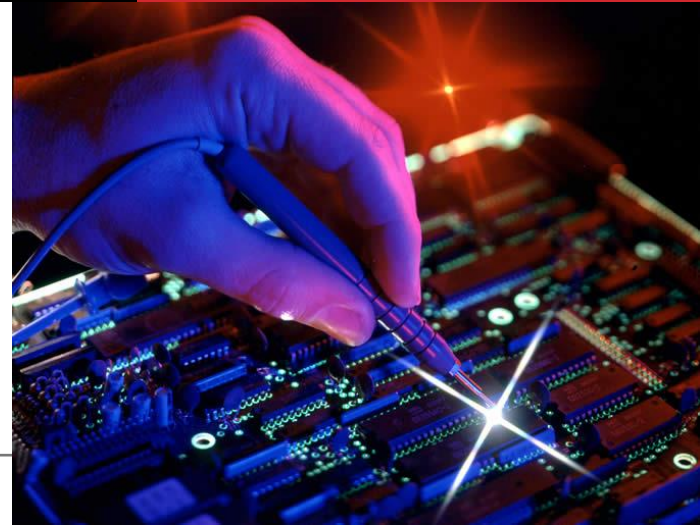




Computer Architecture & Microprocessor System

HARDWARE DESCRIPTION LANGUAGES AND VERILOG

Dennis A. N. Gookyi





CONTENTS

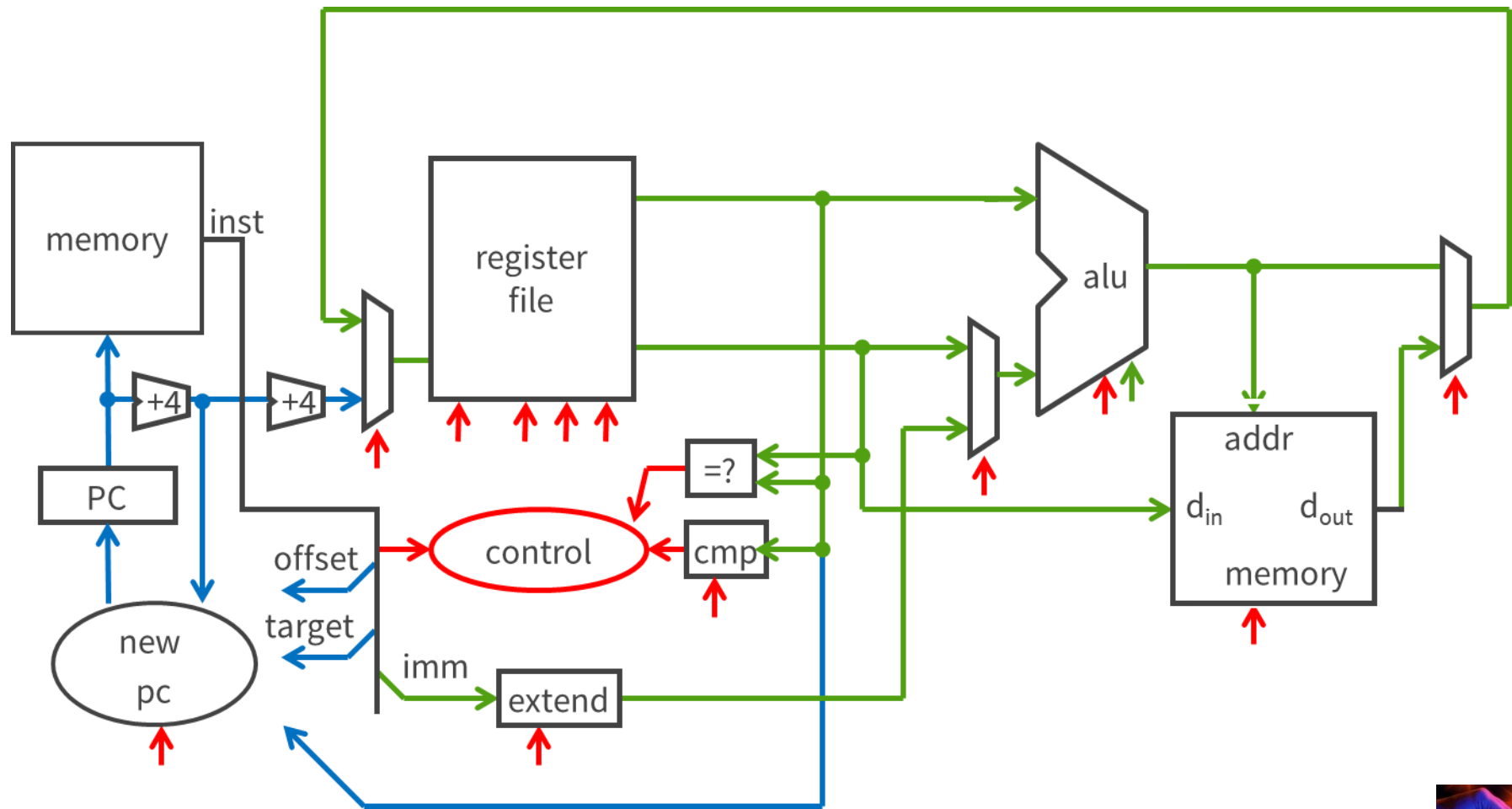
❖ **HARDWARE DESCRIPTION LANGUAGES AND VERILOG**





BIG PICTURE: BUILDING A PROCESSOR

❖ Single cycle processor





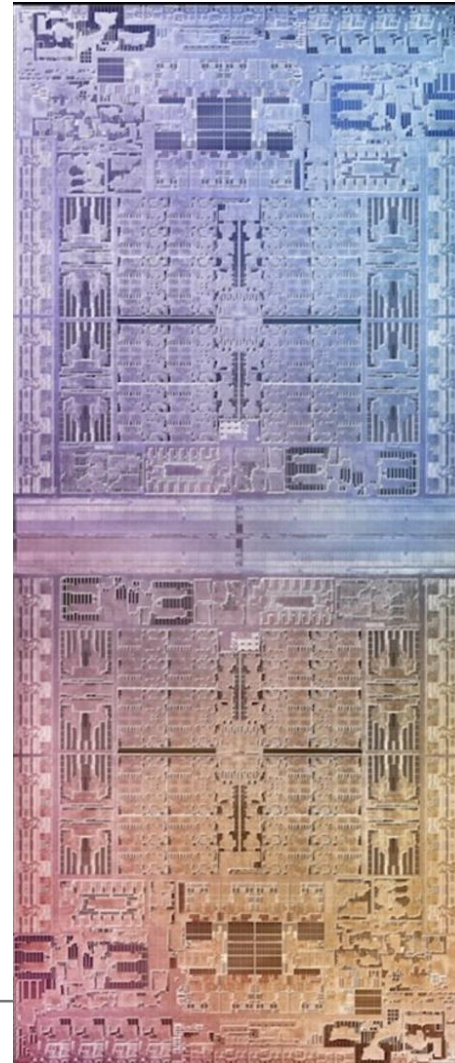
APPLE M1 ULTRA

❖ The internal architecture of Apple M1 Ultra:

□ How does it work?

- 16 High-Perf GP Cores
- 4 Efficient GP Cores
- 64-Core GPU
- 32-Core Neural Engine
- Lots of Cache
- Many Caches
- 32x Memory Channels
- 128 GB DRAM

□ **114B transistors**





TRANSISTOR COUNTS

❖ Transistor counts are growing

Year	Component	Name	Number of MOSFETs (in billions)
2022	Flash memory	Micron's V-NAND chip	5,333 (stacked package of sixteen 232-layer 3D NAND dies)
2020	any processor	Wafer Scale Engine 2	2,600 (wafer-scale design consisting of 84 exposed fields (dies))
2022	microprocessor (commercial)	M1 Ultra	114 (dual-die SoC; entire M1 Ultra is a multi-chip module)
2022	GPU	Nvidia H100	80
2020	DLP	Colossus Mk2 GC200	59.4

In terms of computer systems that consist of numerous integrated circuits, the supercomputer with the highest transistor count as of 2016 is the Chinese-designed Sunway TaihuLight, which has for all CPUs/nodes combined "about 400 trillion transistors in the processing part of the hardware" and "the DRAM includes about 12 quadrillion transistors, and that's about 97 percent of all the transistors."^[9] To compare, the smallest computer, as of 2018 dwarfed by a grain of rice, has on the order of 100,000 transistors.





DEALING WITH TRANSISTOR COUNT COMPLEXITY

- ❖ Hardware Description Languages (HDLs)
- ❖ What we need for hardware design:
 - Ability to describe (specify) complex designs
 - ... and to simulate their behavior (functional & timing)
 - ... and to synthesize (automatically design) portions of it
 - have an error-free path to implementation
- ❖ Hardware Description Languages enable all of the above
 - Languages designed to describe hardware
 - There are similarly-featured HDLs (e.g., Verilog, VHDL, ...)
 - if you learn one, it is not hard to learn another
 - mapping between languages is typically mechanical, especially for the commonly used subset





HDLS

- ❖ Two well-known hardware description languages
- ❖ Verilog
 - Developed in 1984 by Gateway Design Automation
 - Became an IEEE standard (1364) in 1995
 - More popular in US
- ❖ VHDL (VHSIC Hardware Description Language)
 - Developed in 1981 by the US Department of Defense
 - Became an IEEE standard (1076) in 1987
 - More popular in Europe
- ❖ We will use Verilog in this course





HDLS

- ❖ HDLs enable easy description of hardware structures
 - Wires, gates, registers, flip-flops, clock, rising/falling edge, ...
 - Combinational and sequential logic elements

- ❖ HDLs enable seamless expression of parallelism inherent in hardware
 - All hardware logic operates concurrently

- ❖ Both of the above ease specification, simulation & synthesis





HDLS

- ❖ Design a hierarchy of modules
 - Predefined “primitive” gates (AND, OR, ...)
 - Simple modules are built by instantiating these gates (e.g., components like MUXes)
 - Complex modules are built by instantiating simple modules, ...

- ❖ Hierarchy controls complexity
 - Analogous to the use of function/method abstraction in programming

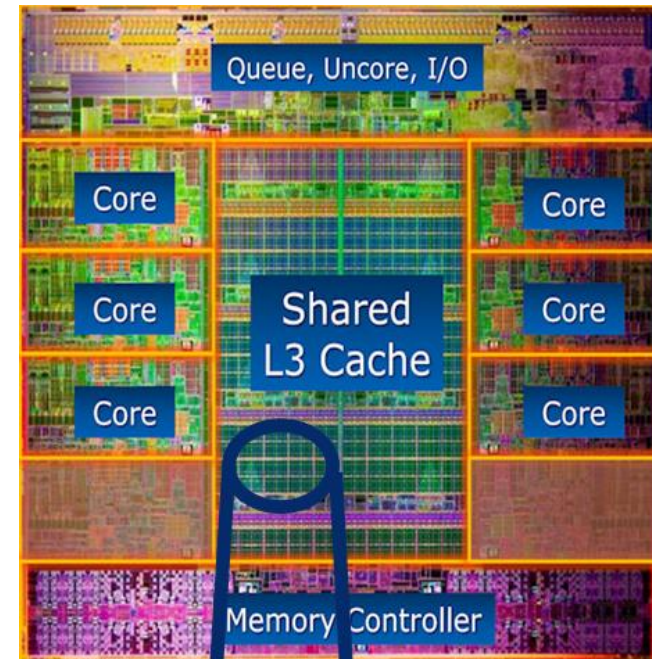
- ❖ Complexity is a BIG deal
 - In real world, how big is the size of a module (that is described in HDL and then synthesized to gates)?



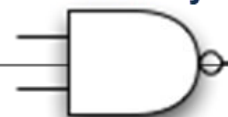


HDLS

- ❖ Design a hierarchy of modules



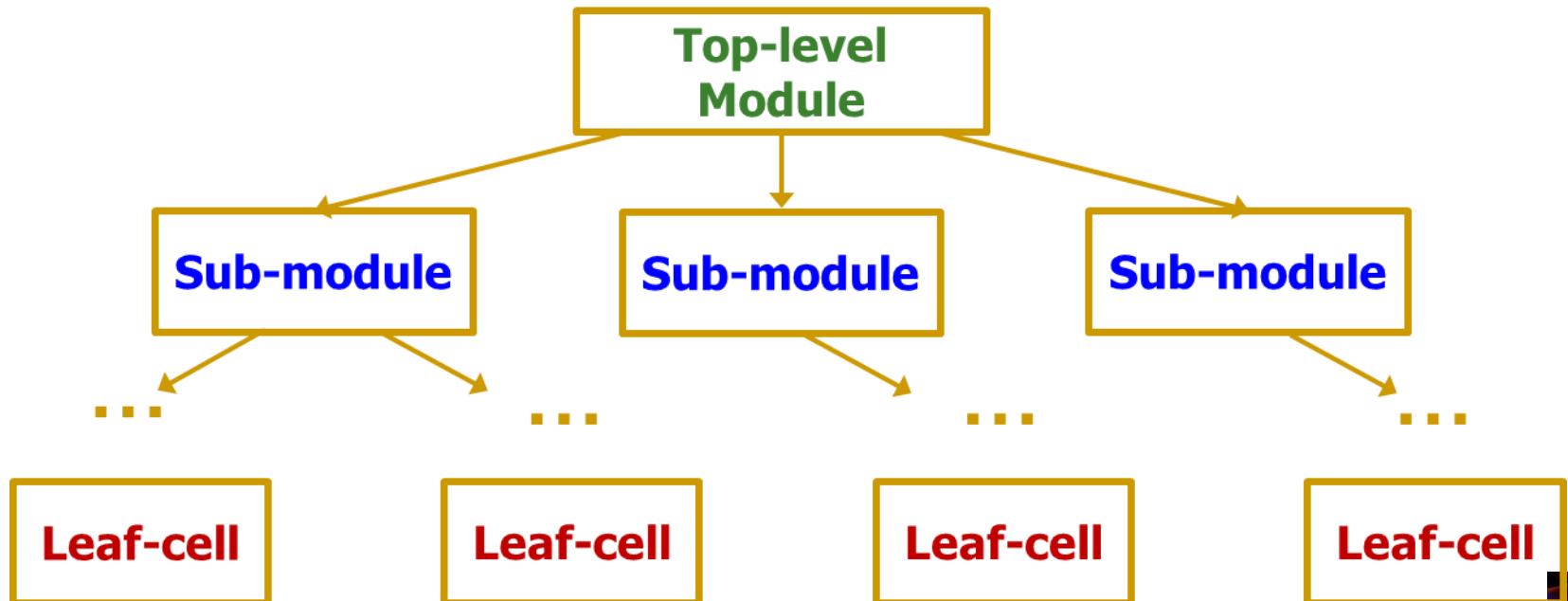
How many?





DESIGN METHODOLOGY: TOP-DOWN

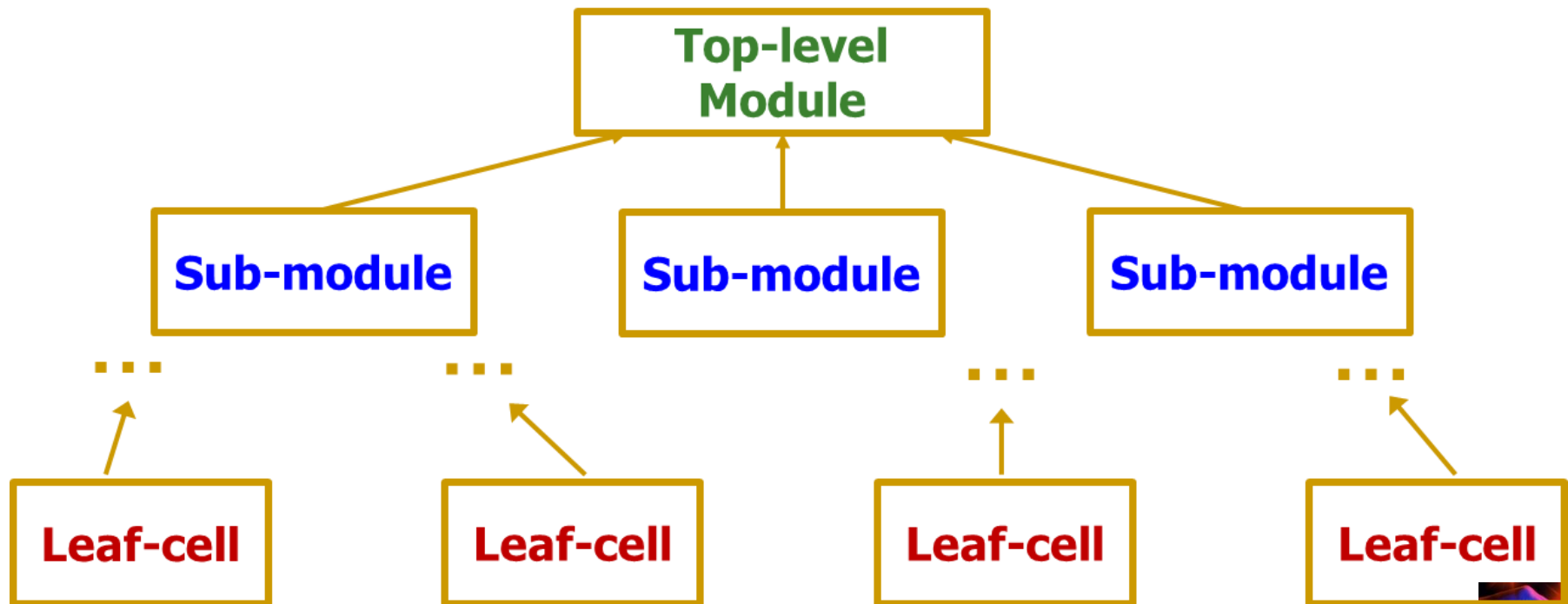
- ❖ We define the top-level module and identify the sub-modules necessary to build the top-level module
- ❖ Subdivide the sub-modules until we come to leaf cells
 - Leaf cell: circuit components that cannot further be divided (e.g., logic gates, primitive cell library elements)





DESIGN METHODOLOGY: BOTTOM-UP

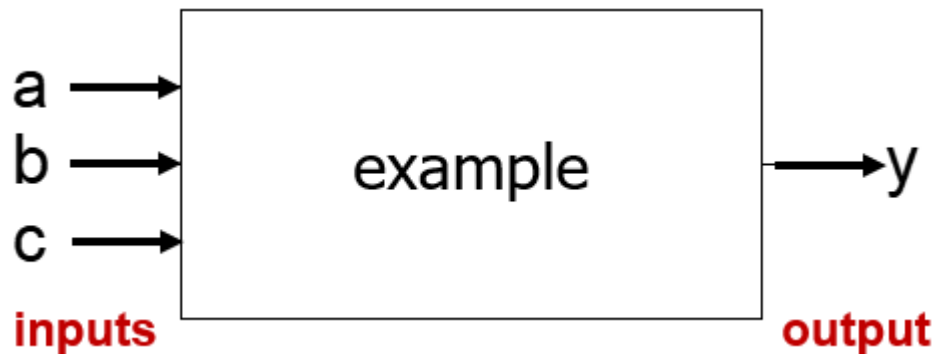
- ❖ We first identify the building blocks that are available to us
- ❖ Build bigger modules, using these building blocks
- ❖ These modules are then used for higher-level modules until we build the top-level module in the design





MODULE IN VERILOG

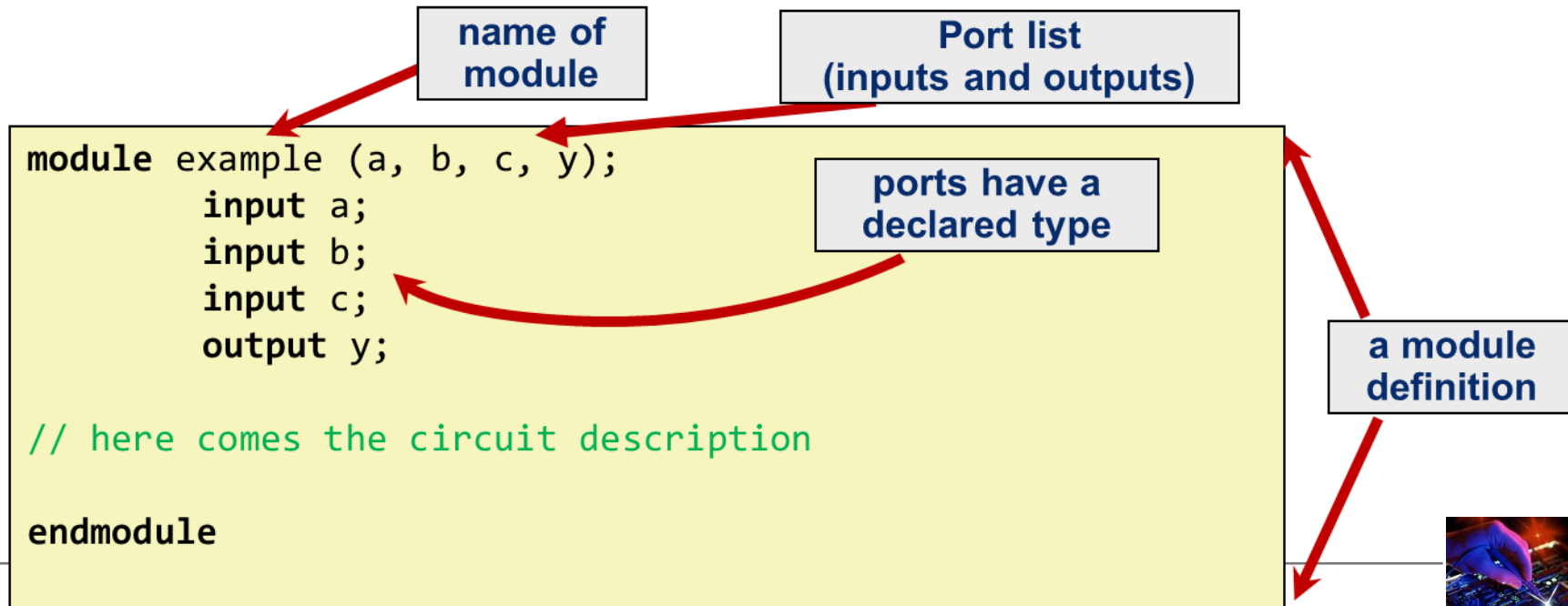
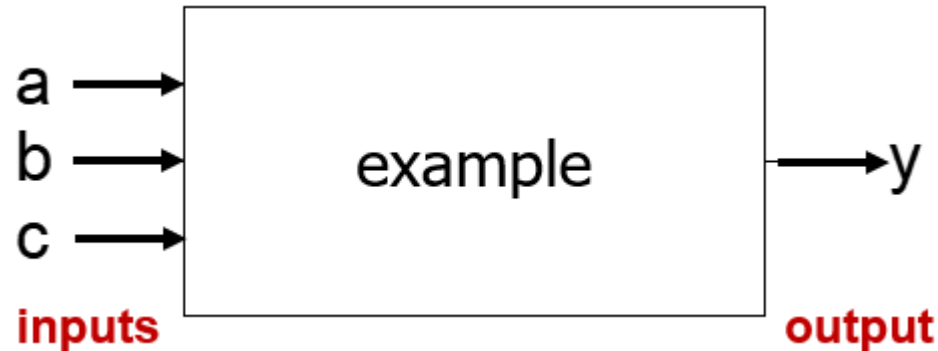
- ❖ A module is the main building block in Verilog
- ❖ We first need to define:
 - Name of the module
 - Directions of its ports (e.g., input, output)
 - Names of its ports
- ❖ Then:
 - Describe the functionality of the module





MODULE IN VERILOG

- ❖ A module is the main building block in Verilog





MODULE IN VERILOG

- ❖ A module is the main building block in Verilog
- ❖ The following two codes are functionally identical

```
module test ( a, b, y );  
    input a;  
    input b;  
    output y;  
  
endmodule
```

```
module test ( input a,  
              input b,  
              output y );  
  
endmodule
```

port name and direction declaration
can be combined





MULTI-BIT INPUT/OUTPUT

❖ You can also define multi-bit Input/Output (Bus)

- [range_end : range_start]
- Number of bits: range_end – range_start + 1

❖ Example:

```
input  [31:0] a;    // a[31], a[30] .. a[0]
output [15:8] b1;   // b1[15], b1[14] .. b1[8]
output [7:0]  b2;   // b2[7], b2[6] .. b2[0]
input                c;    // single signal
```

- ❖ a represents a 32-bit value, so we prefer to define it as: [31:0] a
- ❖ It is preferred over [0:31] a which resembles array definition
- ❖ It is good practice to be consistent with the representation of multi-bit signals, i.e., always [31:0] or always [0:31]





MANIPULATING BITS

- ❖ Bit Slicing
- ❖ Concatenation
- ❖ Duplication

```
// You can assign partial buses
wire [15:0] longbus;
wire [7:0] shortbus;
assign shortbus = longbus[12:5];

// Concatenating is by {}
assign y = {a[2],a[1],a[0],a[0]};

// Possible to define multiple copies
assign x = {a[0], a[0], a[0], a[0]}
assign y = { 4{a[0]} }
```





BASIC SYNTAX

- ❖ Verilog is case sensitive
 - **SomeName** and **somename** are not the same
- ❖ Names cannot start with numbers:
 - **2good** is not a valid name
- ❖ Whitespaces are ignored

```
// Single line comments start with a //  
  
/* Multiline comments  
   are defined like this */
```





HDL IMPLEMENTATION

❖ Two main types of HDL implementation

□ Structural (Gate-Level)

- The module body contains a gate-level description of the circuit
- Describe how modules are interconnected
- Each module contains other modules (instances)
- ... and interconnections between those modules
- Describes a hierarchy of modules defined as gates

□ Behavioral

- The module body contains a functional description of the circuit
- Contains logical and mathematical operators
- Level of abstraction is higher than gate-level
 - ◇ Many possible gate-level realizations of a behavioral description

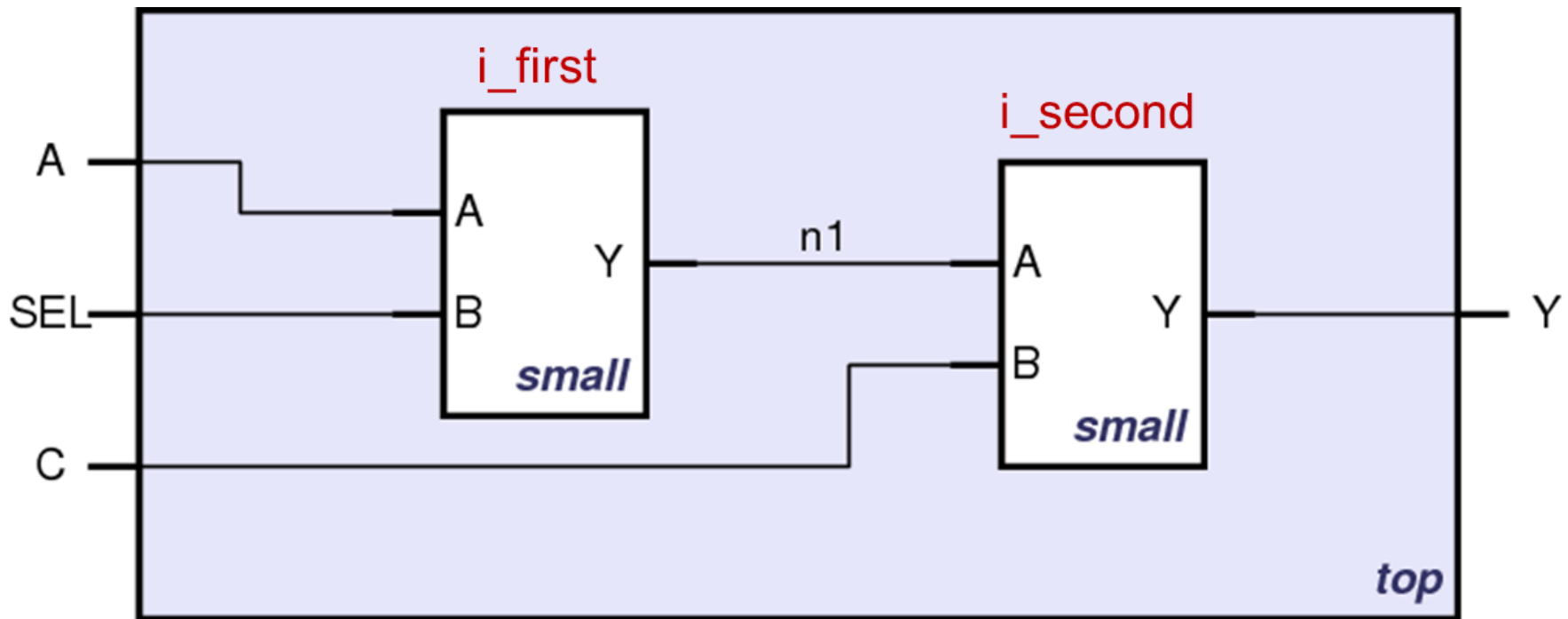
□ Many practical designs use a combination of both





STRUCTURAL (GATE-LEVEL) HDL

- ❖ Instantiating a module
- ❖ Schematic of module “top” that is built from two instances of module “small”

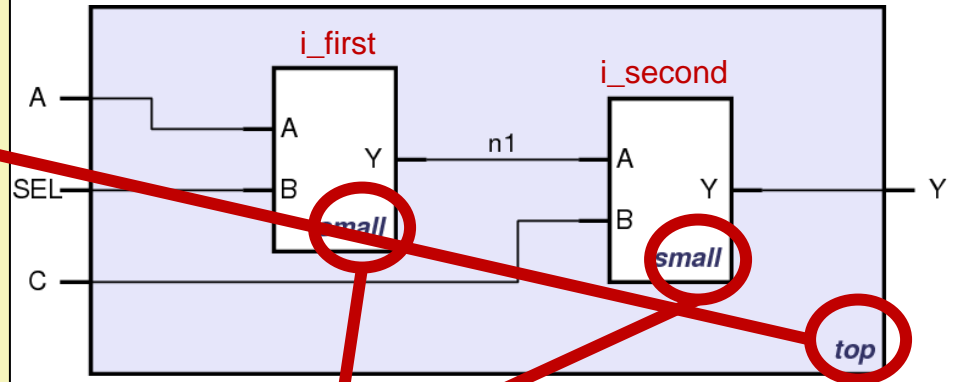


STRUCTURAL (GATE-LEVEL) HDL

❖ Module definitions in Verilog

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

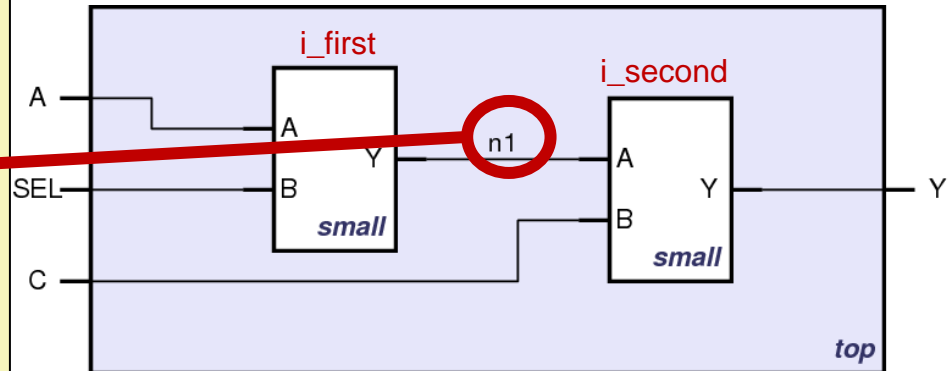


STRUCTURAL (GATE-LEVEL) HDL

❖ Defining wires (module interconnections)

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```





STRUCTURAL (GATE-LEVEL) HDL

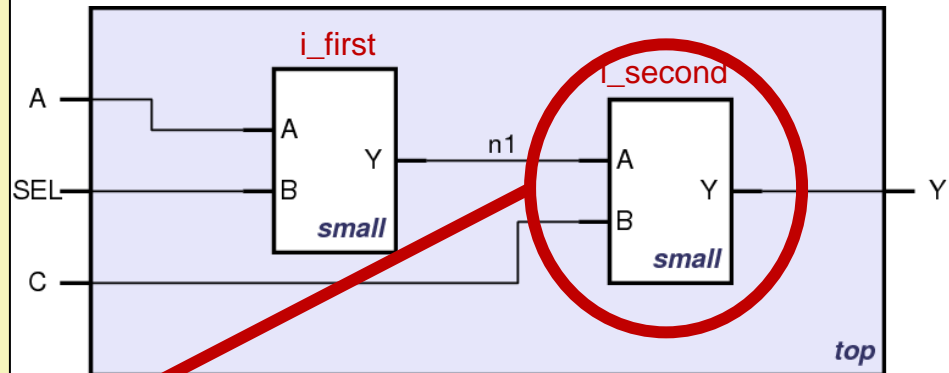
- ❖ The second instantiation of the “small” module

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
// instantiate small once  
small i_first ( .A(A),  
                .B(SEL),  
                .Y(n1) );
```

```
// instantiate small second time  
small i_second ( .A(n1),  
                 .B(C),  
                 .Y(Y) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small  
endmodule
```

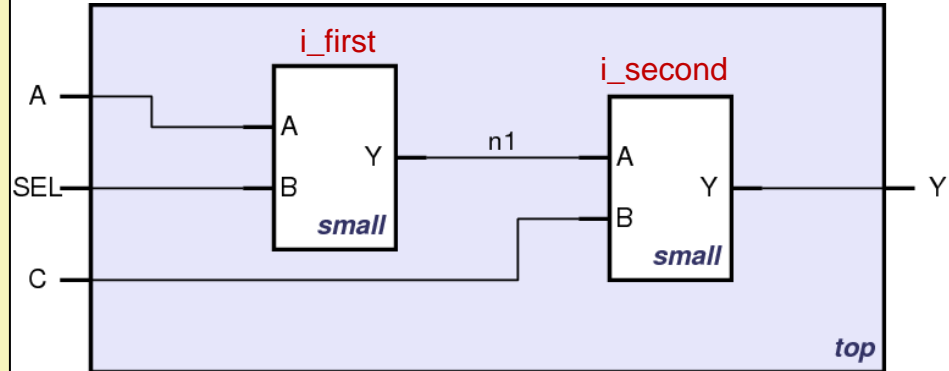




STRUCTURAL (GATE-LEVEL) HDL

❖ Short form of module instantiation

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;  
  
  // alternative short form  
  small i_first ( A, SEL, n1 );  
  
  /* In the short form above,  
     pin order very important */  
  
  // safer choice; any pin order  
  small i_second ( .B(C),  
                  .Y(Y),  
                  .A(n1) );  
  
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;  
  
  // description of small  
  
endmodule
```

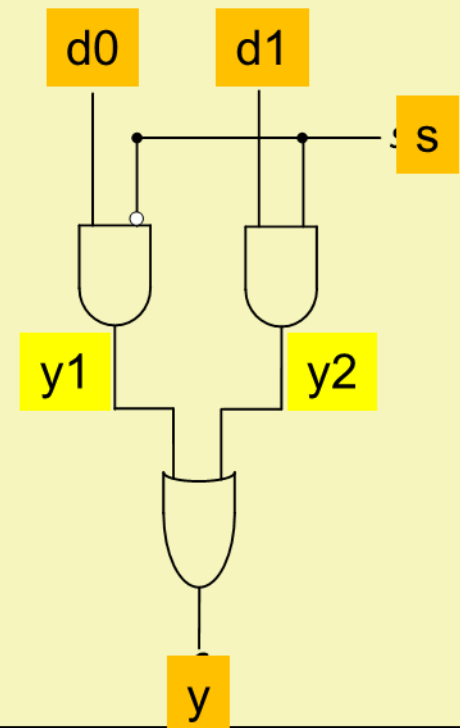




STRUCTURAL (GATE-LEVEL) HDL

- ❖ Verilog supports basic logic gates as predefined primitives
 - These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition

```
module mux2(input d0, d1,  
            input s,  
            output y);  
    wire ns, y1, y2;  
  
    not    g1 (ns, s);  
    and    g2 (y1, d0, ns);  
    and    g3 (y2, d1, s);  
    or     g4 (y, y1, y2);  
  
endmodule
```





BEHAVIORAL HDL

❖ Defining functionality

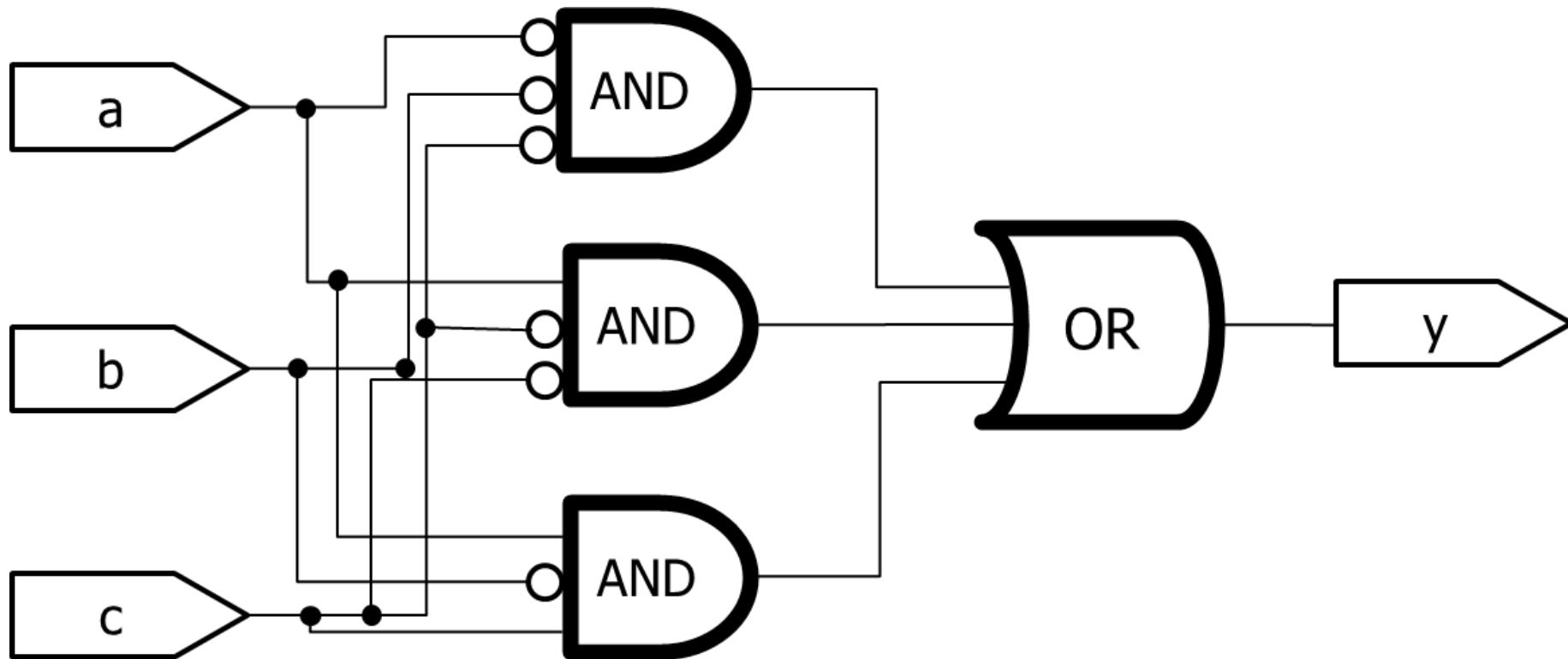
```
module example (a, b, c, y);  
    input a;  
    input b;  
    input c;  
    output y;  
  
    // here comes the circuit description  
    assign y = ~a & ~b & ~c |  
               a & ~b & ~c |  
               a & ~b & c;  
  
endmodule
```





BEHAVIORAL HDL

- ❖ Schematic view
- ❖ A behavioral implementation still models a hardware circuit





BEHAVIORAL HDL

❖ Bitwise operators in Behavioral Verilog

```
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 buses */

    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

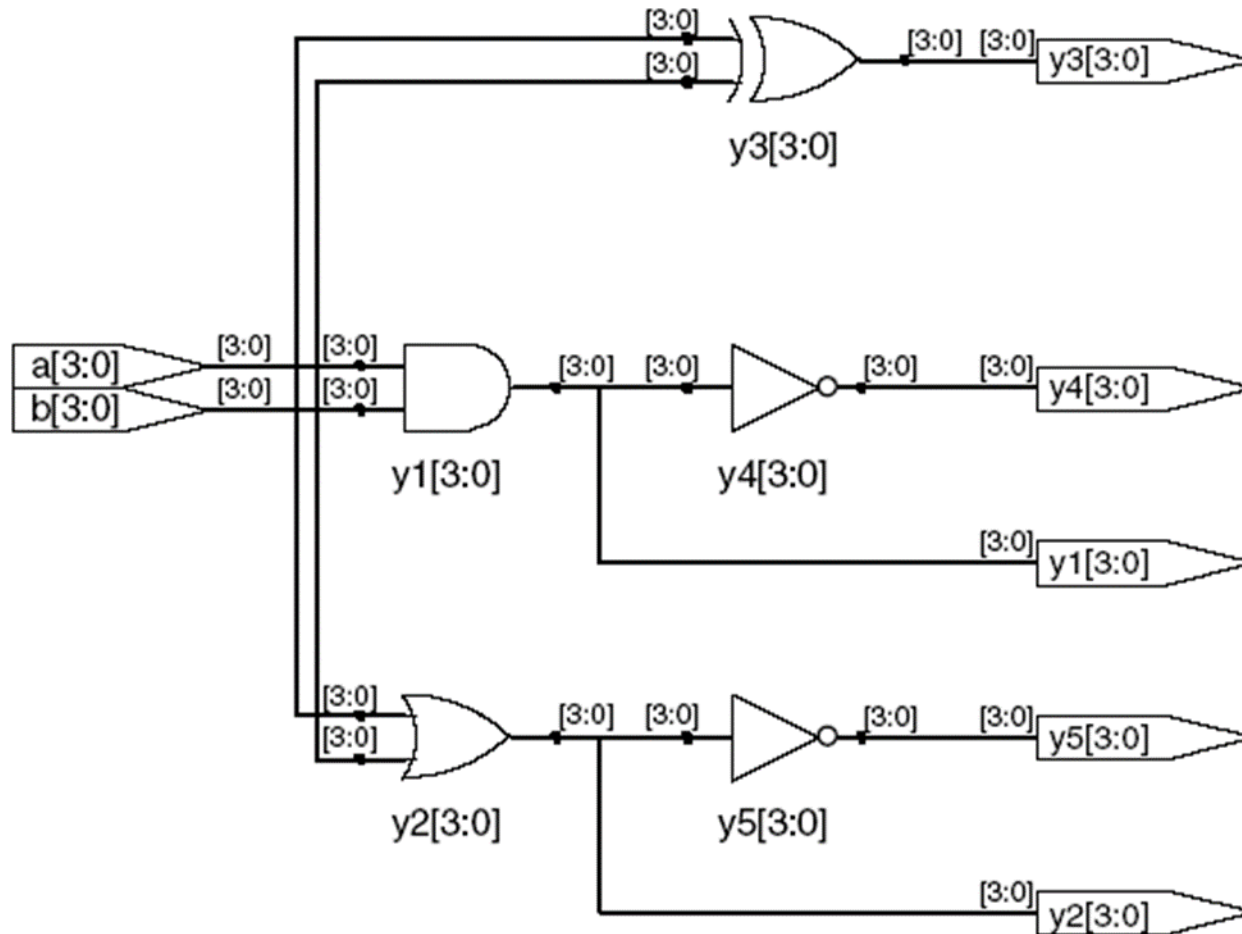
endmodule
```





BEHAVIORAL HDL

❖ Schematic view





BEHAVIORAL HDL

❖ Reduction operators

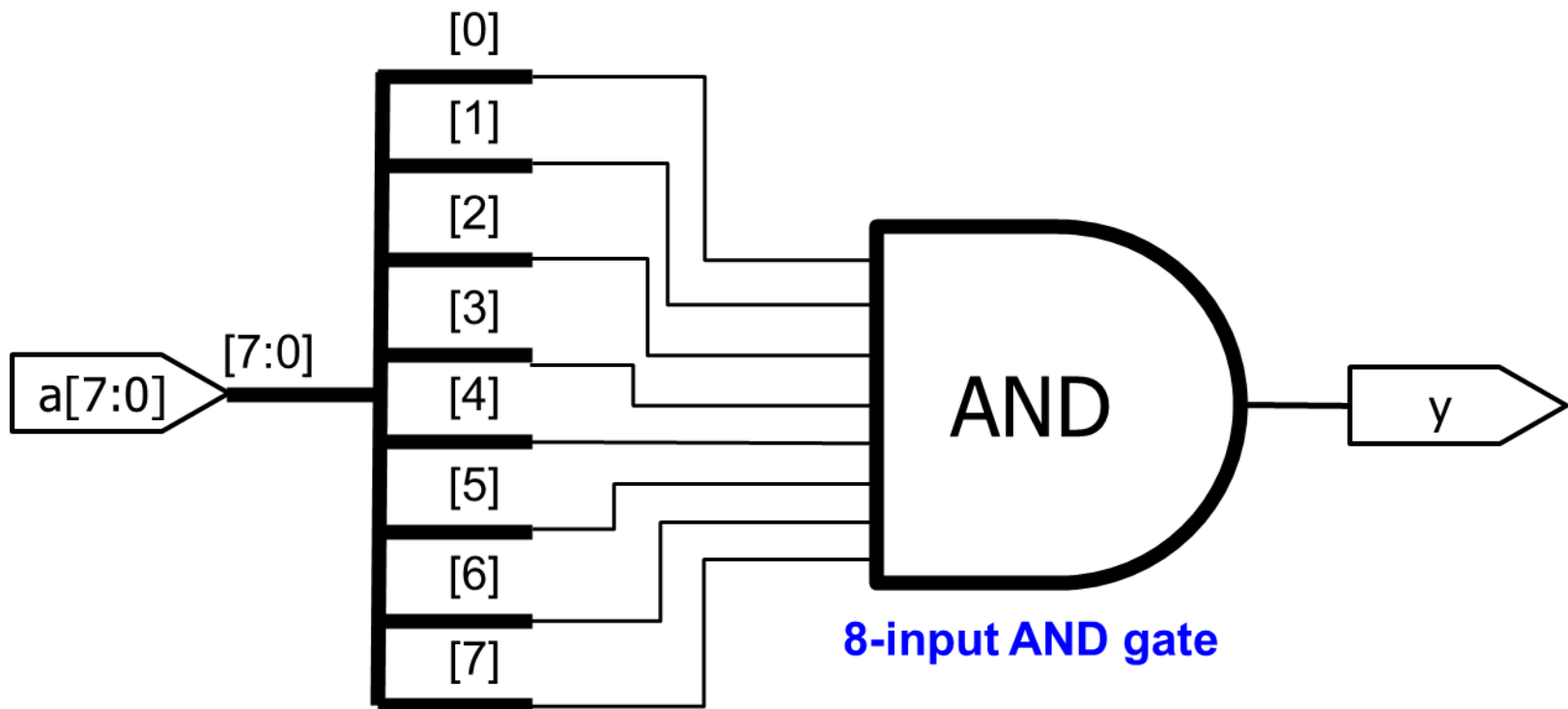
```
module and8(input  [7:0] a,  
            output  y);  
  
    assign y = &a;  
  
    // &a is much easier to write than  
    // assign y = a[7] & a[6] & a[5] & a[4] &  
    //           a[3] & a[2] & a[1] & a[0];  
  
endmodule
```





BEHAVIORAL HDL

❖ Schematic view





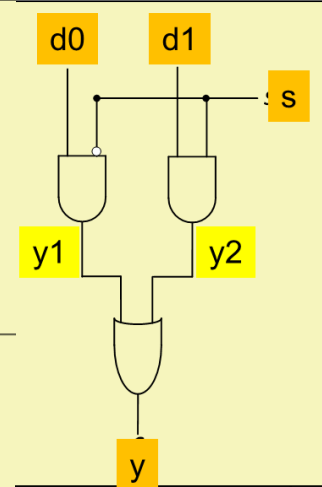
BEHAVIORAL HDL

❖ Conditional assignment

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

Structural HDL

```
module mux2(input d0, d1,  
            input s,  
            output y);  
    wire ns, y1, y2;  
  
    not g1 (ns, s);  
    and g2 (y1, d0, ns);  
    and g3 (y2, d1, s);  
    or  g4 (y, y1, y2);  
  
endmodule
```



❖ ? : is also called a ternary operator as it operates on three inputs:

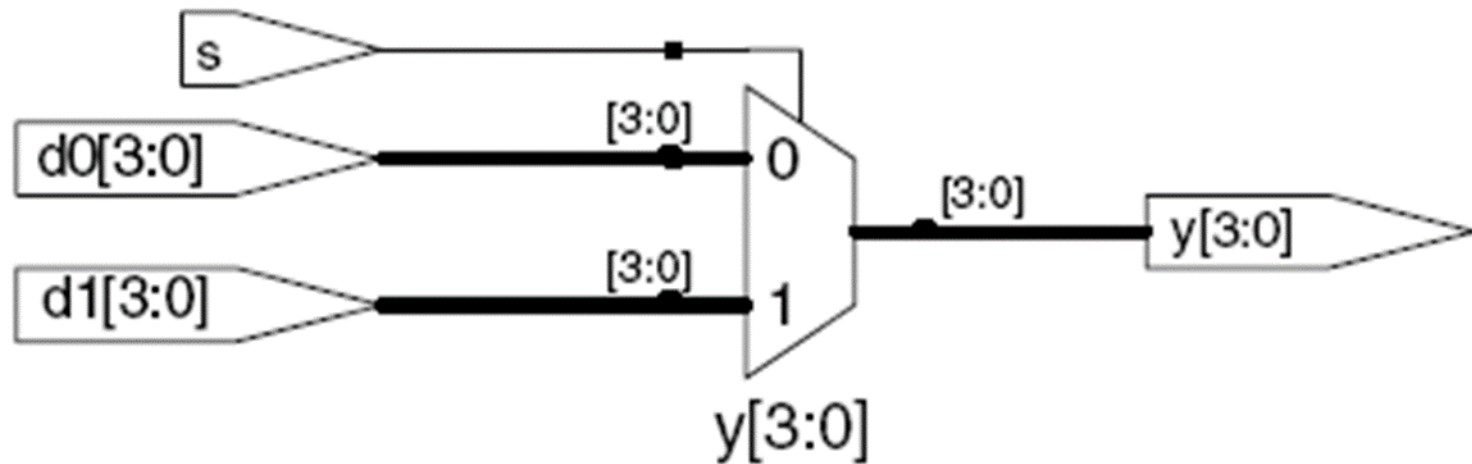
- s
- d1
- d0





BEHAVIORAL HDL

❖ Schematic view





BEHAVIORAL HDL

❖ More complex conditional assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

    assign y = s[1] ? ( s[0] ? d3 : d2)
               : ( s[0] ? d1 : d0);

    // if (s1) then
    //     if (s0) then y=d3 else y=d2
    // else
    //     if (s0) then y=d1 else y=d0

endmodule
```





BEHAVIORAL HDL

- ❖ Even more complex conditional assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

    assign y = (s == 2'b11) ? d3 :
               (s == 2'b10) ? d2 :
               (s == 2'b01) ? d1 :
               d0;

    // if      (s = "11" ) then y= d3
    // else if (s = "10" ) then y= d2
    // else if (s = "01" ) then y= d1
    // else                      y= d0

endmodule
```





OPERATORS IN VERILOG

❖ Precedence of operations in Verilog

Highest

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

Lowest





NUMBERS IN VERILOG

N'Bxx

8'b0000_0001

❖ (N) Number of bits

- Expresses how many bits will be used to store the value

❖ (B) Base

- Can be b (binary), h (hexadecimal), d (decimal), o (octal)

❖ (xx) Number


- The value expressed in base
- Can also have X (invalid) and Z (floating), as values
- Underscore _ can be used to improve readability





NUMBERS IN VERILOG

❖ Number representation in Verilog

Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1010 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001  32 bits (default)	12'h0	0000 0000 0000

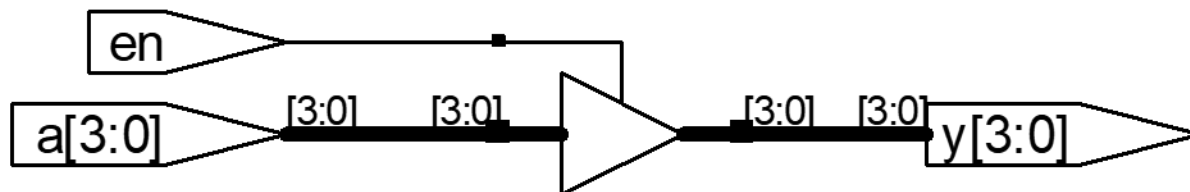




FLOATING SIGNAL (Z)

- ❖ Floating signal: Signal that is not driven by any circuit
 - Open circuit, floating wire
- ❖ Also known as: high impedance, hi-Z, tri-stated signals

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





TRUTH TABLE WITH Z AND X

❖ Truth table for AND gate with Z and X

AND		A			
		0	1	Z	X
B	0	0	0	0	0
	1	0	1	X	X
	Z	0	X	X	X
	X	0	X	X	X





HDL CONVERSION

❖ Synthesis (i.e., Hardware Synthesis)

- ❑ Modern tools are able to map synthesizable HDL code into low-level cell libraries -> netlist describing gates and wires
- ❑ They can perform many optimizations
- ❑ ... however they can not guarantee that a solution is optimal
 - Mainly due to computationally expensive placement and routing algorithms
 - Need to describe your circuit in HDL in a nice-to-synthesize way
- ❑ Most common way of Digital Design these days

❖ Simulation

- ❑ Allows the behavior of the circuit to be verified without actually manufacturing the circuit
- ❑ Simulators can work on structural or behavioral HDL
- ❑ Simulation is essential for functional and timing verification





NOTE

❖ A note on hardware synthesis

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as 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.





COMPARING TWO NUMBERS

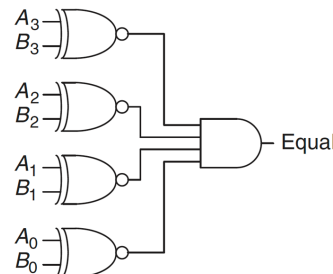
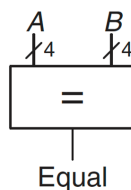
- ❖ Defining your own gates as new modules
- ❖ We will use our gates to show different ways of implementing a 4-bit comparator (equality checker)

A 2-input XNOR gate

```
module MyXnor (input A, B,  
               output Z);  
  
    assign Z = ~(A ^ B); //not XOR  
  
endmodule
```

A 2-input AND gate

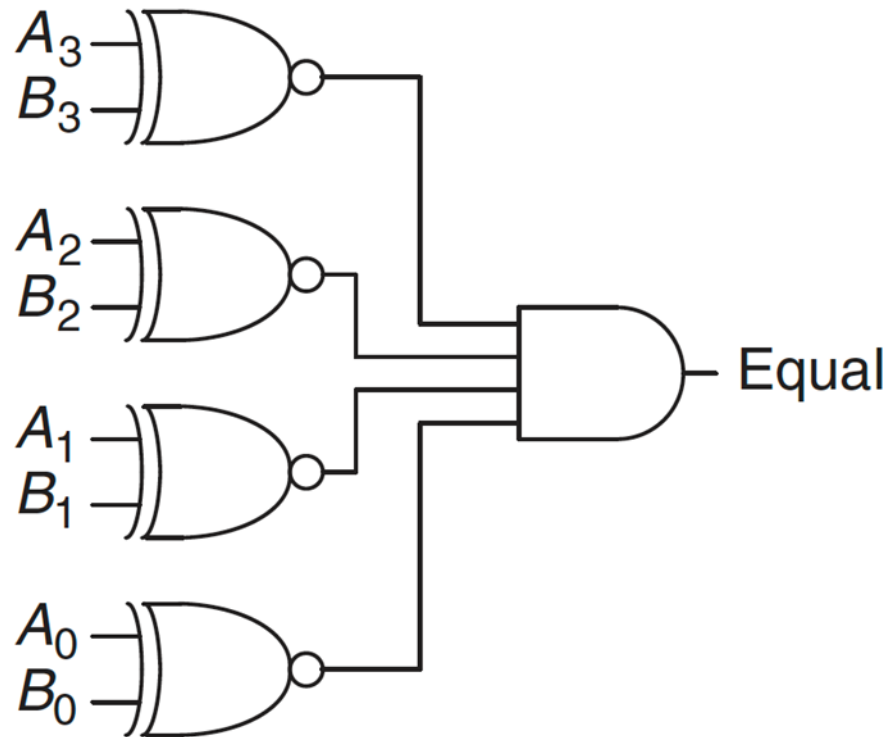
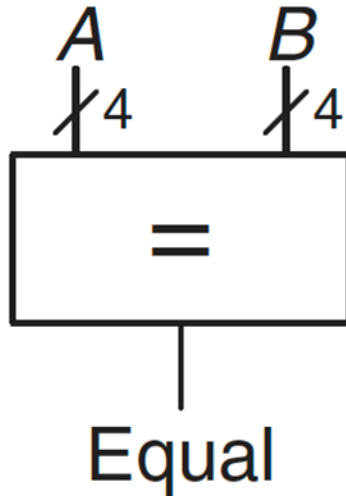
```
module MyAnd (input A, B,  
              output Z);  
  
    assign Z = A & B;    // AND  
  
endmodule
```





COMPARING TWO NUMBERS

❖ Example: 4-bit Comparator





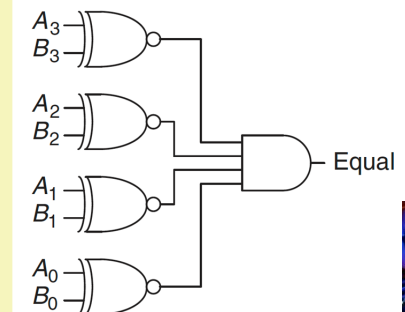
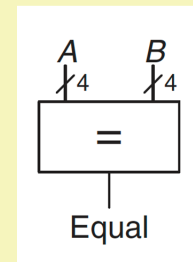
COMPARING TWO NUMBERS

❖ Gate-level implementation

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,  
                output eq);  
    wire c0, c1, c2, c3, c01, c23;
```

```
MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR  
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR  
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR  
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR  
MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND  
MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND  
MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND
```

```
endmodule
```



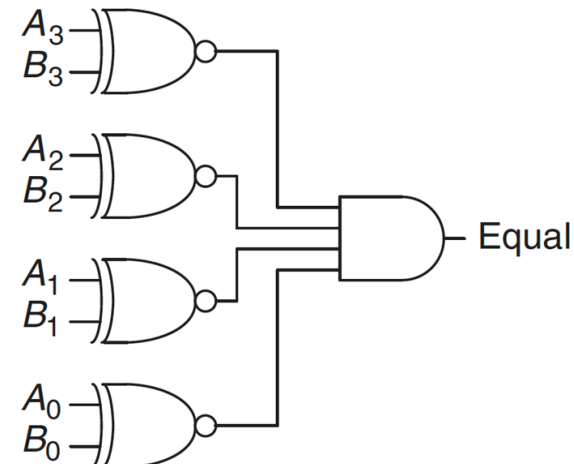
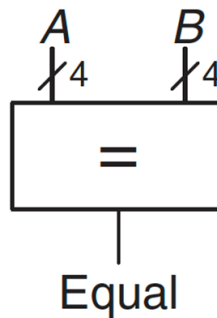
COMPARING TWO NUMBERS

❖ Using logical operators

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                 output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    assign c01 = c0 & c1;
    assign c23 = c2 & c3;
    assign eq   = c01 & c23;

endmodule
```



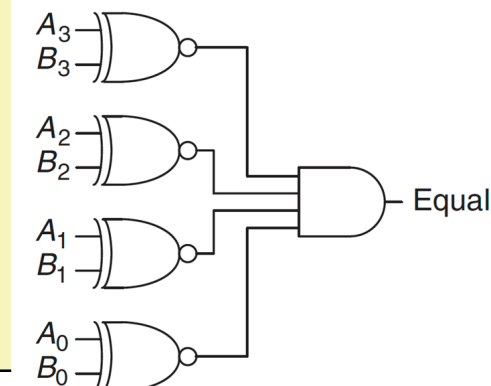
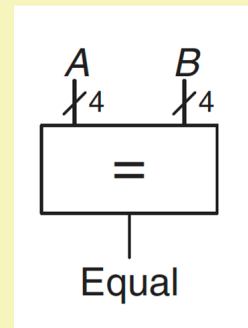
COMPARING TWO NUMBERS

❖ Eliminating intermediate signals

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                 output eq);
    wire c0, c1, c2, c3;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    // assign c01 = c0 & c1;
    // assign c23 = c2 & c3;
    // assign eq  = c01 & c23;
    assign eq  = c0 & c1 & c2 & c3;

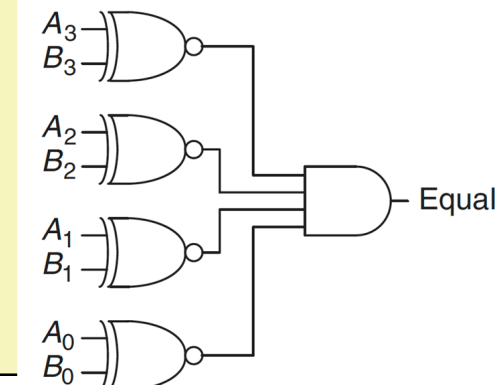
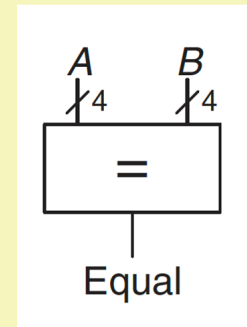
endmodule
```



COMPARING TWO NUMBERS

❖ Multi-bit signals (Bus)

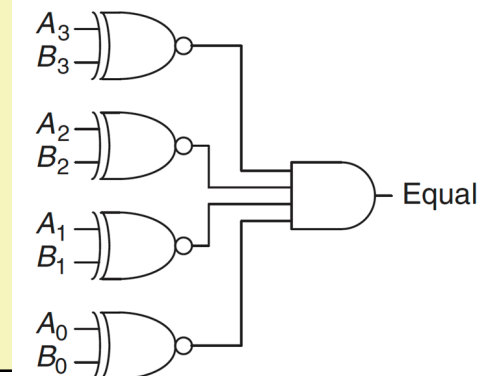
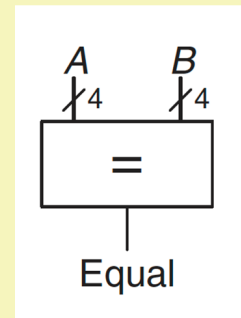
```
module compare (input [3:0] a, input [3:0] b,  
                output eq);  
    wire [3:0] c; // bus definition  
  
    MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) ); // XNOR  
    MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) ); // XNOR  
    MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) ); // XNOR  
    MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) ); // XNOR  
  
    assign eq = &c; // short format  
  
endmodule
```



COMPARING TWO NUMBERS

❖ Bitwise operators

```
module compare (input [3:0] a, input [3:0] b,  
                output eq);  
    wire [3:0] c; // bus definition  
  
    // MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) );  
    // MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) );  
    // MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) );  
    // MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) );  
  
    assign c = ~(a ^ b); // XNOR  
  
    assign eq = &c; // short format  
  
endmodule
```



COMPARING TWO NUMBERS

❖ Highest abstraction level

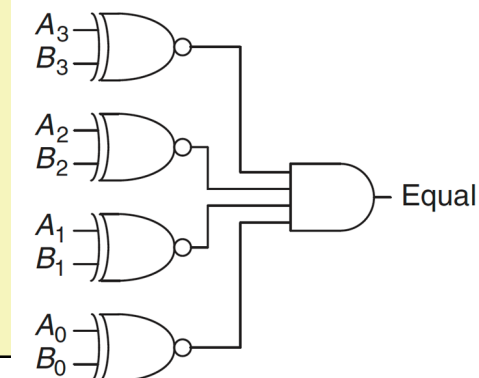
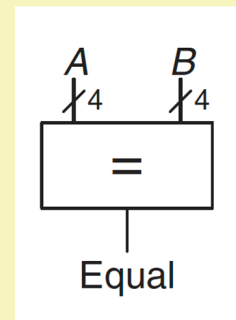
```
module compare (input [3:0] a, input [3:0] b,  
                output eq);
```

```
// assign c = ~(a ^ b); // XNOR
```

```
// assign eq = &c; // short format
```

```
assign eq = (a == b) ? 1 : 0; // really short
```

```
endmodule
```





REUSABLE VERILOG CODE

- ❖ We have a module that can compare two 4-bit numbers
- ❖ What if in the overall design we need to compare:
 - ☐ 5-bit numbers?
 - ☐ 6-bit numbers?
 - ☐ ...
 - ☐ N-bit numbers?
- ❖ Writing code for each case looks tedious
- ❖ What could be a better way?





PARAMETERIZED MODULES

- ❖ In Verilog, we can define module parameters

```
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
```

- ❖ We can set the parameters to different values when instantiating the module





PARAMETERIZED MODULES

❖ Instantiating parameterized modules

```
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
```

```
// If the parameter is not given, the default (8) is assumed
mux2 i_mux (d0, d1, s, out);

// The same module with 12-bit bus width:
mux2 #(12) i_mux_b (d0, d1, s, out);

// A more verbose version:
mux2 #(.width(12)) i_mux_b (.d0(d0), .d1(d1),
                           .s(s), .out(out));
```





SEQUENTIAL LOGIC IN VERILOG

- ❖ Define blocks that have memory
 - Flip-Flops, Latches, Finite State Machines

- ❖ Sequential Logic state transition is triggered by a “CLOCK” signal
 - Latches are sensitive to level of the signal
 - Flip-flops are sensitive to the transitioning of signal

- ❖ Combinational HDL constructs are not sufficient to express sequential logic
 - We need new constructs:
 - always
 - posedge/negedge





THE “ALWAYS” BLOCK

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

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

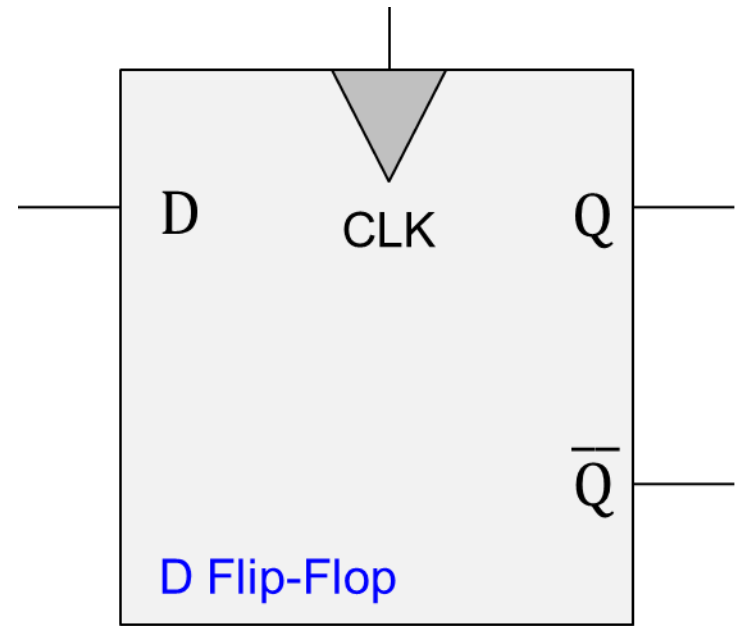




THE “ALWAYS” BLOCK

❖ The D flip-flop

- State change on clock edge,
- Data available for full cycle



- At the rising edge of clock (clock going from 0->1), Q gets assigned D
- At all other times, Q is unchanged





THE “ALWAYS” BLOCK

❖ The D flip-flop

- posedge defines a rising edge (transition from 0 to 1)
- Statement executed when the clk signal rises (posedge of clk)
- Once the clk signal rises: the value of d is copied to q

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

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

endmodule
```





THE “ALWAYS” BLOCK

❖ The D flip-flop

- assign statement is not used within an always block
- <= describes a non-blocking assignment
- We will see the difference between blocking assignment and non-blocking assignment soon

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

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

endmodule
```



THE “ALWAYS” BLOCK

❖ The D flip-flop

- Assigned variables need to be declared as reg
- The name reg does not necessarily mean that the value is a register (It could be, but it does not have to be)

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

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

endmodule
```





RESET

- ❖ Reset signals are used to initialize the hardware to a known state
 - Usually activated at system start (on power up)

- ❖ Asynchronous Reset
 - The reset signal is sampled independent of the clock
 - Reset gets the highest priority
 - Sensitive to glitches, may have metastability issues

- ❖ Synchronous Reset
 - The reset signal is sampled with respect to the clock
 - The reset should be active long enough to get sampled at the clock edge
 - Results in completely synchronous circuit





ASYNCHRONOUS RESET

- ❖ D Flip-Flop with Asynchronous Reset
- ❖ In this example: two events can trigger the process:
 - A rising edge on clk
 - A falling edge on reset

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

  always @ (posedge clk, negedge reset)
  begin
    if (reset == 0) q <= 0;    // when reset
    else           q <= d;    // when clk
  end
endmodule
```



ASYNCHRONOUS RESET

- ❖ D Flip-Flop with Asynchronous Reset
- ❖ For longer statements, a begin-end pair can be used
 - To improve readability
 - In this example, it was not necessary, but it is a good idea

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

    always @ (posedge clk, negedge reset)
        begin
            if (reset == 0) q <= 0;    // when reset
            else            q <= d;    // when clk
        end
endmodule
```



ASYNCHRONOUS RESET

- ❖ D Flip-Flop with Asynchronous Reset
- ❖ First reset is checked: if reset is 0, q is set to 0.
 - This is an asynchronous reset as the reset can happen independently of the clock (on the negative edge of reset signal)
- ❖ If there is no reset, then regular assignment takes effect

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

  always @ (posedge clk, negedge reset)
  begin
    if (reset == 0) q <= 0; // when reset
    else           q <= d;  // when clk
  end
endmodule
```





SYNCHRONOUS RESET

- ❖ D Flip-Flop with Synchronous Reset
- ❖ The process is sensitive to only clock
 - Reset happens only when the clock rises
 - This is a synchronous reset

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

    always @ (posedge clk)
    begin
        if (reset == '0') q <= 0;    // when reset
        else               q <= d;    // when clk
    end
endmodule
```





ENABLE AND RESET

- ❖ D Flip-Flop with Enable and Reset
- ❖ A flip-flop with enable and reset
 - Note that the en signal is not in the sensitivity list
- ❖ q gets d only when clk is rising and en is 1

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

  always @ (posedge clk, negedge reset)
  begin
    if (reset == '0') q <= 0;    // when reset
    else if (en)      q <= d;    // when en AND clk
  end
endmodule
```





LATCH

❖ D latch

```
module latch (input          clk,  
              input          [3:0] d,  
              output reg [3:0] q);  
  
  always @ (clk, d)  
    if (clk) q <= d;      // latch is transparent when  
                          // clock is 1  
  
endmodule
```





BASICS OF “ALWAYS” BLOCK

- ❖ You can have as many always blocks as needed
- ❖ Assignment to the same signal in different always blocks is not allowed

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

    wire [3:0] normal;           // standard wire
    reg  [3:0] special;          // assigned in always

    always @ (posedge clk)
        special <= d;            // first FF array

    assign normal = ~special;    // simple assignment

    always @ (posedge clk)
        q <= normal;            // second FF array
endmodule
```





BASICS OF “ALWAYS” BLOCK

❖ Why Does an always Block Remember?

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

    always @ (posedge clk)
        begin
            q <= d;    // when clk rises copy d to q
        end
endmodule
```

- This statement describes what happens to signal q
- ... but what happens when the clock is not rising?
 - The value of q is preserved (remembered)





BASICS OF “ALWAYS” BLOCK

- ❖ An always block does not remember

```
module comb (input          inv,
              input    [3:0] data,
              output reg [3:0] result);

    always @ (inv, data)          // trigger with inv, data
        if (inv) result <= ~data; // result is inverted data
        else    result <= data;  // result is data

endmodule
```

- This statement describes what happens to signal result
 - When inv is 1, result is ~data
 - When inv is not 1, result is data
- The circuit is combinational (no memory)
 - result is assigned a value whenever an input value changes & in all cases of the if .. else block





ALWAYS BLOCKS FOR COMBINATIONAL CIRCUITS

- ❖ An always block defines combinational logic if:
 - All outputs are always (continuously) updated
 - All right-hand side signals are in the sensitivity list
 - You can use always @* for short
 - All left-hand side signals get assigned in every possible condition of if .. else and case blocks

- ❖ It is easy to make mistakes and unintentionally describe memorizing elements (latches)
 - Vivado will most likely warn you. Make sure you check the warning messages

- ❖ Always blocks allow powerful combinational logic statements
 - if .. else
 - case





SEQUENTIAL OR COMBINATIONAL

❖ Sequential or combinational?

```
wire enable, data;
reg out_a, out_b;

always @ (*) begin
    out_a = 1'b0;
    if(enable) begin
        out_a = data;
        out_b = data;
    end
end
```

No assignment for ~enable

```
wire enable, data;
reg out_a, out_b;

always @ (data) begin
    out_a = 1'b0;
    out_b = 1'b0;
    if(enable) begin
        out_a = data;
        out_b = data;
    end
end
```

Not in the sensitivity list





ALWAYS BLOCKS FOR COMBINATIONAL CIRCUITS

- ❖ The always block is not always practical

```
reg [31:0] result;  
wire [31:0] a, b, comb;  
wire      sel,  
  
always @ (a, b, sel)    // trigger with a, b, sel  
    if (sel) result <= a; // result is a  
    else      result <= b; // result is b  
  
assign comb = sel ? a : b;
```

- Both statements describe the same multiplexer
- In this case, the always block is more work





ALWAYS BLOCKS FOR COMBINATIONAL CIRCUITS

- ❖ The always block is not always practical
- ❖ always block for case statements

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

    always @ ( * )                // * is short for all signals
    case (data)                   // case statement
        4'd0: segments = 7'b111_1110; // when data is 0
        4'd1: segments = 7'b011_0000; // when data is 1
        4'd2: segments = 7'b110_1101;
        4'd3: segments = 7'b111_1001;
        4'd4: segments = 7'b011_0011;
        4'd5: segments = 7'b101_1011;
        // etc etc
        default: segments = 7'b000_0000; // required
    endcase

endmodule
```





NON-BLOCKING AND BLOCKING ASSIGNMENTS

❖ Non-blocking and blocking assignments

Non-blocking (\leq)

```
always @ (a)
begin
    a <= 2'b01;
    b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- All assignments are made at the end of the block
- All assignments are made in parallel, process flow is not-blocked

Blocking ($=$)

```
always @ (a)
begin
    a = 2'b01;
// a is 2'b01
    b = a;
// b is now 2'b01 as well
end
```

- Each assignment is made immediately
- Process waits until the first assignment is complete, it **blocks** progress
- Similar to sequential programs





NON-BLOCKING AND BLOCKING ASSIGNMENTS

- ❖ Why use non-blocking or blocking assignments
 - Non-blocking statements allow operating on “old” values
 - Enable easy sequential logic descriptions
 - Blocking statements allow a sequence of operations
 - Allow operating on immediately updated values
 - More like a “software” programming language
 - If the sensitivity list is correct, a block with non-blocking statements will eventually evaluate to the same result as the same block with blocking statements
 - This may require some additional iterations





NON-BLOCKING AND BLOCKING ASSIGNMENTS

❖ Blocking assignment

- Assume all inputs are initially '0'

```
always @ ( * )  
begin  
    p    = a ^ b ;           // p    = 0  
    g    = a & b ;           // g    = 0  
    s    = p ^ cin ;         // s    = 0  
    cout = g | (p & cin) ;    // cout = 0  
end
```

- If a changes to '1'
 - All values are updated in order





NON-BLOCKING AND BLOCKING ASSIGNMENTS

❖ Non-blocking assignment

- Assume all inputs are initially '0'

```
always @ ( * )  
begin  
    p    <= a ^ b ;           // p    = 0  
    g    <= a & b ;           // g    = 0  
    s    <= p ^ cin ;         // s    = 0  
    cout <= g | (p & cin) ;   // cout = 0  
end
```

- If a changes to '1'
 - All assignments are concurrent
 - When s is being assigned, p is still 0





NON-BLOCKING AND BLOCKING ASSIGNMENTS

❖ Non-blocking assignment

- After the first iteration, p has changed to '1' as well

```
always @ ( * )  
begin  
    p    <= a ^ b ;           // p    = 1  
    g    <= a & b ;           // g    = 0  
    s    <= p ^ cin ;         // s    = 0  
    cout <= g | (p & cin) ;   // cout = 0  
end
```

- Since there is a change in p, the process triggers again
- This time s is calculated with p=1





RULES FOR SIGNAL ASSIGNMENT

- ❖ Use always @(posedge clk) and non-blocking assignments (<=) to model synchronous sequential logic

```
always @ (posedge clk)
    q <= d; // non-blocking
```

- ❖ Use continuous assignments (assign) to model simple combinational logic

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





RULES FOR SIGNAL ASSIGNMENT

- ❖ Use always @ (*) and blocking assignments (=) to model more complicated combinational logic
- ❖ You cannot make assignments to the same signal in more than one always block or in a continuous assignment

~~**always @ (*)**
a = b;

always @ (*)
a = c;~~

~~**always @ (*)**
a = b;

assign a = c;~~

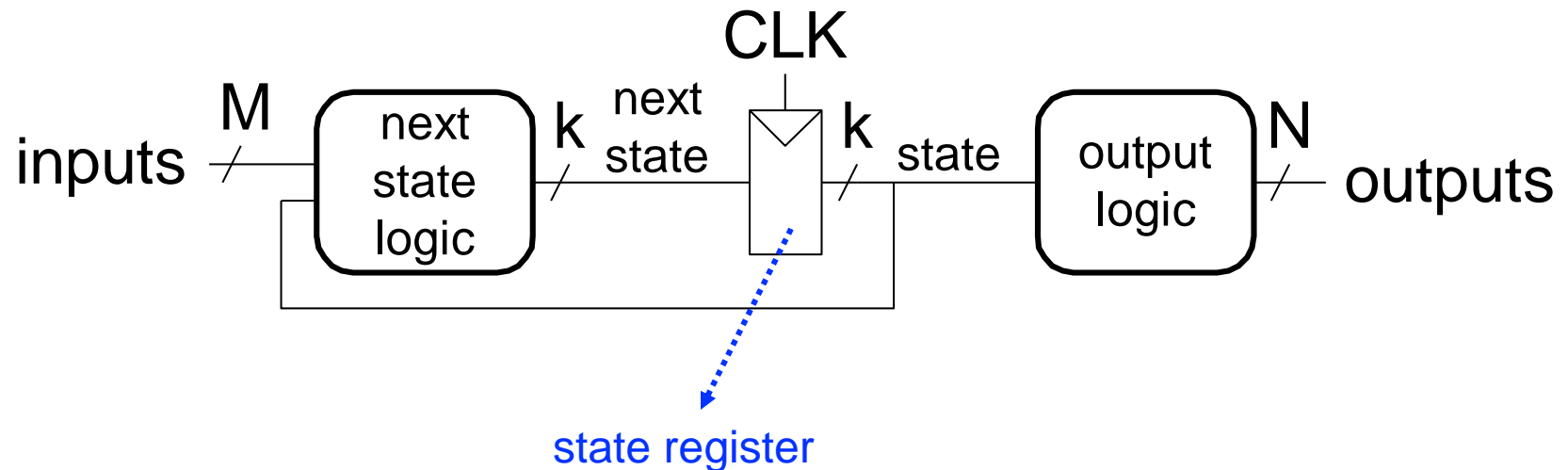




FSMS

❖ Each FSM consists of three separate parts:

- next state logic
- state register
- output logic



At the beginning of the clock cycle, next state is latched into the state register

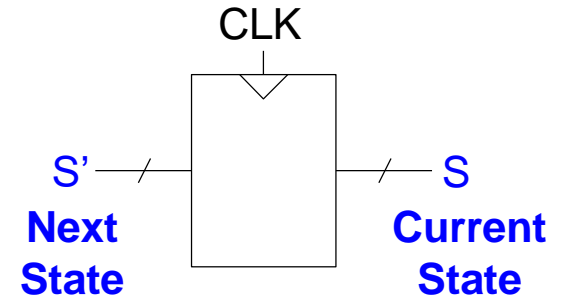




FSMS

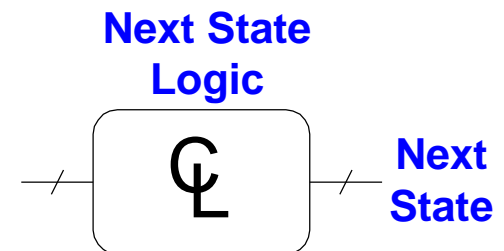
❖ Sequential Circuits

- ❑ State register(s)
- ❑ Store the current state and
 - Load the next state at the clock edge



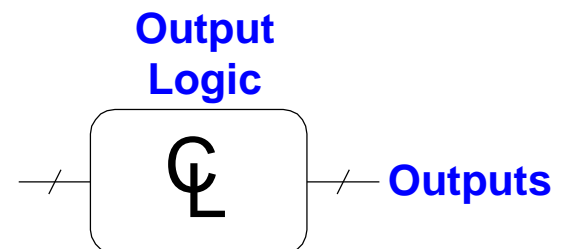
❖ Combinational Circuits

- ❑ Next state logic
 - Determines what the next state will be



❖ Output logic

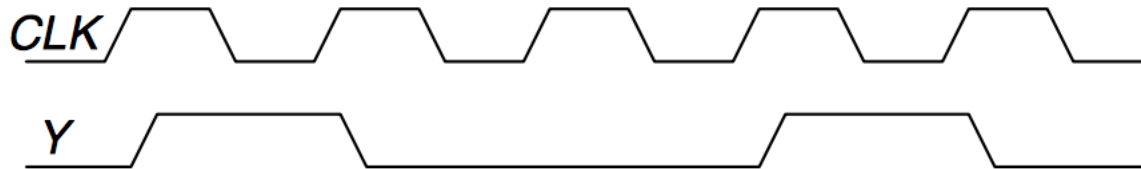
- ❑ Generates the outputs



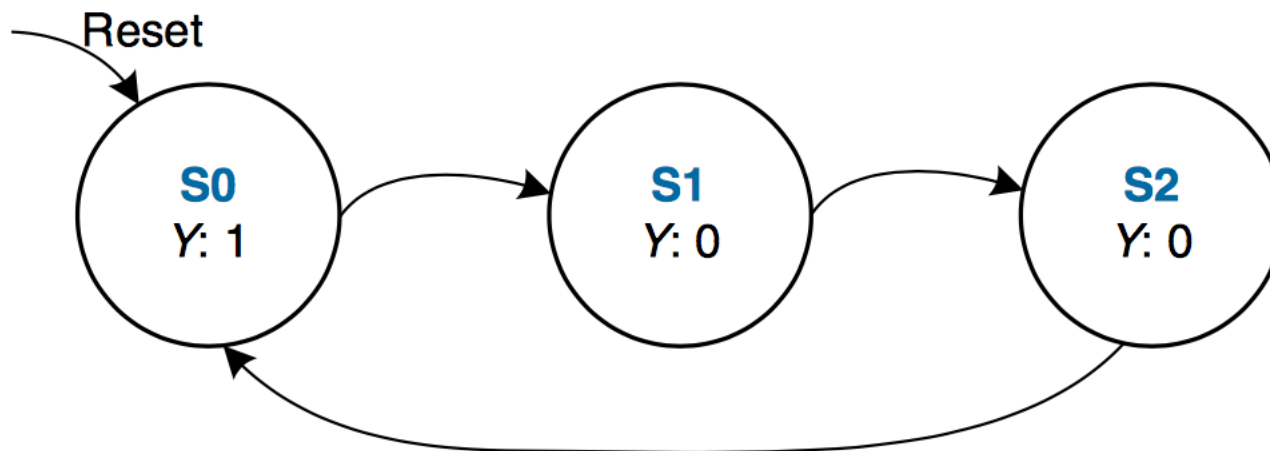


FSMS

- ❖ Example: Divide the clock frequency by 3



The output *Y* is HIGH for **one clock cycle out of every 3**. In other words, the output **divides the frequency of the clock by 3**.





FSMS

❖ Definitions

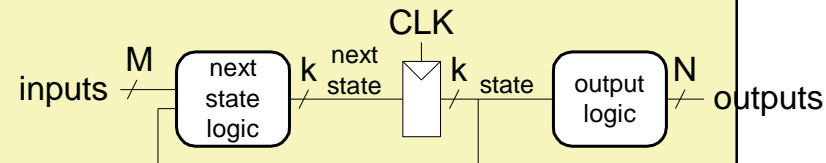
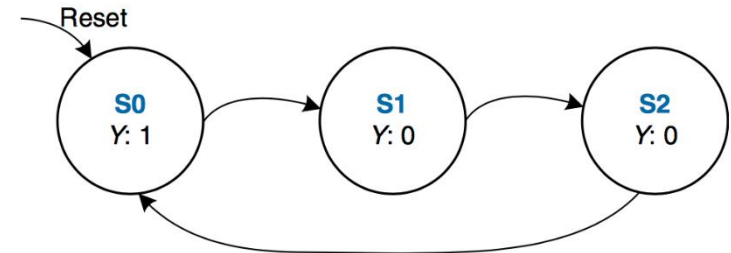
```
module divideby3FSM (input clk,  
                    input reset,  
                    output q);
```

```
    reg [1:0] state, nextstate;
```

```
    parameter S0 = 2'b00;
```

```
    parameter S1 = 2'b01;
```

```
    parameter S2 = 2'b10;
```



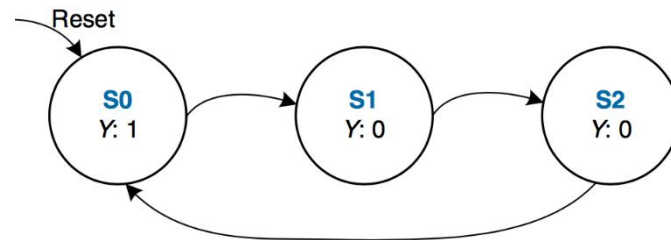
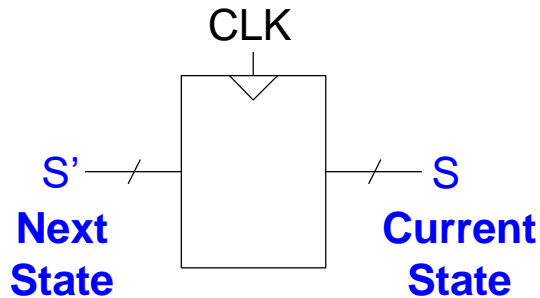
- We define state and nextstate as 2-bit reg
- The parameter descriptions are optional, it makes reading easier





FSMS

❖ State register



```
// state register
always @ (posedge clk, posedge reset)
  if (reset) state <= S0;
  else      state <= nextstate;
```

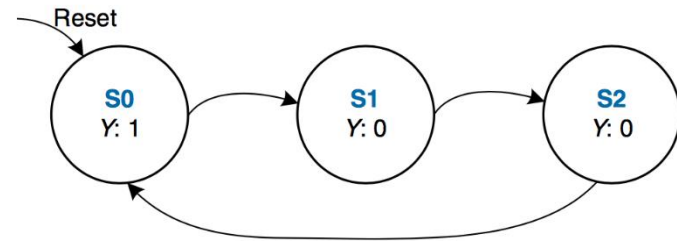
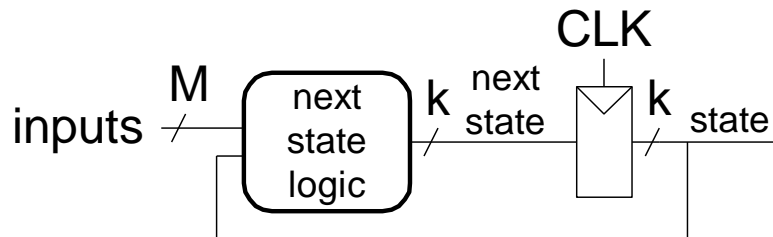
- ❑ This part defines the state register (memorizing process)
- ❑ Sensitive to only clk, reset
- ❑ In this example, reset is active when it is '1' (active-high)





FSMS

❖ Next state logic



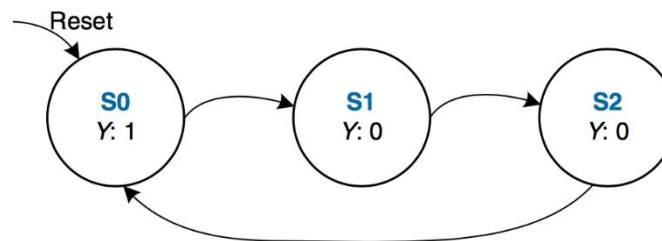
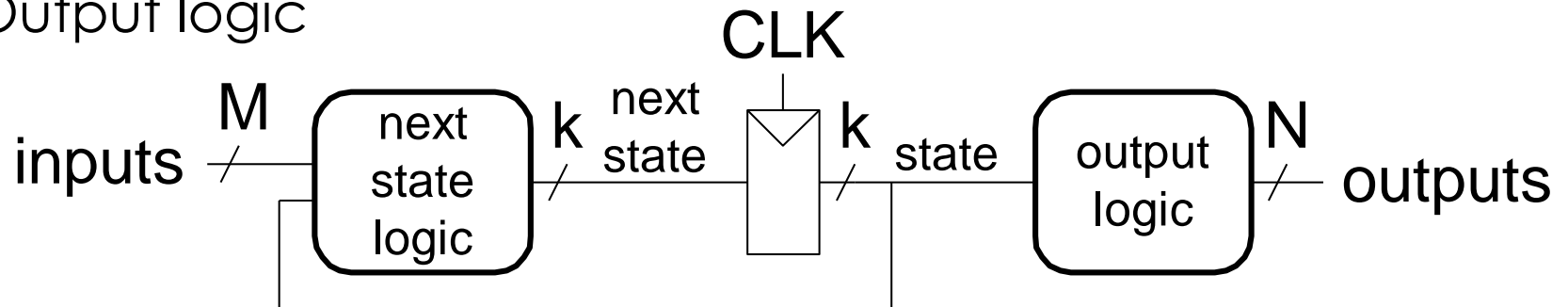
```
// next state logic
always @ (*)
    case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
    endcase
```





FSMS

❖ Output logic



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

- In this example, output depends only on state
 - Moore type FSM





FSMS

❖ Implementation

```
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;  
  
    always @ (posedge clk, posedge reset) // state register  
        if (reset) state <= S0;  
        else      state <= nextstate;  
  
    always @ (*)  
        case (state)  
            S0:      nextstate = S1;  
            S1:      nextstate = S2;  
            S2:      nextstate = S0;  
            default: nextstate = S0;  
        endcase  
    assign q = (state == S0);  
endmodule
```

// next state logic

// output logic

