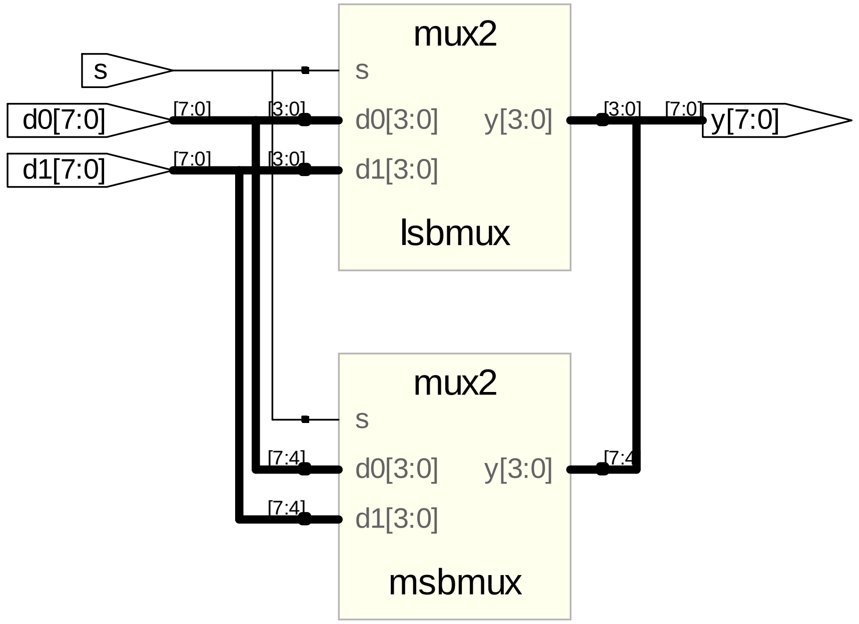
```
module gates(input  [3:0]  a, b,
                    output [3:0] y1, y2, y3, y4, y5);
      /* Five different two-input logic
      gates acting on 4 bit busses */
      assign y1 = a & b;    // AND
      assign y2 = a | b;    // OR
      assign y3 = a ^ b;    // XOR
      assign y4 = ~(a & b); // NAND
      assign y5 = ~(a | b); // NOR
endmodule
```



1.2 Bit Manipulations

```
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
```



1.2 Reduction Operators

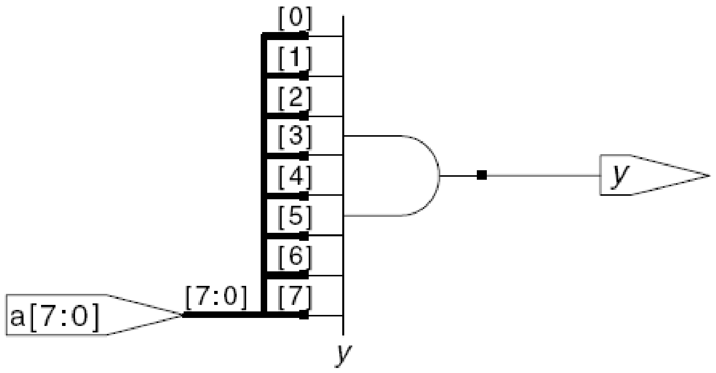
&	→ **AND**
|	→ **OR**
^	→ **XOR**
~&	→ **NAND**
~|	→ **NOR**
~^ ^~	→ **XNOR**

- **One multi-bit operand → One single-bit result**

```
a = 4'b1001;
..
c = |a; // c = 1|0|0|1 = 1
```

1.2 Reduction Operators

```
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
```



1.2 Logical Operators

- && → **logical AND**
- || → **logical OR**
- ! → **logical NOT**
- **Operands evaluated to ONE bit value: *0*, *1* or *x***
- **Result is ONE bit value: *0*, *1* or *x***
  - **All value should be changed to true or false before operate**
  - **4'b0100 && 4'b0101 ⇒** true && true = true(1)
  - **4'b0100 & 4'b0101 = 4'b0100**

```
A = 6;        A && B → 1 && 0 → 0
B = 0;        A || !B → 1 || 1 → 1
C = x;        C || B → x || 0 → x
```

careful !!

but C&&B=0

1.2 Shift Operators

- `>>` $\rightarrow$ **shift right**
- `<<` $\rightarrow$ **shift left**

- **Result is same size as first operand,** always zero filled

Example:

```
      a = 4'b1010;
      ...
      d = a >> 2;      // d = 0010
      c = a << 1;      // c = 0100
```

1.2 Concatenation Operator

- **{op1, op2, ..}** → **concatenates op1, op2, .. to single number**
- **Operands must be sized !!**

```
reg a;
reg [2:0] b, c;
..
a = 1’b 1;
b = 3’b 010;
c = 3’b 101;
catx = {a, b, c};         // catx = 1_010_101
caty = {b, 2’b11, a};     // caty = 010_11_1
catz = {b, 1};            // WRONG !!
```

- **Replication ..**

```
catr = {4{a}, b, 2{c}};   // catr = 1111_010_101101
```

1.2 Relational Operators

- > → **greater than**
- < → **less than**
- >= → **greater or equal than**
- <= → **less or equal than**

- **Result is one bit value: *0*, *1* or *x***

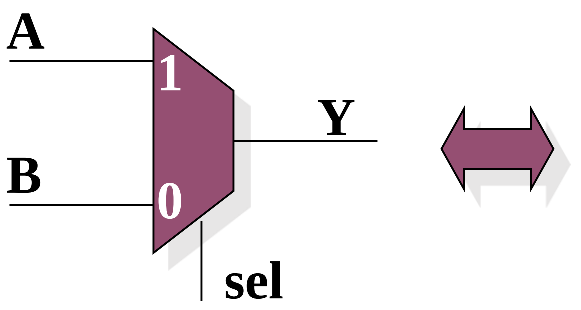
```
1 > 0              → 1
'b1x1 <= 0         → x
10 < z             → x
```

1.2 Equality Operators

- `==` → **logical equality**
- `!=` → **logical inequality**
- `===` → **case equality**
- `!==` → **case inequality**
- **example**
  - `4'b 1z0x == 4'b 1z0x` → ***x***
  - `4'b 1z0x != 4'b 1z0x` → ***x***
  - `4'b 1z0x === 4'b 1z0x` → ***1***
  - `4'b 1z0x !== 4'b 1z0x` → ***0***

1.2 Conditional Operator

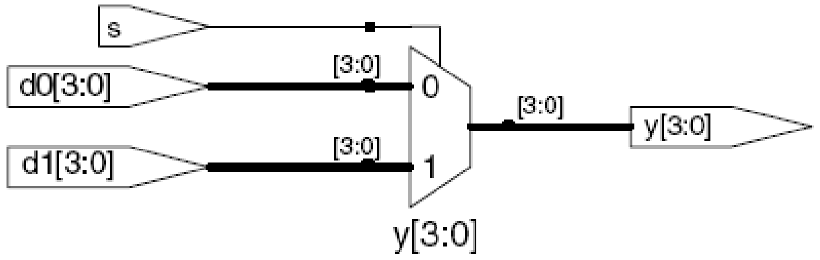
- `cond_expr ? true_expr : false_expr`
- **Like a 2-to-1 mux ..**



```
Y = (sel)? A : B;
```

1.2 Conditional Operator

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



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

1.2 Arithmetic Operators

- + → **Arithmetic addition**
- - → **Arithmetic subtraction**
- * → **Arithmetic multiplication**
- / → **Arithmetic division**
- % → **Modular arithmetic**
- **If any operand is *x* the result is *x***
- **Negative registers:**
  - **regs can be assigned negative but are treated as unsigned**

```
reg [15:0] regA;
..
regA = -4'd12;      // stored as 2^16-12 = 65524
regA/3              //evaluates to 21841
```

1.2 Arithmetic Operators

- **Negative integers:**
  - **can be assigned negative values**
  - **different treatment depending on base specification or not**

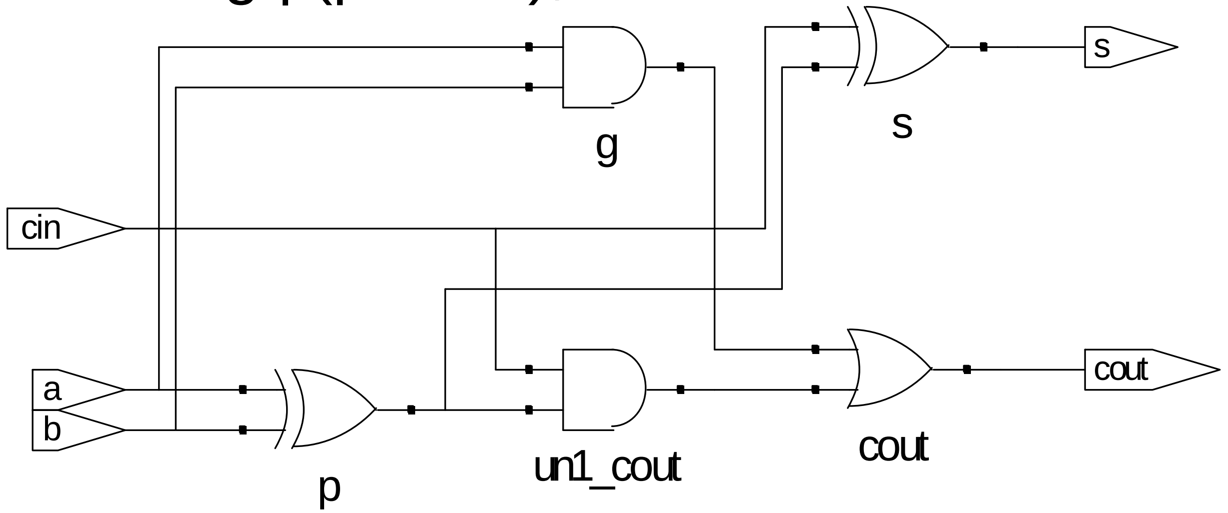
```
reg [15:0] regA;
integer intA;
..
intA = -12/3;   // evaluates to -4 (no base spec)
```

1.2 Internal Variables

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

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

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



1.2 Precedence

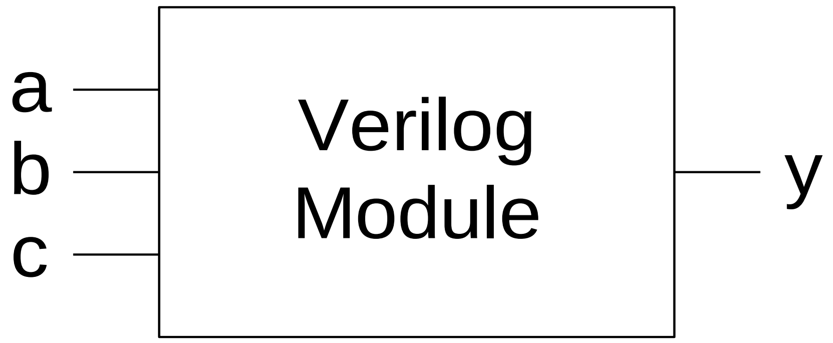
Order of Operators

Highest

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

Lowest

1.3 Verilog Modules



- **Two types of Modules:**
  - **Behavioral: describe what a module does**
  - **Structural: describe how it is built from simpler modules**

1.3 Modules Declaration

```
module full_add (A, B, CI, S, CO) ;

input A, B, CI ;
output S, CO ;

wire N1, N2, N3;

half_add HA1 (A, B, N1, N2),
         HA2 (N1, CI, S, N3);

or P1 (CO, N3, N2);

endmodule
```

```
module half_add (X, Y, S, C);

input X, Y ;
output S, C ;

xor  (S, X, Y) ;
and (C, X, Y) ;

endmodule
```

1.3 Behavioral Verilog

- **Verilog:**

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

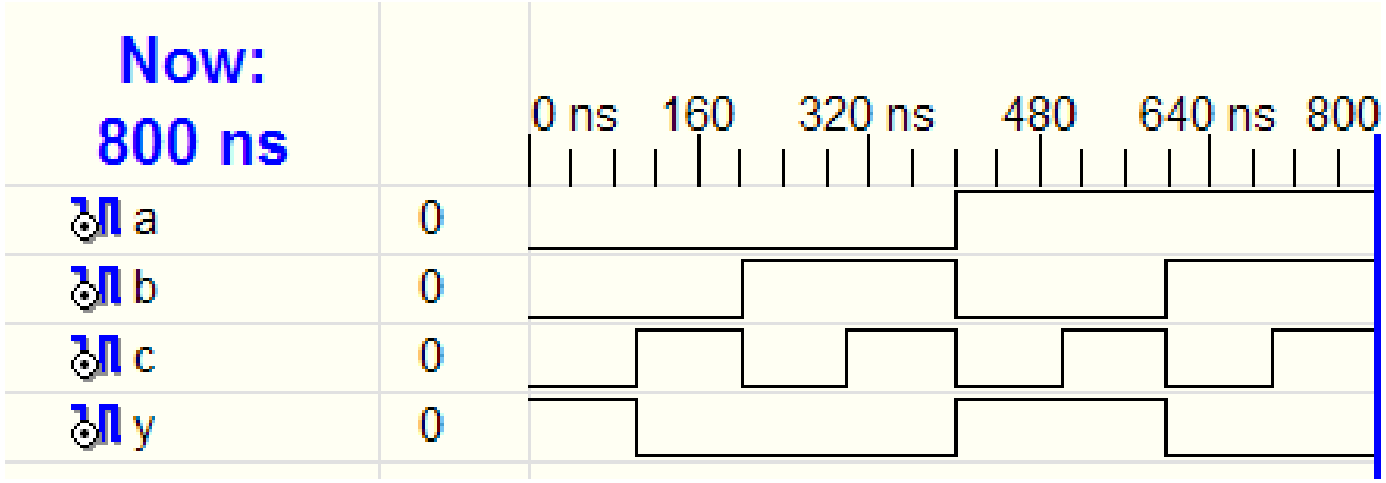
- **module/endmodule: required to begin/end module**
- **example: name of the module**
- **Operators:**

  ~: NOT
  &: AND
  |: OR

1.3 HDL Simulation

- **Verilog:**

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

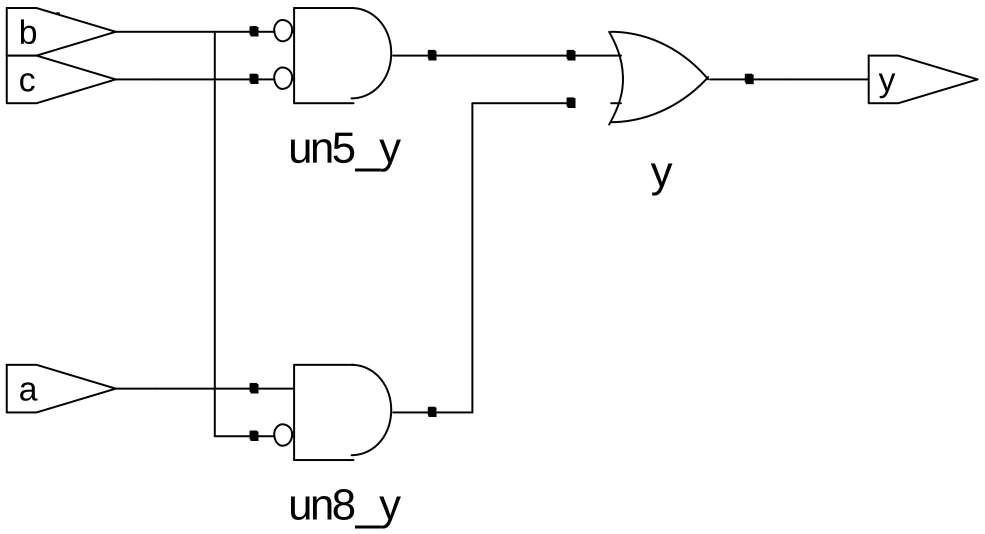


1.3 HDL Synthesis

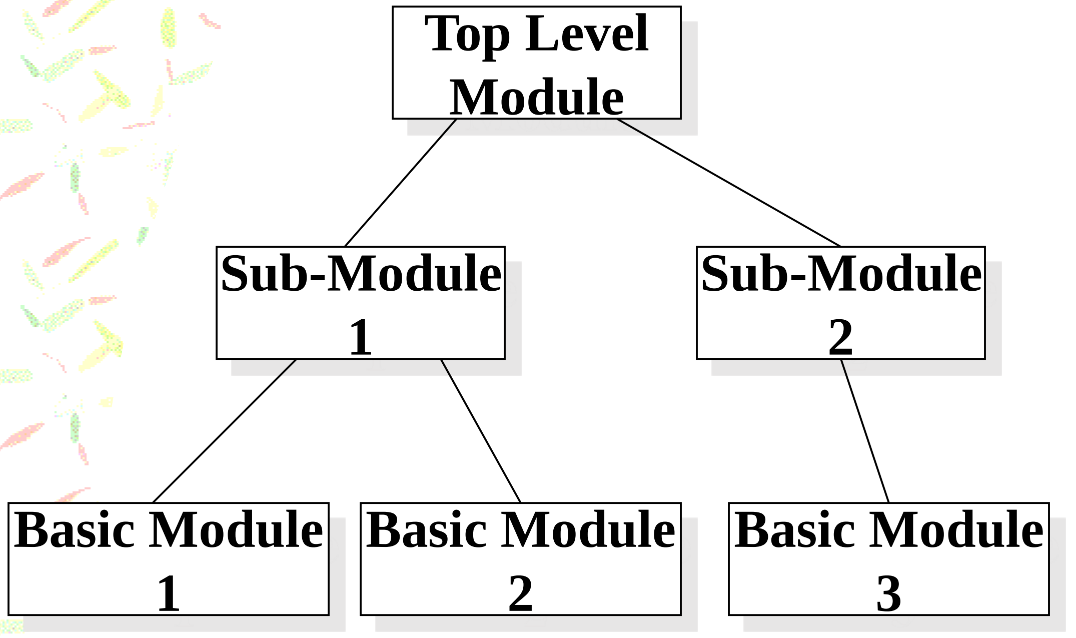
- **Verilog:**

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

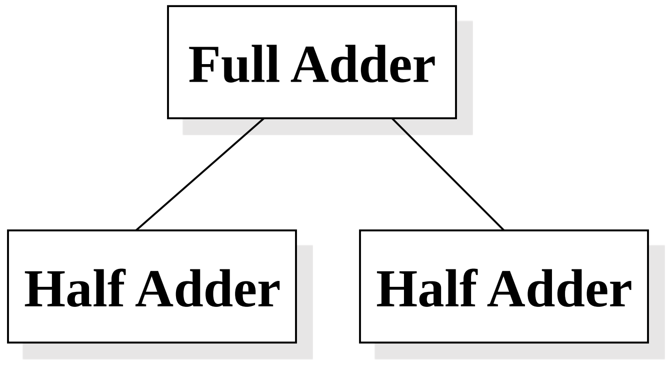
- **Synthesis:**



1.3 Structural Modeling - Hierarchy



E.g.



- Concurrency
- Structure
- Procedural Statements
- Time

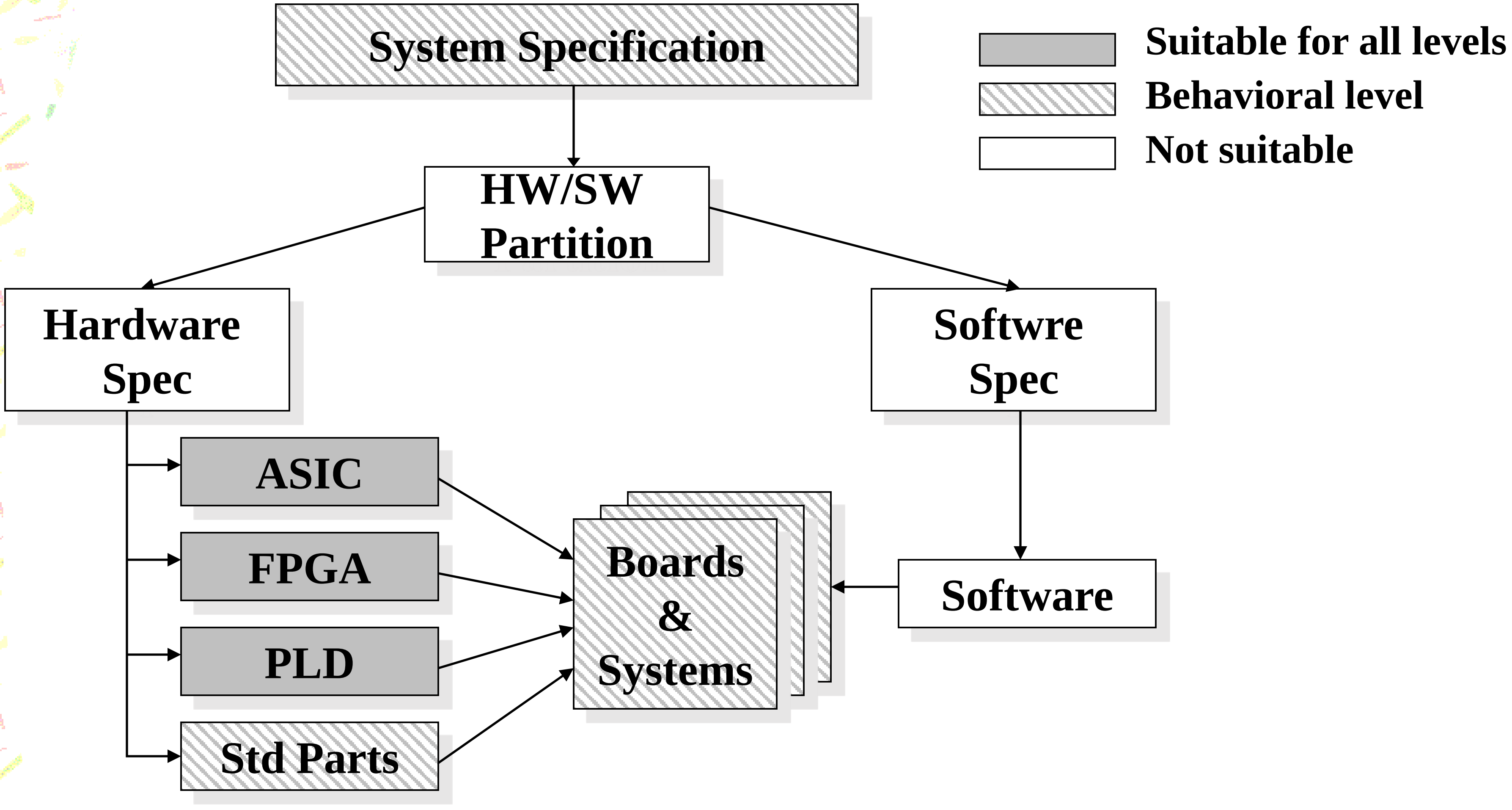
1.3 Structural Modeling - Hierarchy

```
module and3(input a, b, c, output y);
      assign y = a & b & c;
endmodule

module inv(input a, output y);
      assign y = ~a;
endmodule

module nand3(input a, b, c output y);
      wire n1;                  // internal signal
      and3 andgate(a, b, c, n1);  // instance of and3
      inv  inverter(n1, y);      // instance of inv
endmodule
```

2 Verilog Application



2.1 Sequential Logic

- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

2.1 Always Statement

General Structure:

```
always @(sensitivity list)
   statement;
```

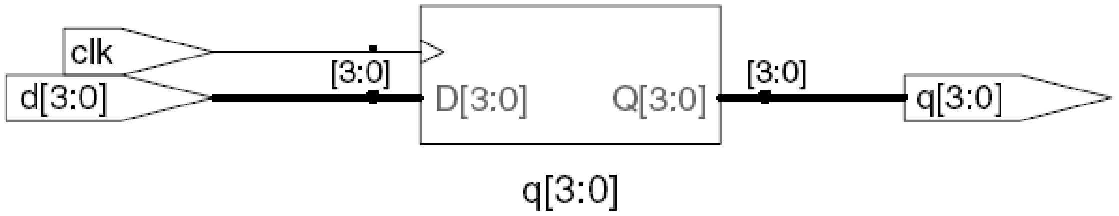
Whenever the event in `sensitivity list` **occurs,** `statement` **is executed**

2.1 D Flip-Flop

```
module flop(input           clk,
            input   [3:0] d,
            output reg [3:0] q);

    always @(posedge clk)
        q <= d;             // pronounced "q gets d"

endmodule
```

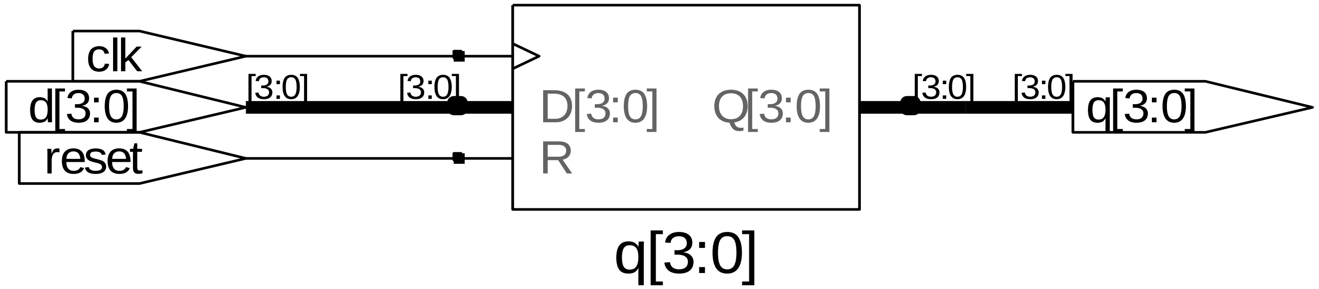


2.1 Resettable D Flip-Flop

```
module flopr(input       clk,
             input       reset,
             input    [3:0] d,
             output  reg [3:0] q);

    // synchronous reset
    always@(posedge clk)
        if (reset) q <= 4'b0;
        else       q <= d;

endmodule
```

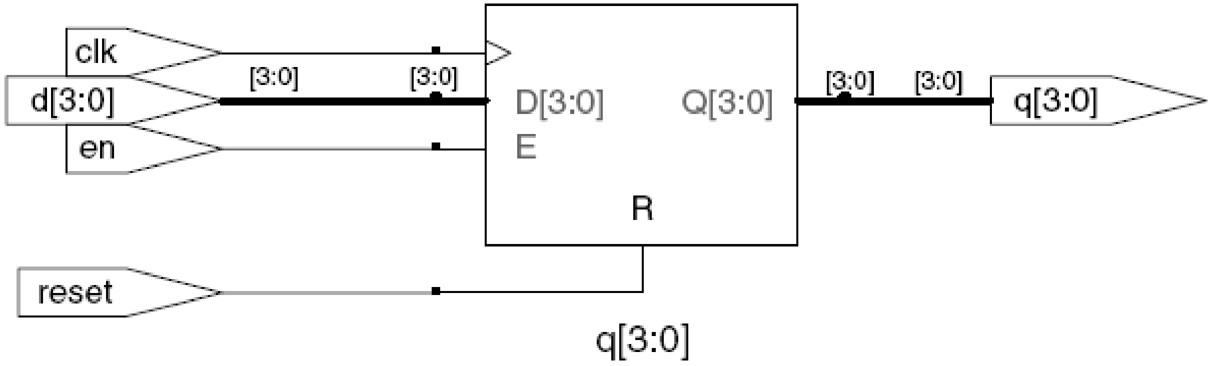


2.1 D Flip-Flop with Enable

```
module flopren(input         clk,
               input     reset,
               input         en,
               input   [3:0] d,
               output reg [3:0] q);

    // asynchronous reset and enable
    always @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else if (en)    q <= d;

endmodule
```

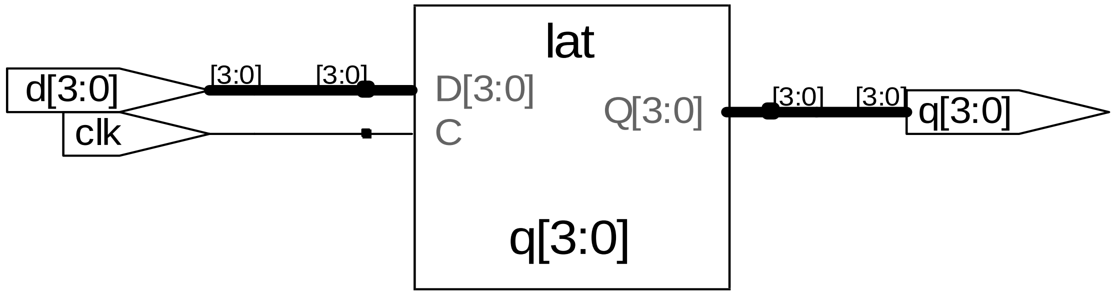


2.1 Latch

```
module latch(input        clk,
             input   [3:0] d,
             output reg [3:0] q);

    always@(clk,d)
        if (clk) q <= d;

endmodule
```



2.1 Other Behavioral Statements

- **Statements that must be inside always statements:**
  - **if / else**
  - **case, casez**

2.2 Combinational Logic using always

```
// combinational logic using an always statement
module gates(input    [3:0] a, b,
                      output  reg [3:0] y1, y2, y3, y4, y5);
      always@(*)       // need begin/end because there is
       begin             // more than one statement in always
            y1 = a & b;       // AND
            y2 = a | b;        // OR
            y3 = a ^ b;       // XOR
            y4 = ~(a & b); // NAND
            y5 = ~(a | b);  // NOR
       end
endmodule
```

This hardware could be described with assign statements using fewer lines of code, so it's better to use assign statements in this case.

2.2 Combinational Logic using case

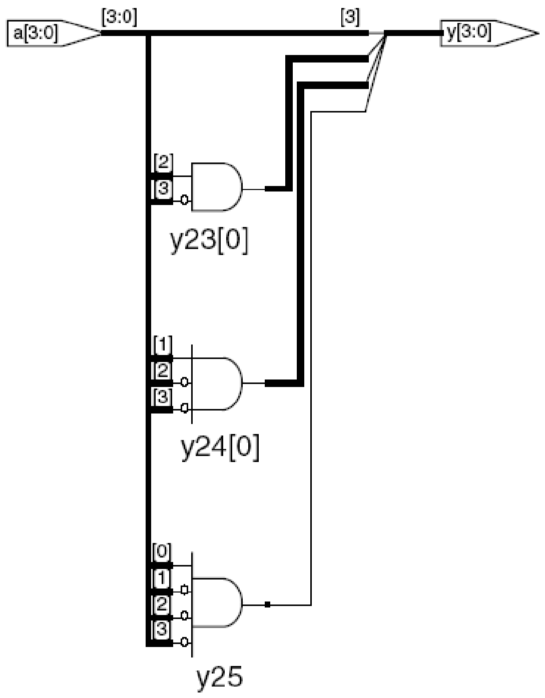
```
module sevenseg(input   [3:0] data,
                        output reg [6:0] segments);
      always@(*)
          case (data)
              0: segments =     7'b111_1110;     //   abc_defg
              1: segments =     7'b011_0000;
              2: segments =     7'b110_1101;
              3: segments =     7'b111_1001;
              4: segments =     7'b011_0011;
              5: segments =     7'b101_1011;
              6: segments =     7'b101_1111;
              7: segments =     7'b111_0000;
              8: segments =     7'b111_1111;
              9: segments =     7'b111_0011;
              default: segments = 7'b000_0000; // required
          endcase
endmodule
```

2.2 Combinational Logic using case

- **case statement implies combinational logic only if all possible input combinations described**
- **Remember to use default statement**

2.2 Combinational Logic using casez

```
module priority_casez(input   [3:0] a,
                              output reg [3:0] y);
       always@(*)
          casez(a)
              4'b1???: y = 4'b1000;  // ? = don't care
              4'b01??: y = 4'b0100;
              4'b001?: y = 4'b0010;
              4'b0001: y = 4'b0001;
              default: y = 4'b0000;
          endcase
endmodule
```

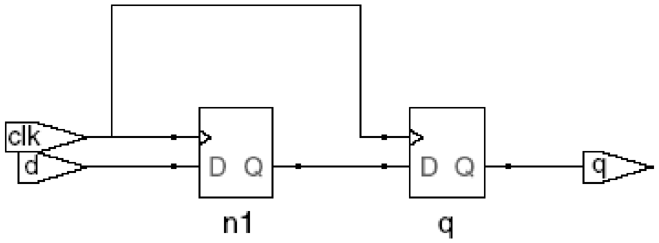


2.2 Blocking vs. Nonblocking Assignment

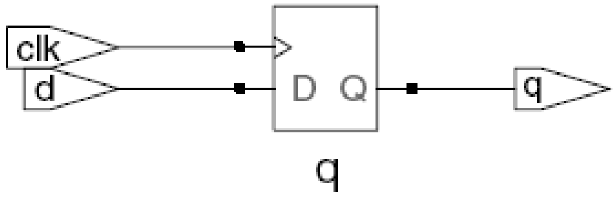
- **<= is nonblocking assignment**
  - **Occurs simultaneously with others**
- **= is blocking assignment**
  - **Occurs in order it appears in file**

2.2 Blocking vs. Nonblocking Assignment

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  clk,
                input     d,
                output  reg q);
   wire n1;
   always @(posedge clk)
      begin
         n1 <= d;  // nonblocking
         q  <= n1; // nonblocking
      end
endmodule
```



```
// Bad synchronizer using
// blocking assignments
module syncbad(input clk,
               input    d,
               output  reg q);
   wire n1;
   always @(posedge clk)
      begin
         n1 = d;  // blocking
         q  = n1; // blocking
      end
endmodule
```



2.2 Rules for Assignment

- **Synchronous sequential logic: use** always @(posedge clk) **and nonblocking assignments (**<=**)**

```
always @ (posedge clk)
q <= d; // nonblocking
```

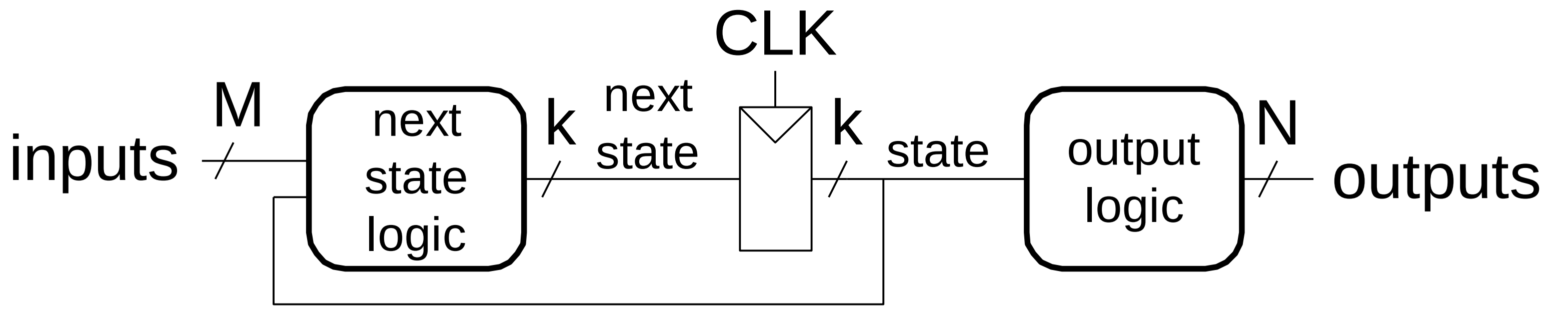
- **Simple combinational logic: use continuous assignments (**assign…**)**

```
assign y = a & b;
```

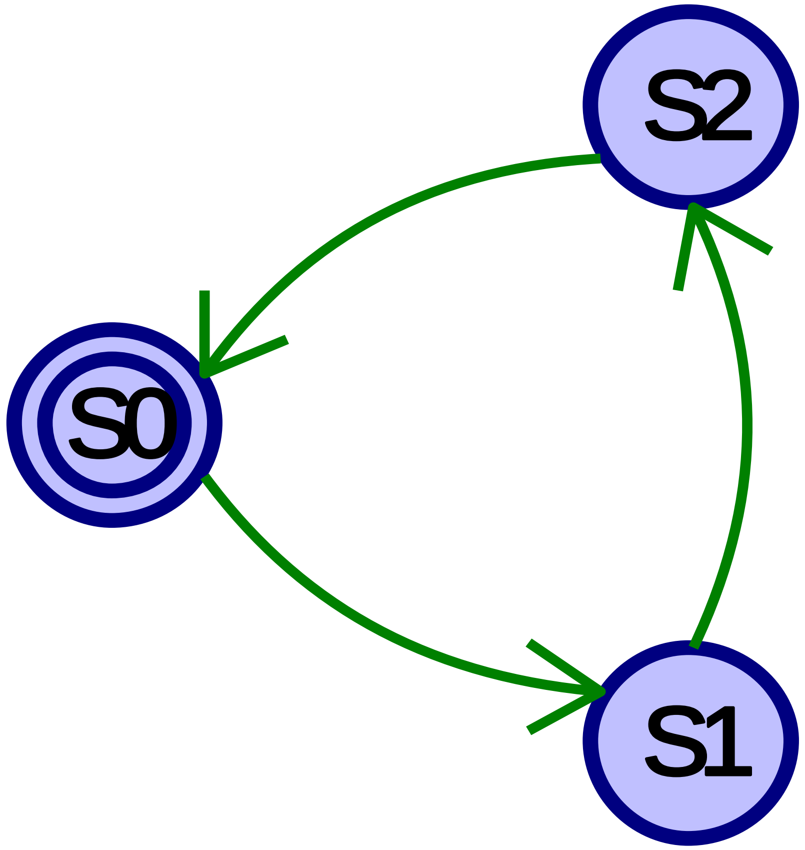
- **More complicated combinational logic: use** always@(*) **and blocking assignments (**=**)**
- **Assign a signal in only one always statement or continuous assignment statement.**

2.3 Finite State Machines(FSM)

- **Three blocks:**
  - **next state logic**
  - **state register**
  - **output logic**



2.3 FSM example: Divide by 3

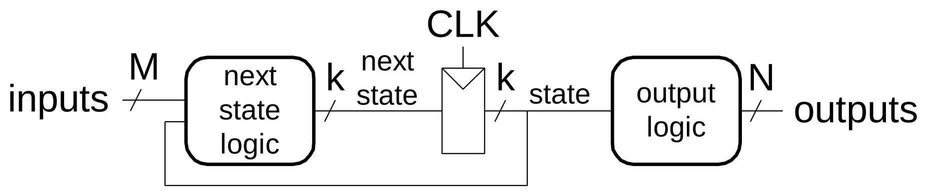


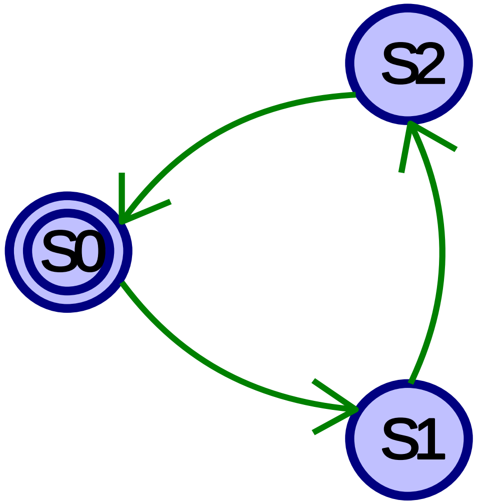
The double circle indicates the reset state

2.3 FSM in Verilog

```
module divideby3FSM (input clk,
                             input  reset,
                             output q);
  reg [1:0] state, nextstate;
  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  // state register
  always@ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;
 // next state logic
  always@(*)
    case (state)
      S0:     nextstate = S1;
      S1:     nextstate = S2;
      S2:     nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign q = (state == S0);
endmodule
```





2.4 Parameterized Modules

2:1 mux:

```
module mux2
    #(parameter width = 8)  // name and default value
      (input [width-1:0] d0, d1,
       input                     s,
       output [width-1:0]        y);
    assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
    mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
    mux2 #(12) lowmux(d0, d1, s, out);
```

2.5 Testbenches

- **HDL that tests another module:** *device under test* **(dut)**
- **Not synthesizeable**
- **Types:**
  - **Simple**
  - **Self-checking**
  - **Self-checking with testvectors**

2.5 Testbench Example

- **Write SystemVerilog code to implement the following function in hardware:**

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

- **Name the module sillyfunction**

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

2.5 Simple Testbench

```
module testbench1();
      reg a, b, c;
      wire y;
      // instantiate device under test
      sillyfunction 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
```

2.5 Self-check Testbench

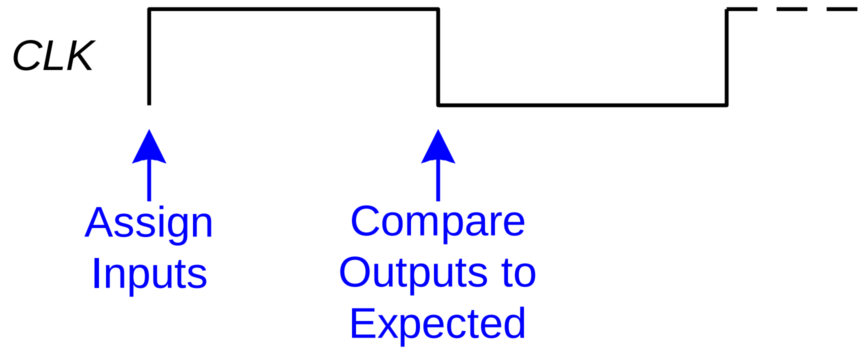
```
module testbench2();
      reg  a, b, c;
      wire y;
      sillyfunction dut(a, b, c, y);  // instantiate dut
      initial begin       //check results one at a time
          a = 0; b = 0; c = 0; #10;
          if (y !== 1) $display("000 failed.");
              c = 1; #10;
          if (y !== 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
```

2.5 Testbench with Testvectors

- **Testvector file: inputs and expected outputs**
- **Testbench:**
  - **Generate clock for assigning inputs, reading outputs**
  - **Read testvectors file into array**
  - **Assign inputs, expected outputs**
  - **Compare outputs with expected outputs and report errors**

2.5 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**

2.5 Testvector 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
```

2.5 (1) Generate Clock

```
module testbench3();
    reg       clk, reset;
    reg       a, b, c, yexpected;
    wire       y;
    reg [31:0] vectornum, errors;    // book keeping variables
    reg [3:0]  testvectors[10000:0]; // array of testvectors

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

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

2.5 (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; #27; reset = 0;
    end


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

2.5 (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
```

2.5 (4) Compared 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});
```

2.5 (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);
            $finish;
        end
    end
endmodule
```

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