

Lecture 2

Hardware Design with SystemVerilog

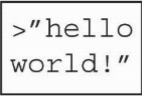


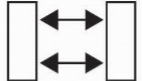
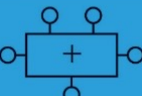
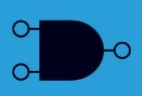
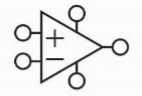


Prof Peter YK Cheung
Imperial College London

URL: www.ee.ic.ac.uk/pcheung/teaching/EIE2-IAC/
E-mail: p.cheung@imperial.ac.uk

Learning outcome for week 2

- ❖ Modules in SystemVerilog (SV)
- ❖ Syntax, operators, number formats in SV
- ❖ Behavioural vs structural description
- ❖ Combinational circuit description
- ❖ Sequential circuit description
- ❖ Blocking and non-blocking assignment

Slides in this lecture are derived and modified from:
“*Digital Design and Computer Architecture (RISC-V Edition)*” by Sarah Harris and David Harris (H&H), Morgan Kaufmann, 2022

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Hardware description Languages

❖ Hardware description language (HDL):

- Specifies logic function only
- Computer-aided design (CAD) tool produces or synthesizes the optimized gates

❖ Most commercial designs use HDLs

❖ Two leading HDLs:

- **SystemVerilog**
 - Developed in 1984 by Gateway Design Automation (Verilog)
 - 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)

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)

❖ Physical design

- Placement, routing, chip layout, – not considered in this module

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

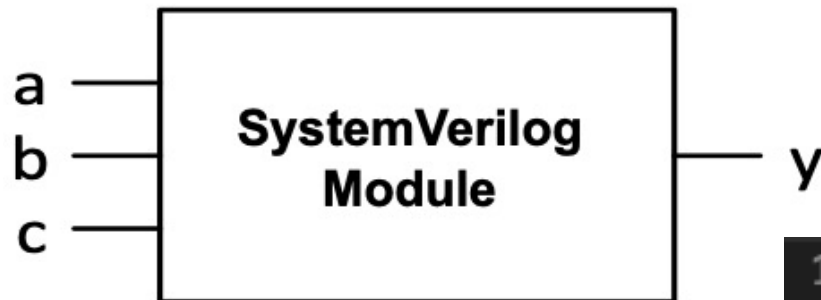
Beware of treating HDL like software and coding without thinking of the hardware.

H&H 171-173

SystemVerilog: Module Declaration

❖ Two types of Modules:

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



```
1  module example(input  logic a, b, c,  
2  |              |              |      output logic y);  
3  |              |              |      // module body goes here  
4  endmodule
```

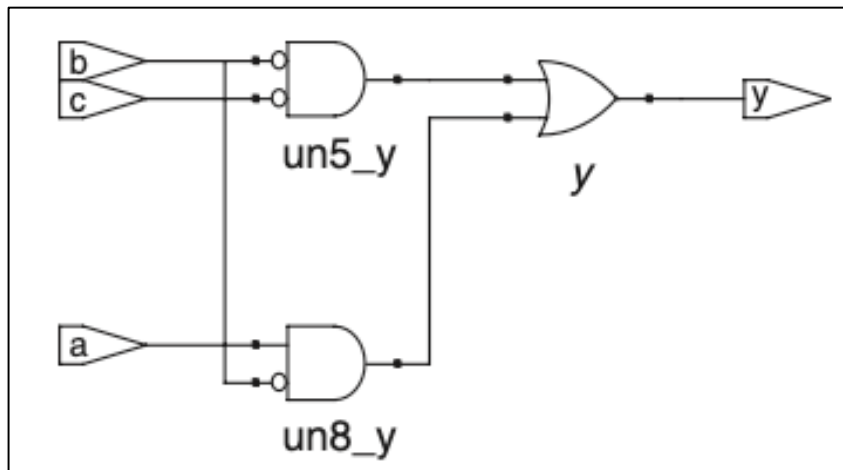
❖ **module/endmodule:** required to begin/end module

❖ **example:** name of the module

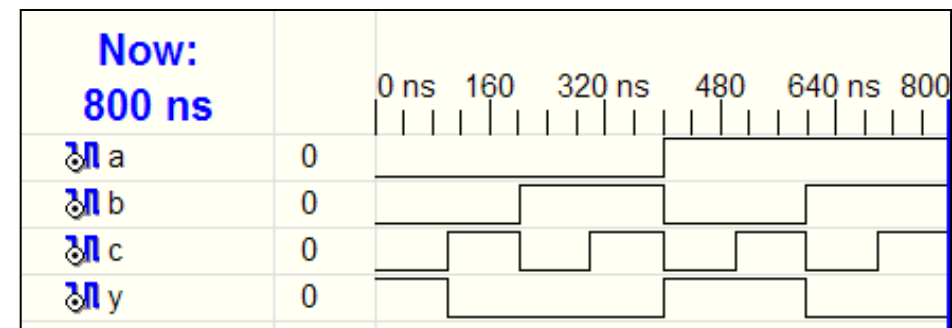
SystemVerilog: Behavioural Description

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

synthesis



simulation



Based on: "Digital Design and Computer Architecture (RISC-V Edition)"
by Sarah Harris and David Harris (H&H),

H&H 174

SystemVerilog: Syntax

❖ Case sensitive

- e.g.: reset and Reset are not the same signal.

❖ No names that start with numbers

- e.g.: 2mux is an invalid name

❖ Whitespace ignored

❖ Comments:

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

SystemVerilog: Structural Description

Behavioural

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

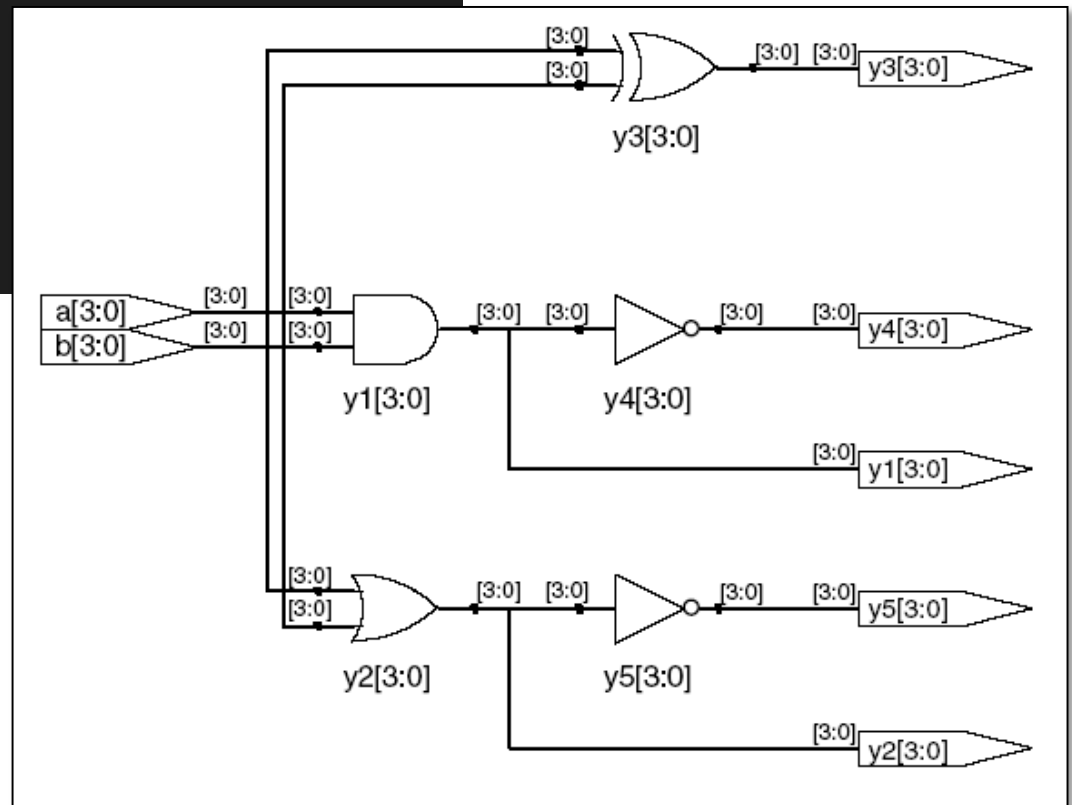
```
module inv(input logic a,  
          output logic y);  
    assign y = ~a;  
endmodule
```

Structural

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

SystemVerilog: Bitwise Operators

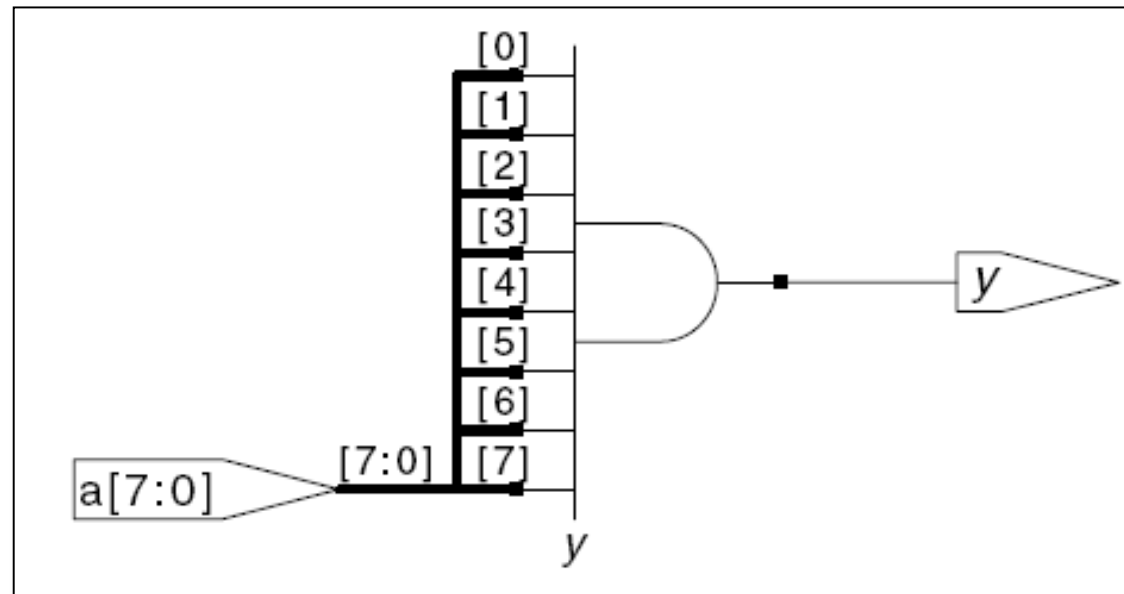
```
module gates(input logic [3:0] a, b,  
            output logic [3:0] y1, y2, y3, y4, y5);  
    /* Five different two-input logic  
       gates acting on 4 bit busses */  
    assign y1 = a & b;    // AND  
    assign y2 = a | b;    // OR  
    assign y3 = a ^ b;    // XOR  
    assign y4 = ~(a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



H&H 177

SystemVerilog: Reduction Operators

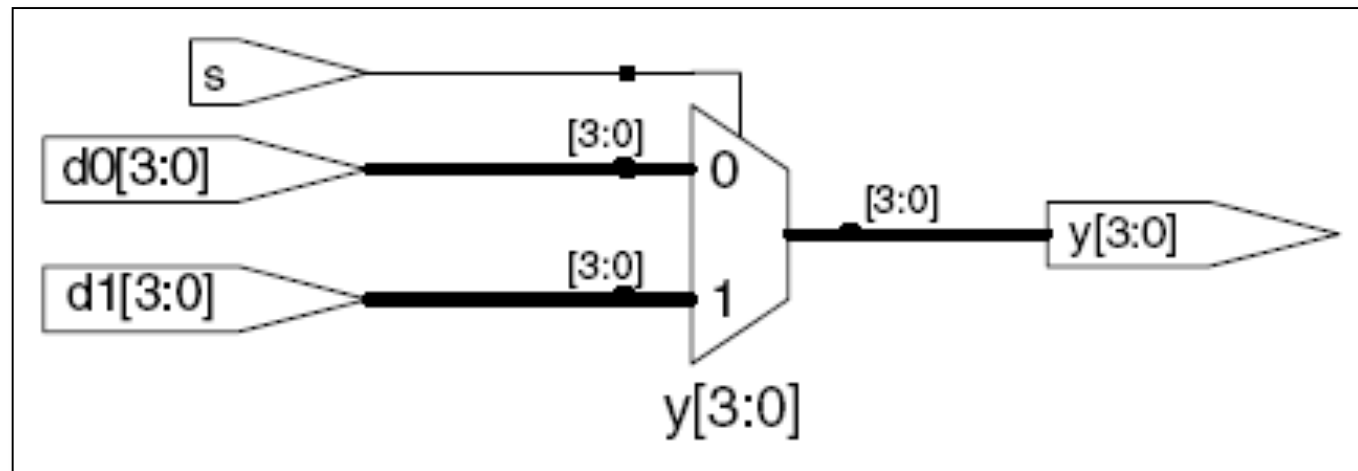
```
module and8(input logic [7:0] a,  
           output logic y);  
  assign y = &a;  
  // &a is much easier to write than  
  // assign y = a[7] & a[6] & a[5] & a[4] &  
  //           a[3] & a[2] & a[1] & a[0];  
endmodule
```



H&H 178

SystemVerilog: Conditional Assignment

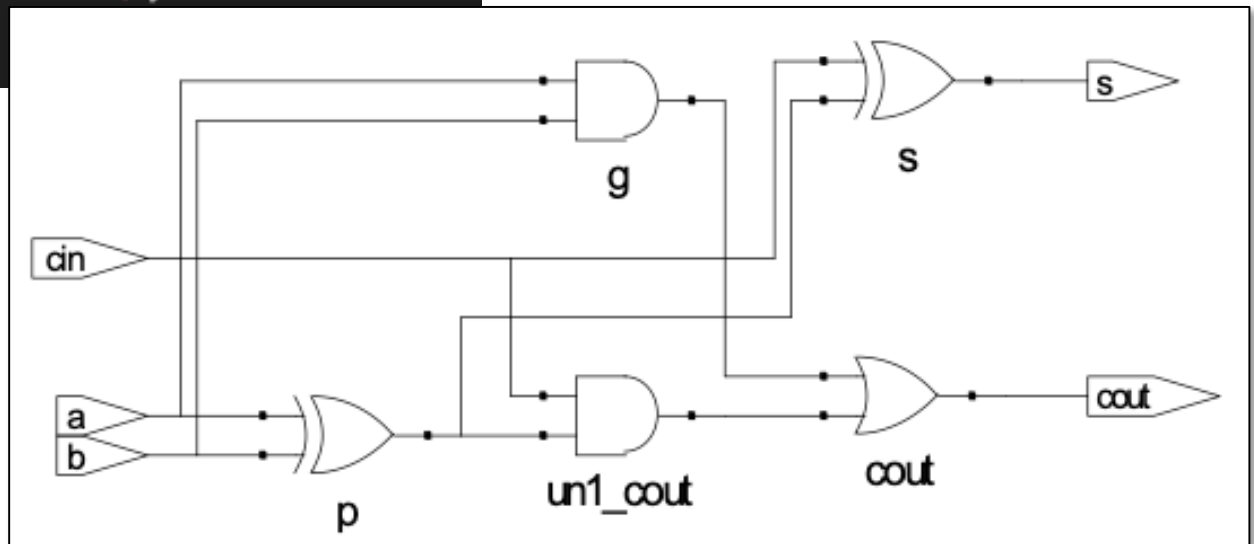
```
module mux2(input logic [3:0] d0, d1,  
           input logic      s,  
           output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```



H&H 179

SystemVerilog: Internal Signals

```
module fulladder(input logic a, b, cin,  
                output logic s, cout);  
    logic p, g; // internal nodes  
  
    assign p = a ^ b;  
    assign g = a & b;  
  
    assign s = p ^ cin;  
    assign cout = g | (p & cin);  
endmodule
```



H&H 182

SystemVerilog: Precedence 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

H&H 183

SystemVerilog: Number Format

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	Unsize	decimal	42	00...0101010

SystemVerilog: Bit Manipulations (1)

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

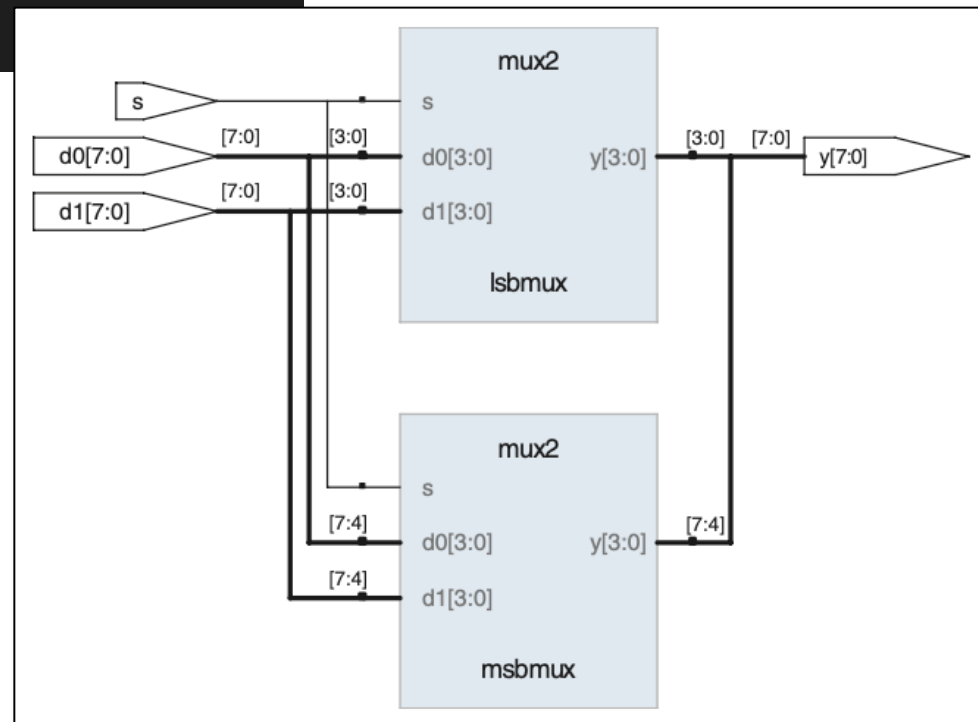
❖ If y is a 12-bit signal, the above statement produces:

```
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. **SystemVerilog ignores them.**

SystemVerilog: Bit Manipulations (2)

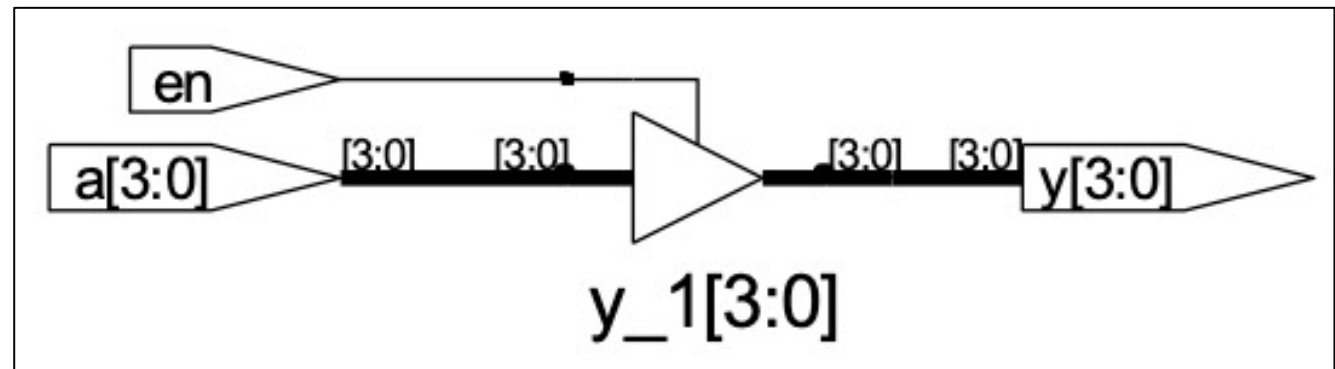
```
module mux2_8(input logic [7:0] d0, d1,  
             input logic      s,  
             output logic [7:0] y);  
  
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);  
endmodule
```



H&H 190

SystemVerilog: Floating Output Z

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



❖ Note that Verilator does not handle floating output Z

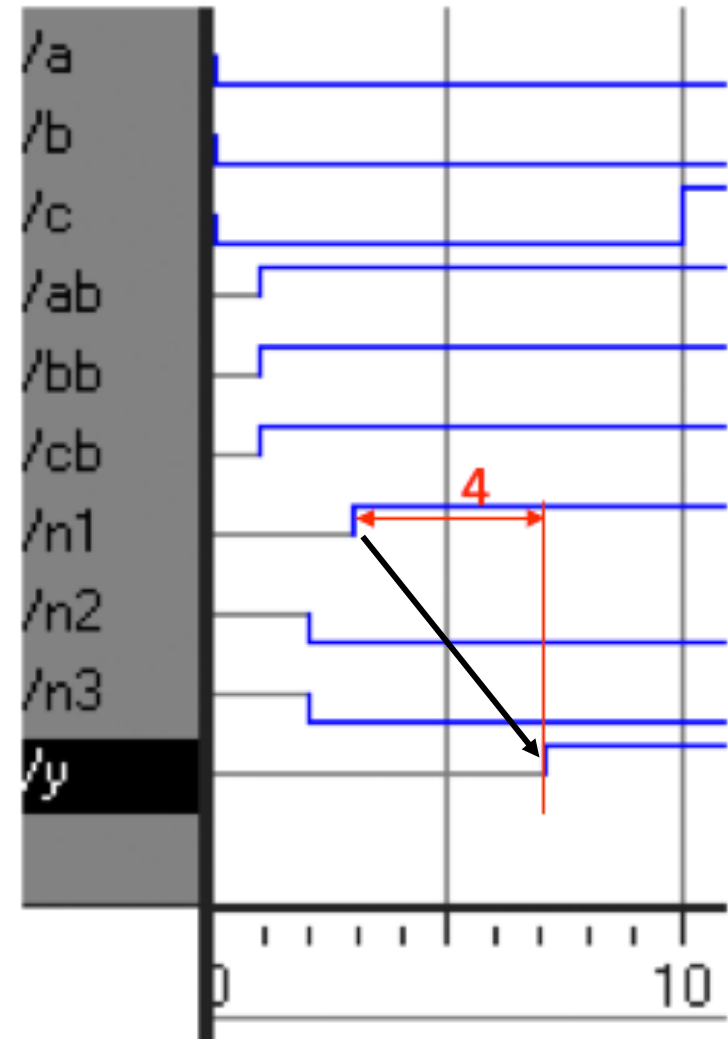
H&H 185

SystemVerilog: Delays

```
module example(input logic a, b, c,  
              output logic y);  
    logic 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
```

- ❖ Delays are for simulation only! They do not determine the delay of your hardware.
- ❖ **Verilator simulator ignores delays** – it is cycle accurate without timing.

H&H 187



SystemVerilog: Sequential Logic

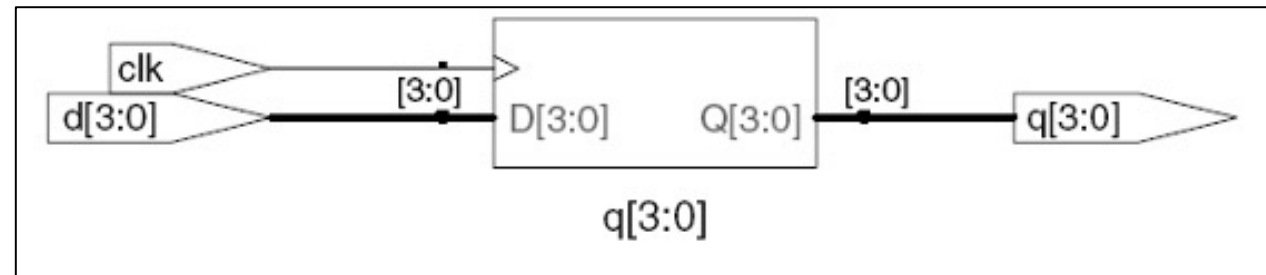
- ❖ SystemVerilog uses **idioms** (or special keywords or groups of words) to describe latches, flip-flops and FSMs
- ❖ Other coding styles may simulate correctly but produce incorrect hardware
- ❖ GENERAL STRUCTURE:

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

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

SystemVerilog: D Flip-Flop

```
module flop(input logic      clk,  
           input logic [3:0] d,  
           output logic [3:0] q);  
  
    always_ff @(posedge clk)  
        q <= d;           // pronounced "q gets d"  
  
endmodule
```



H&H 192

SystemVerilog: Resettable D Flip-Flop

Asynchronous reset

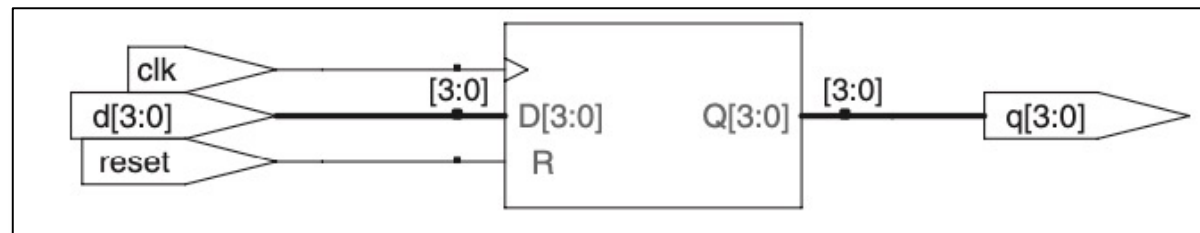
```
module flopr(input logic clk,
            input logic reset,
            input logic [3:0] d,
            output logic [3:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else q <= d;
endmodule
```

Synchronous reset

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

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



H&H 193

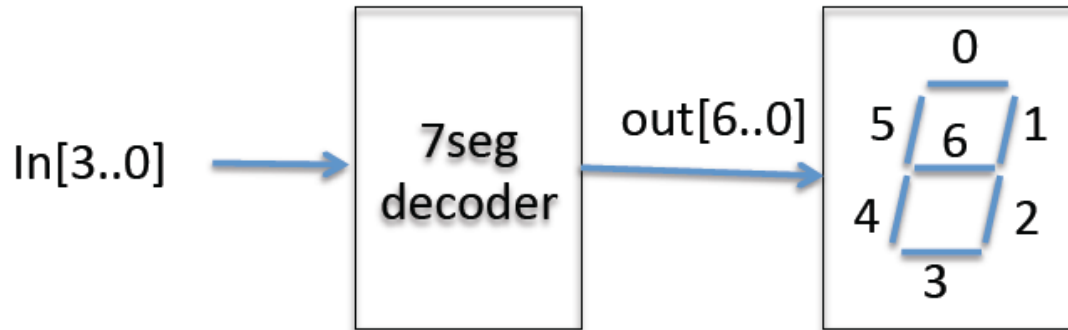
Combinational Logic using always

```
// combinational logic using an always statement
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
    always_comb // 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.

H&H 198

Combinational Logic using always-case



in[3..0]	out[6:0]	Digit	in[3..0]	out[6:0]	Digit
0000	1000000	0	1000	0000000	8
0001	1111001	1	1001	0010000	9
0010	0100100	2	1010	0001000	A
0011	0110000	3	1011	0000011	b
0100	0011001	4	1100	1000110	C
0101	0010010	5	1101	0100001	d
0110	0000010	6	1110	0000110	E
0111	1111000	7	1111	0001110	F

H&H 199

```

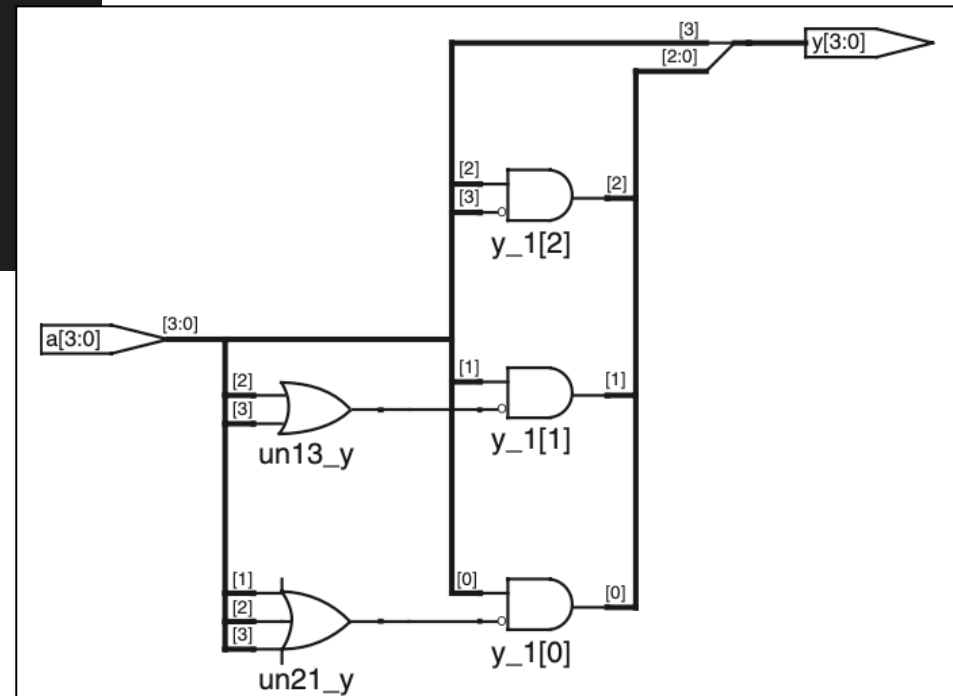
module hex_to_7seg (
    output logic [6:0] out,
    input logic [3:0] in);

    always_comb
        case (in)
            4'h0: out = 7'b1000000;
            4'h1: out = 7'b1111001;
            4'h2: out = 7'b0100100;
            4'h3: out = 7'b0110000;
            4'h4: out = 7'b0011001;
            4'h5: out = 7'b0010010;
            4'h6: out = 7'b0000010;
            4'h7: out = 7'b1111000;
            4'h8: out = 7'b0000000;
            4'h9: out = 7'b0011000;
            4'ha: out = 7'b0001000;
            4'hb: out = 7'b0000011;
            4'hc: out = 7'b1000110;
            4'hd: out = 7'b0100001;
            4'he: out = 7'b0000110;
            4'hf: out = 7'b0001110;
        endcase
endmodule
    
```

Combinational Logic using if-else

❖ Priority encoder circuit

```
module priorityckt( input logic [3:0] a,  
                   output logic [3:0] y);  
  always_comb  
    if (a[3])      y = 4'b1000;  
    else if (a[2]) y = 4'b0100;  
    else if (a[1]) y = 4'b0010;  
    else if (a[0]) y = 4'b0001;  
    else           y = 4'b0000;  
endmodule
```

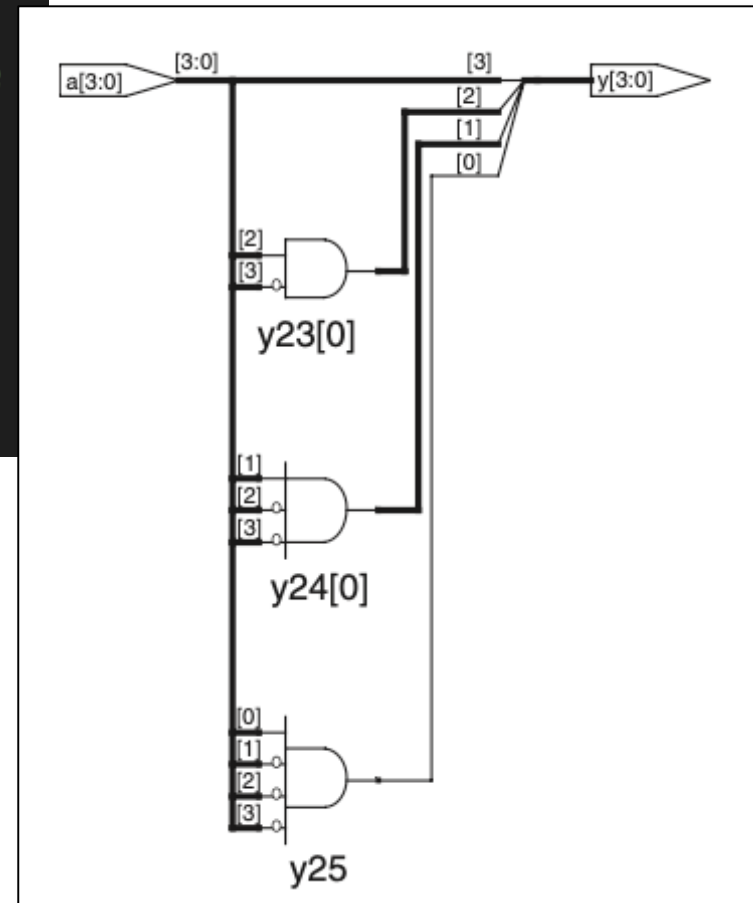


H&H 202

Combinational Logic using casez

```
module priority_casez(input  logic [3:0] a,  
                    output logic [3:0] y);  
    always_comb  
    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
```

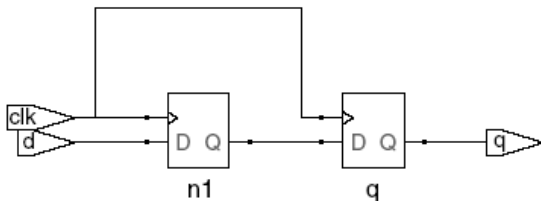
- ❖ ? = don't-care
- ❖ Beware: MUST have default statement in case not all cases are covered!



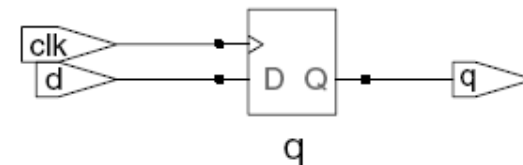
Blocking vs. Nonblocking Assignment

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

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



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



H&H 203

Rules for Signal Assignment

❖ Synchronous sequential logic, use:

`always_ff` and nonblocking assignments (`<=`)

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

❖ Simple combinational logic, use continuous assignments (`assign...`)

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

❖ More complicated combinational logic, use:

`always_comb` and blocking assignments (`=`)

❖ Assign a signal in **ONLY ONE** `always` statement or continuous assignment **statement**.