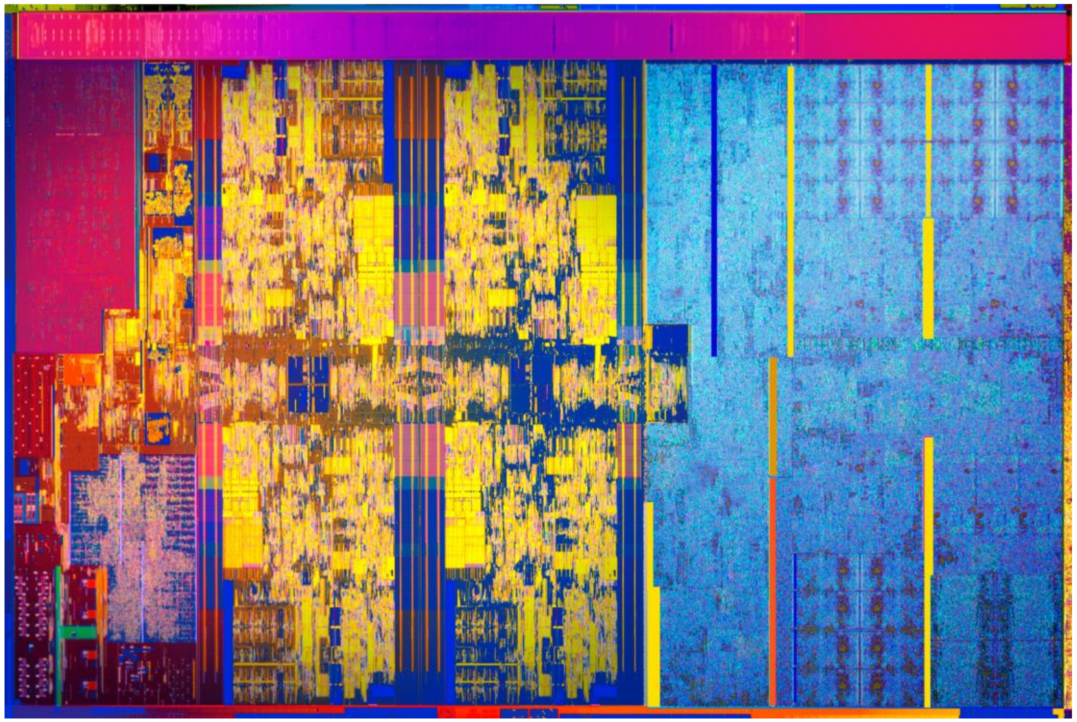
Hardware Description Languages and Verilog

陈刚
副教授
人工智能与无人系统研究所
计算机学院
中山大学

2017: Intel Kaby Lake

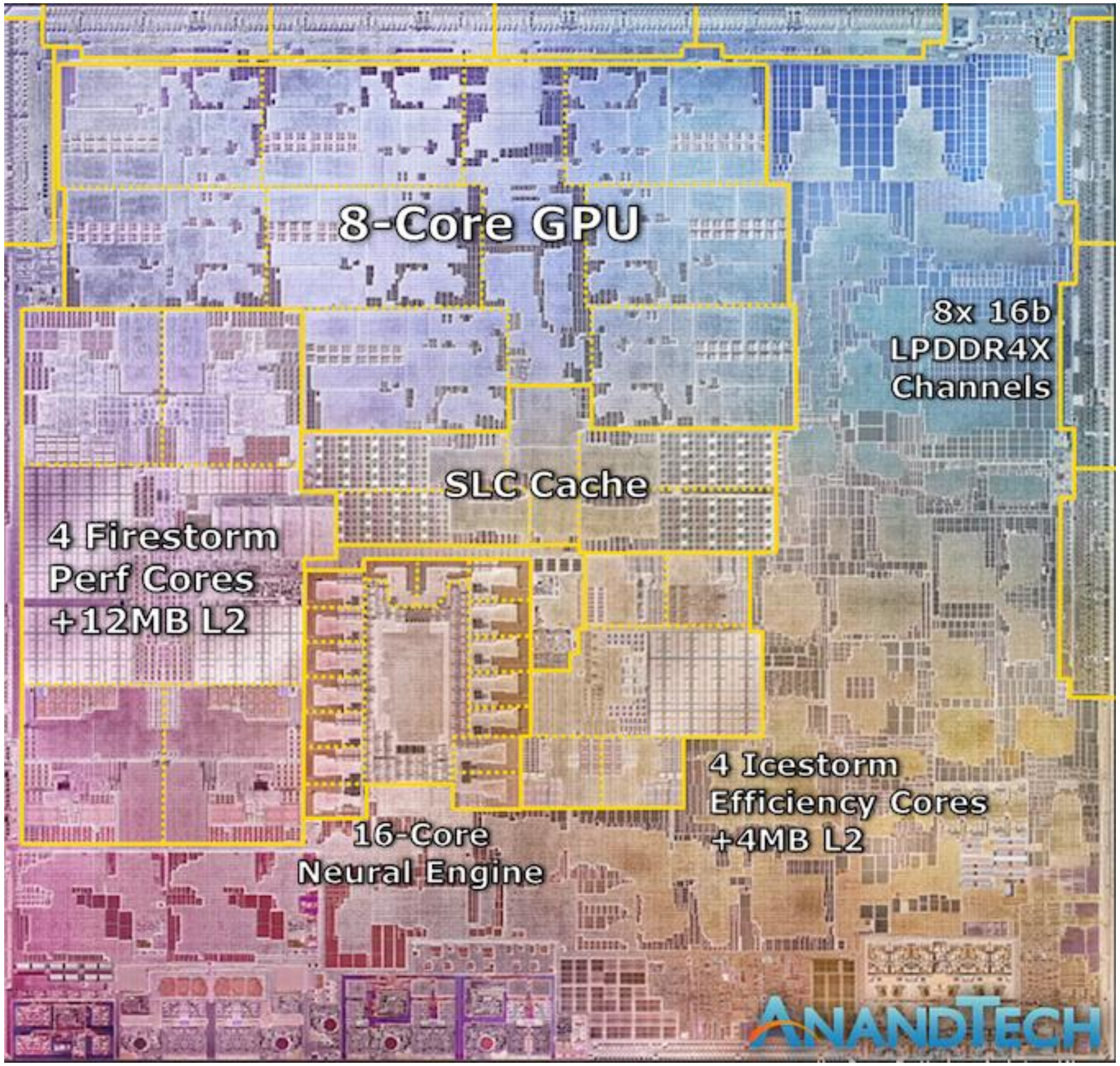

https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake

- 64-bit processor
- 4 cores, 8 threads
- 14-19 stage pipeline
- 3.9 GHz clock freq.

- 1.75B transistors

- In ~47 years, about 1,000,000-fold growth in transistor count and performance!

2021: Apple M1

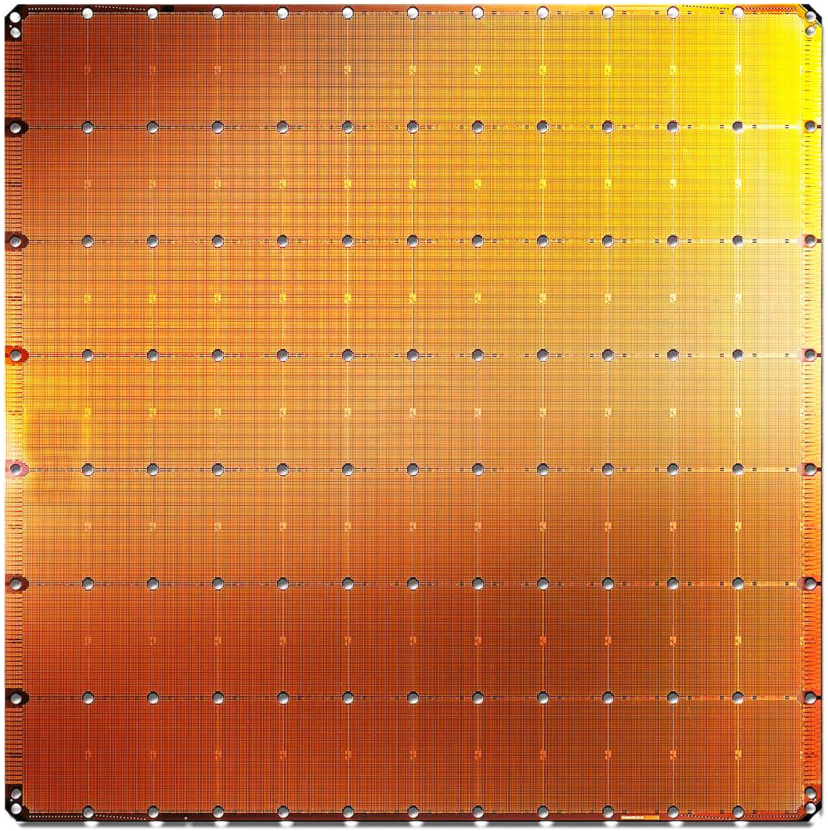


- 4 High-Perf GP Cores
- 4 Efficient GP Cores
- 8-Core GPU
- 16-Core Neural Engine
- Lots of Cache
- Many Caches
- 8x Memory Channels

- 16B transistors

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

2019: Cerebras Wafer Scale Engine



- The largest ML accelerator chip
- 400,000 cores

Cerebras WSE
1.2 Trillion transistors
46,225 mm^2

Largest GPU
21.1 Billion transistors
815 mm^2
NVIDIA TITAN V

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

How to Deal with This Complexity?

- **Hardware Description Languages!**

- **Needs and wants:**
  - Ability to 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 and specify 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

Hardware Description Languages

- **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

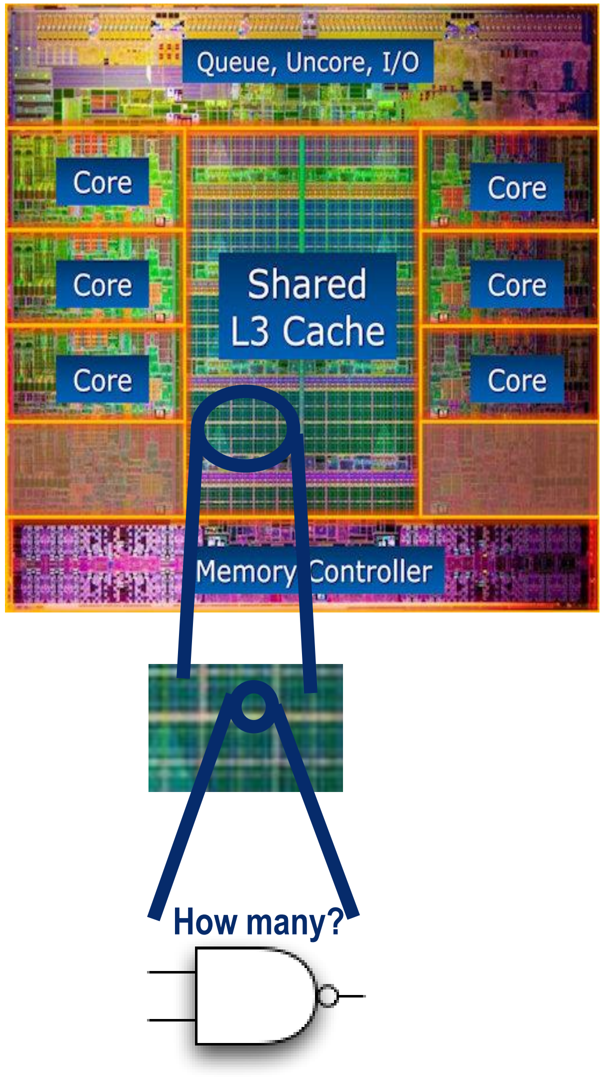
- **We will use Verilog in this course**

Hardware Design Using HDL

Principle: Hierarchical Design

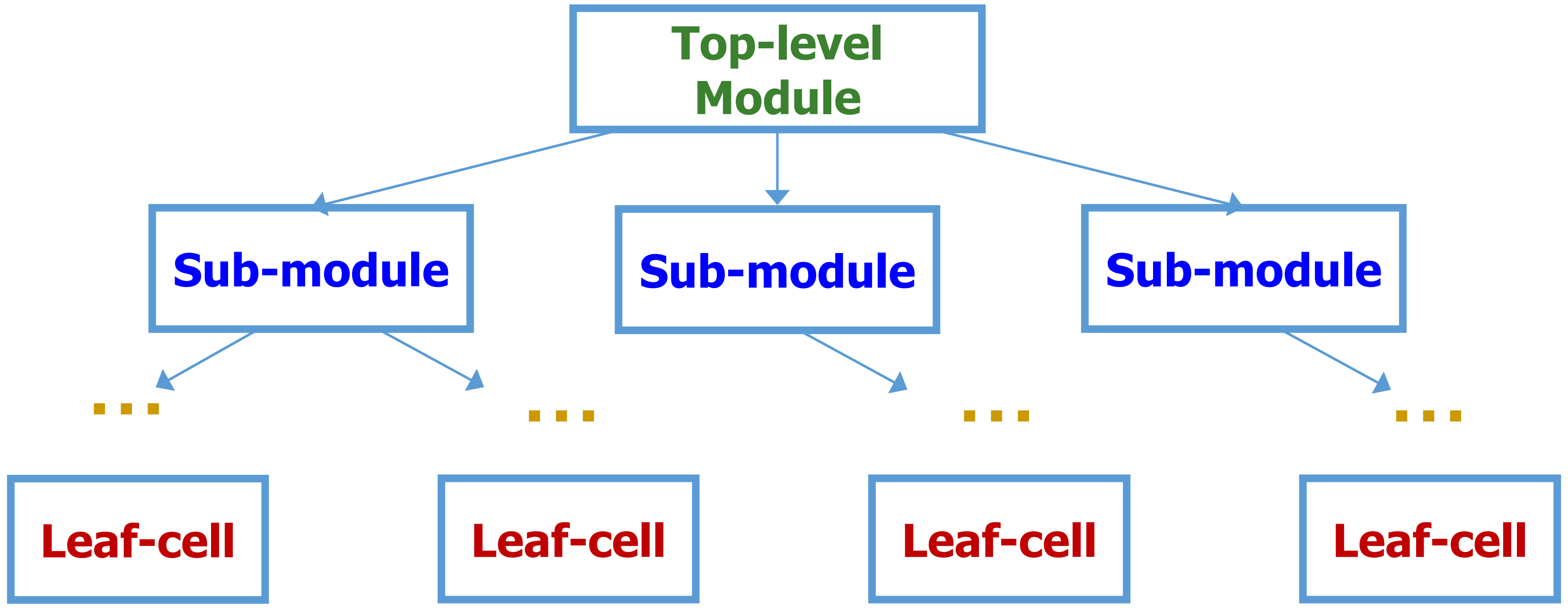
- **Design a hierarchy of modules**
  - Predefined "primitive" gates (AND, OR, …)
  - Simple modules are built by instantiating these gates (components like MUXes)
  - Complex modules are built by instantiating simple modules, …
- **Hierarchy controls complexity**
  - Analogous to the use of function/method 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)?

https://techreport.com/review/21987/intel-core-i7-3960x-processor



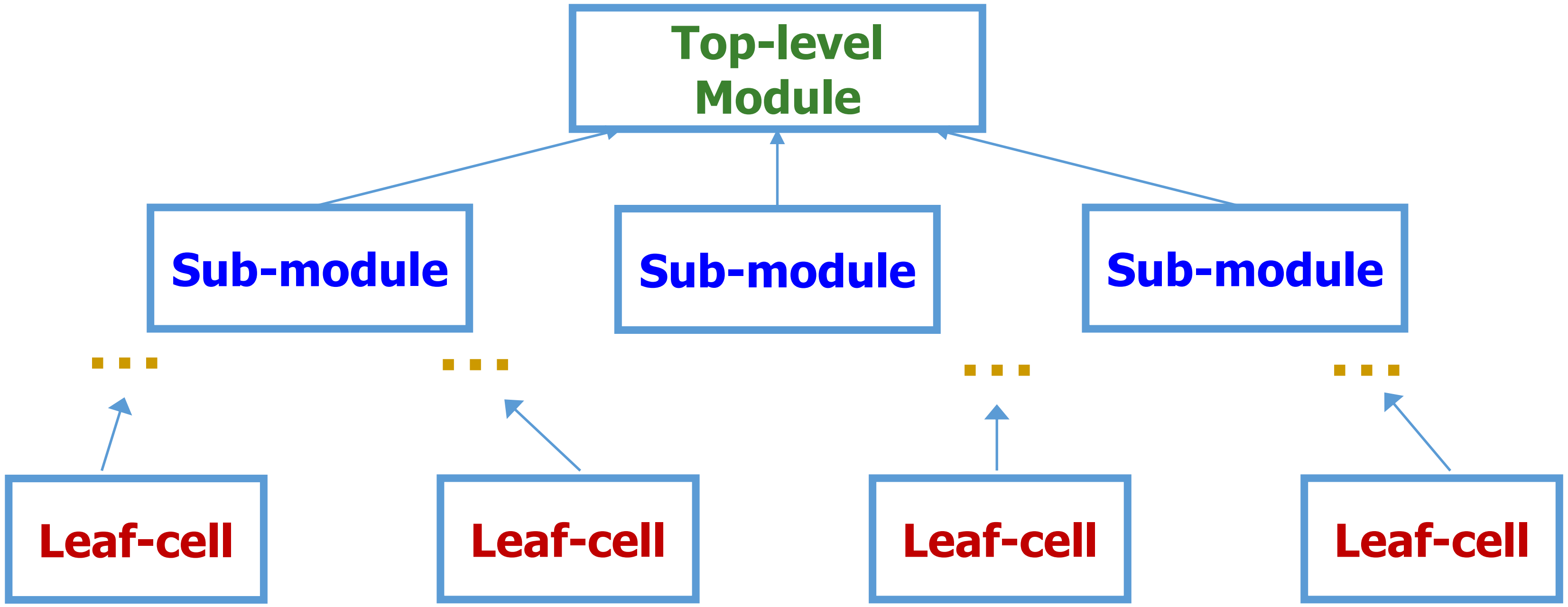
Top-Down Design Methodology

- **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, cell libraries*)



Bottom-Up Design Methodology

- 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



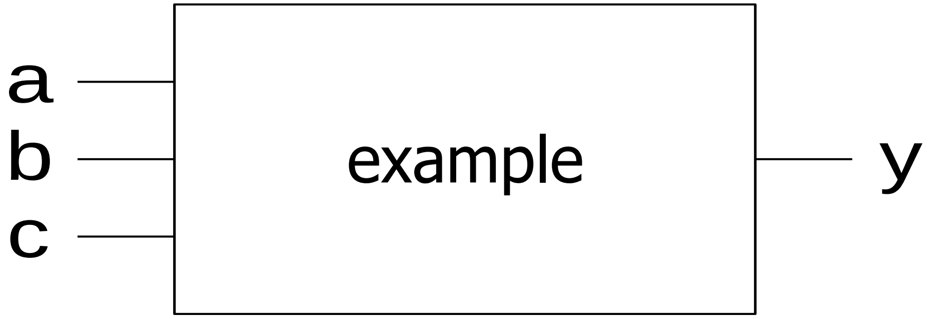
Defining a Module in Verilog

- **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

inputs
output

Implementing a Module in Verilog



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

// here comes the circuit description

endmodule
```

name of module

Port list (inputs and outputs)

ports have a declared type

a module definition

A Question of Style

- **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

What If We Have 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

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 */
```

Two Main Styles of HDL Implementation

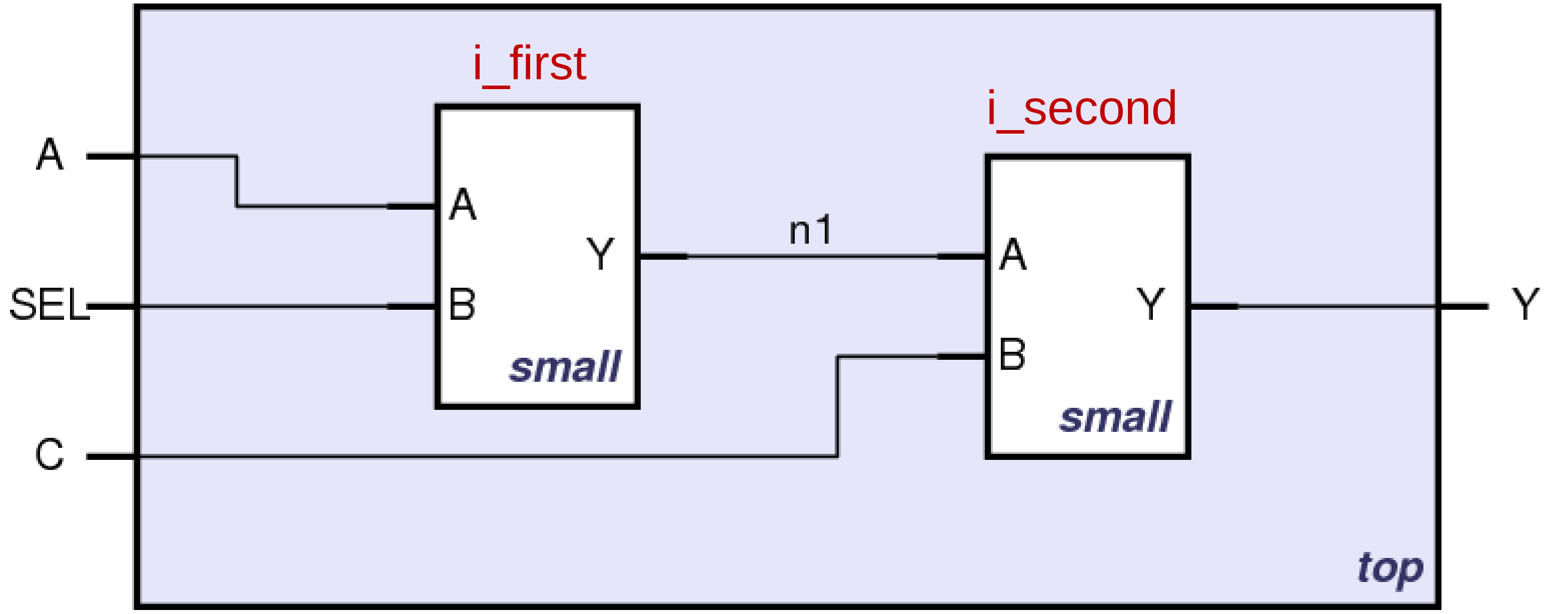
- **Structural (Gate-Level)**
  - The module body contains gate-level description of the circuit
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - … and interconnections between those modules
  - Describes a hierarchy of modules defined as gates

- **Behavioral**
  - The module body contains 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

Structural HDL: Instantiating a Module

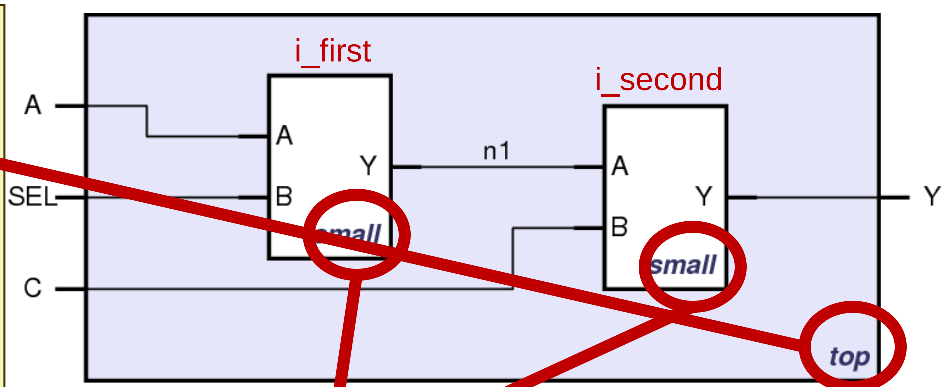


Schematic of module "top" that is built from two instances of module "small"

Structural HDL Example

- **Module Definitions in Verilog**

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








endmodule
```

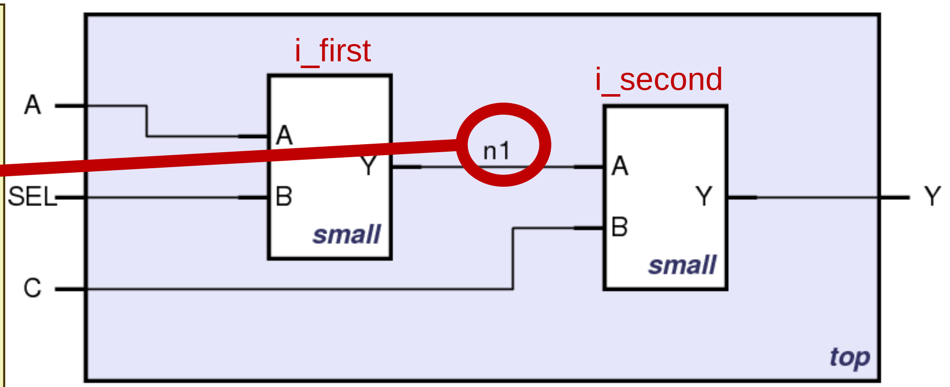


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

- **Defining wires (module interconnections)**

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








endmodule
```

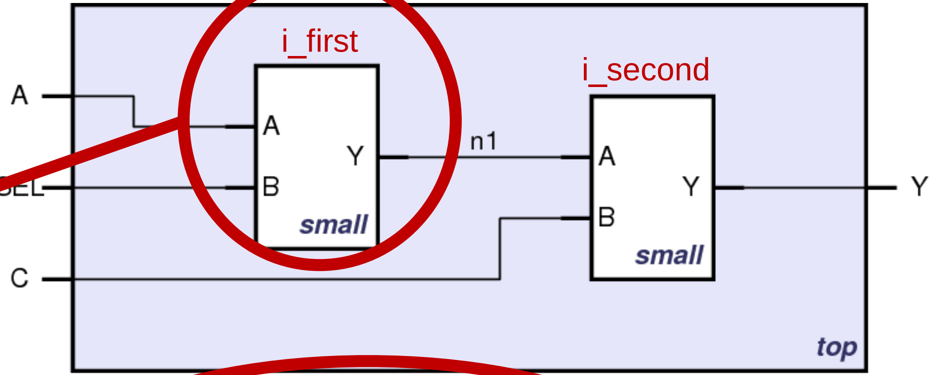


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

- The first 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)    );



endmodule
```

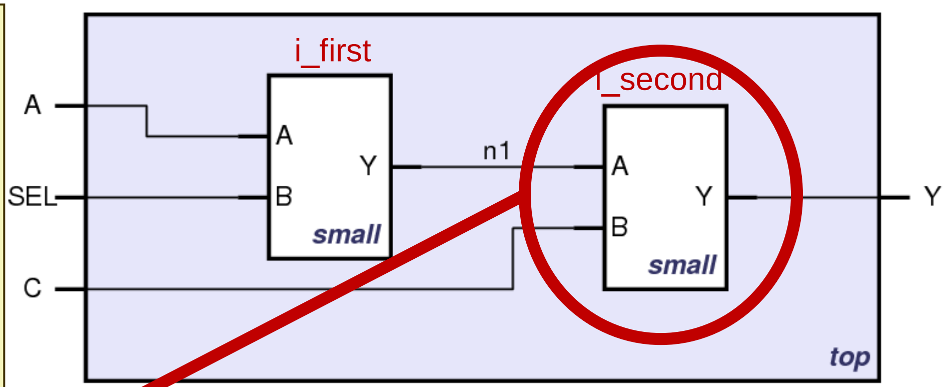


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

- 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
```

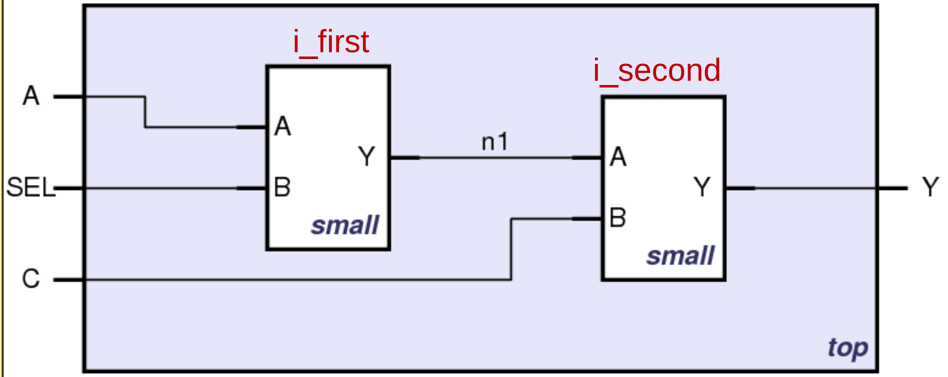


```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Structural HDL Example

- **Short form of module instantiation**

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

// alternative
small i_first ( A, SEL, n1 );

/* Shorter instantiation,
   pin order very important */

// any pin order, safer choice
small i_second ( .B(C),
                 .Y(Y),
                 .A(n1) );

endmodule
```



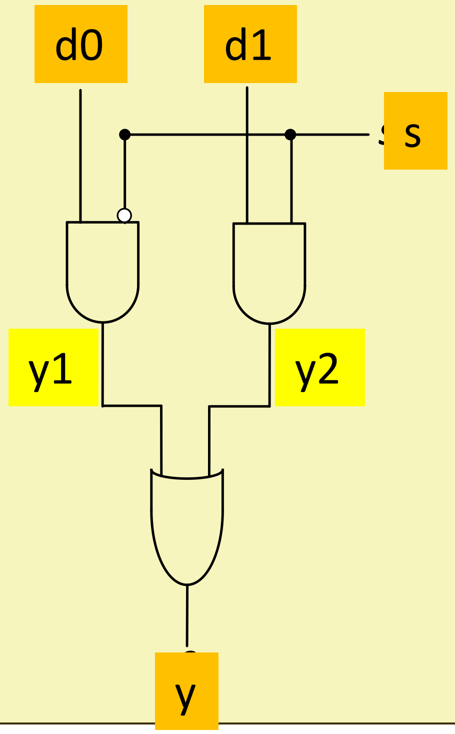
```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Short form is not good practice
as it reduces code maintainability

Structural HDL Example (II)

- **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

Recall: Two Main Styles of HDL Implementation

- **Structural (Gate-Level)**
  - The module body contains 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 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

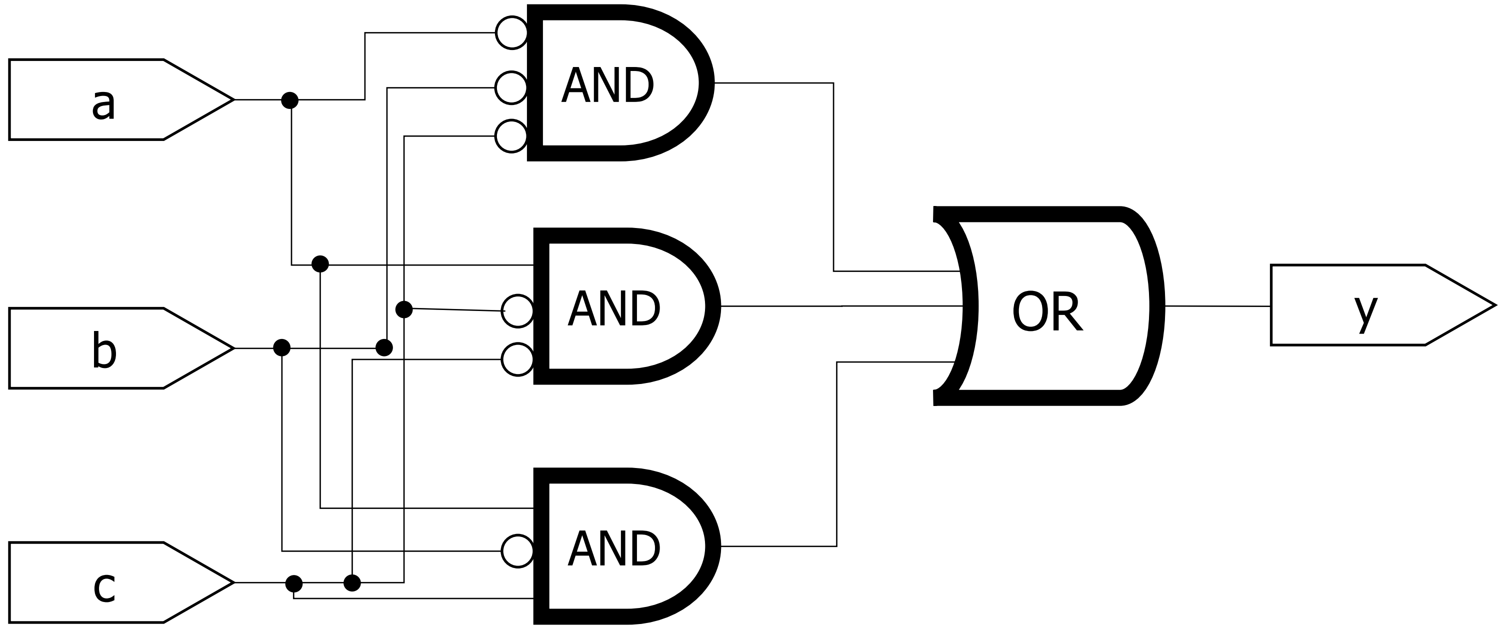
- **Many practical designs use a combination of both**

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

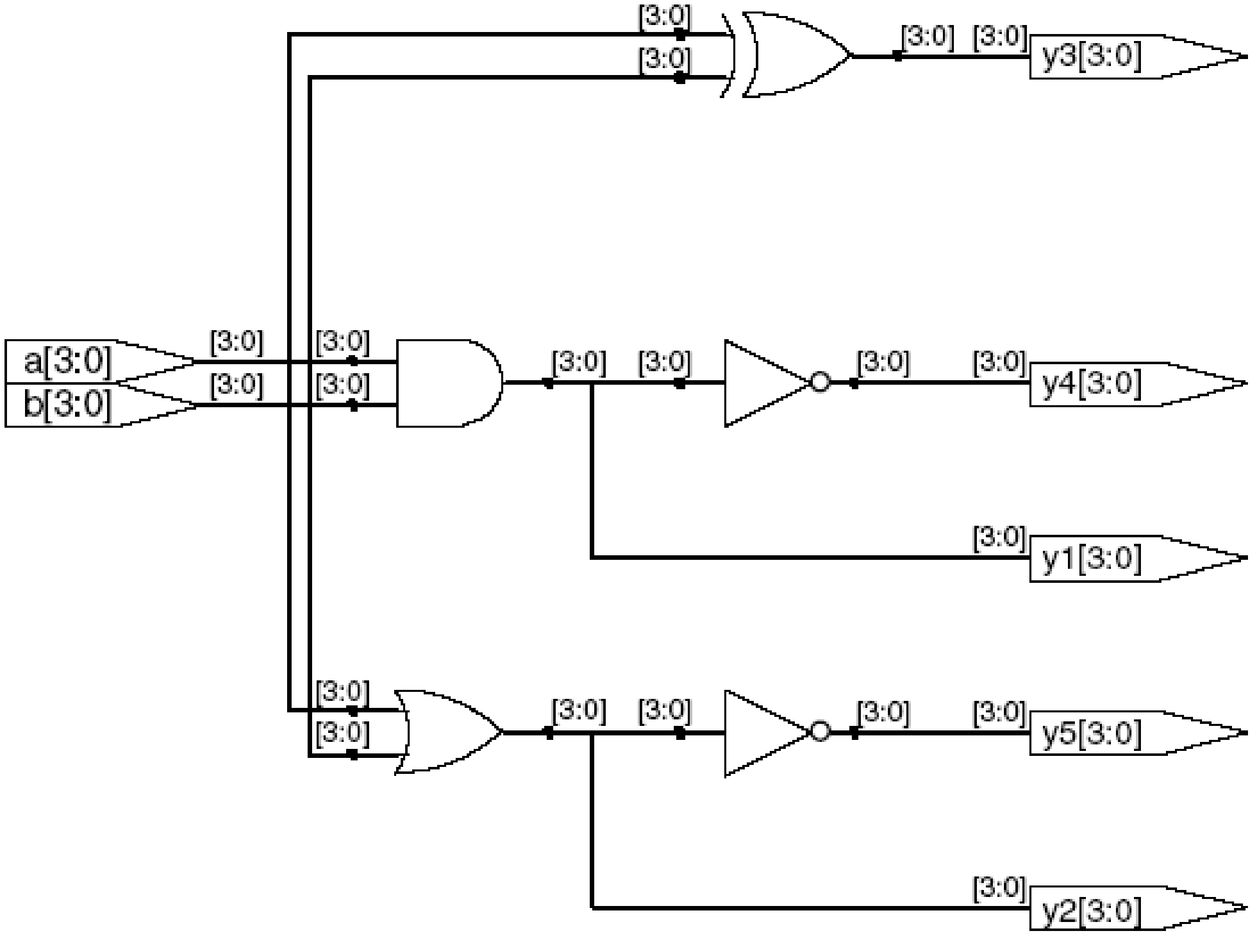
A behavioral implementation still models a hardware circuit!



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
```

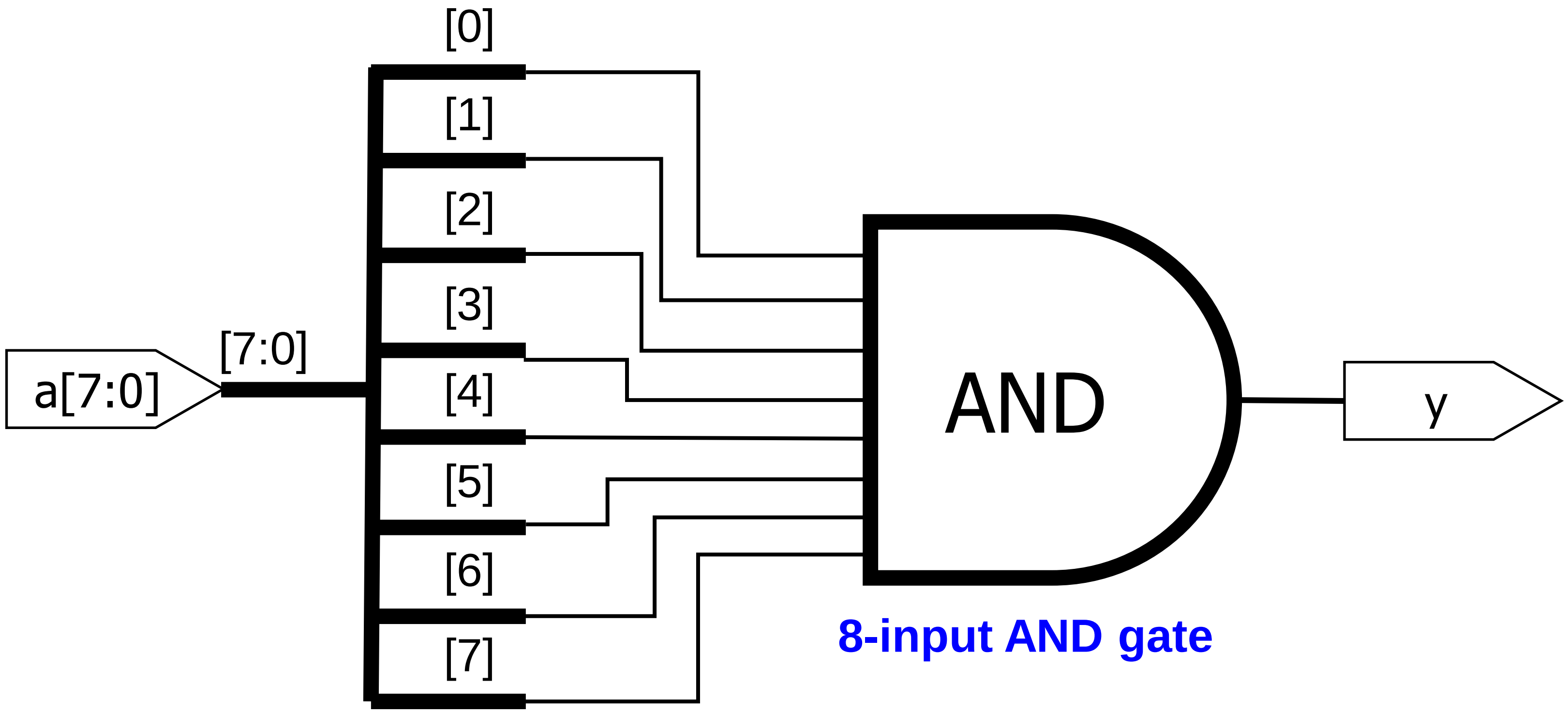
Bitwise Operators: Schematic View



Reduction Operators in Behavioral Verilog

```
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
```

Reduction Operators: Schematic View



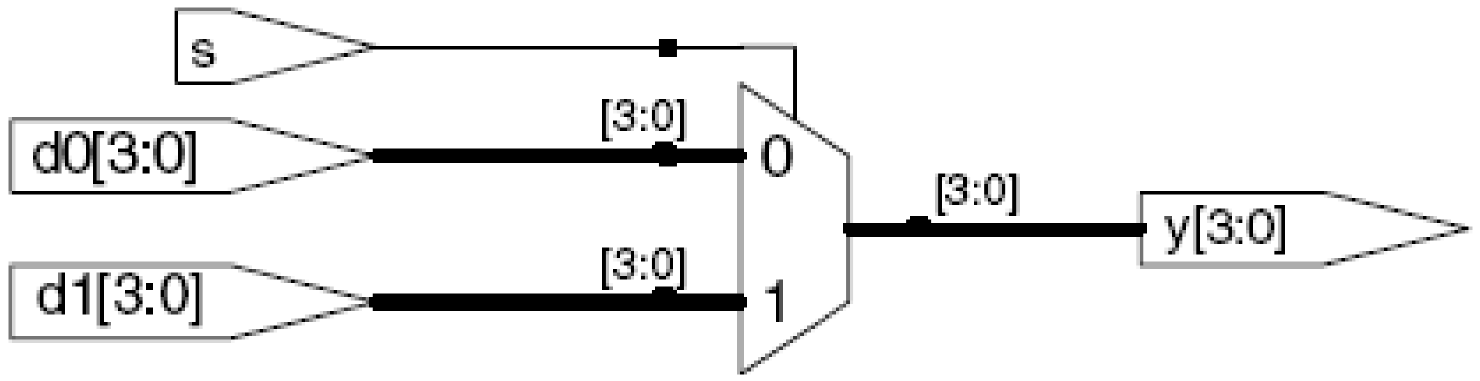
8-input AND gate

Conditional Assignment in Behavioral Verilog

```
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
```

- **? : is also called a ternary operator as it operates on three inputs:**
  - s
  - d1
  - d0

Conditional Assignment: Schematic View



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
```

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
```

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

How to Express Numbers?

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

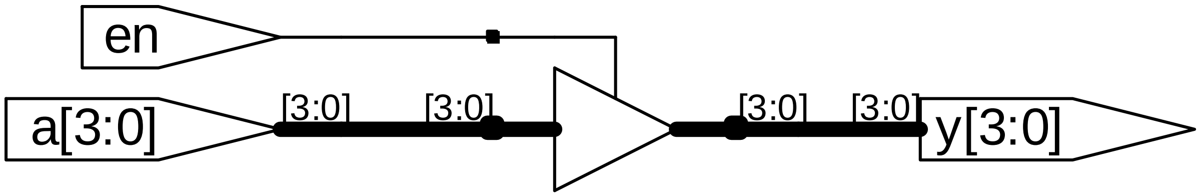
Number Representation in Verilog

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	$\underbrace{\text{0000 .. 0001}}_{\text{32 bits (default)}}$	12’h0	0000 0000 0000

Reminder: Floating Signals (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**

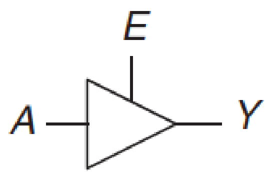
```
module tristate_buffer(input  [3:0] a,
                       input        en,
                       output [3:0] y);

   assign y = en ? a : 4'bz;

endmodule
```



Tri-State Buffer

- **A tri-state buffer enables gating of different signals onto a wire**

Tristate
Buffer



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

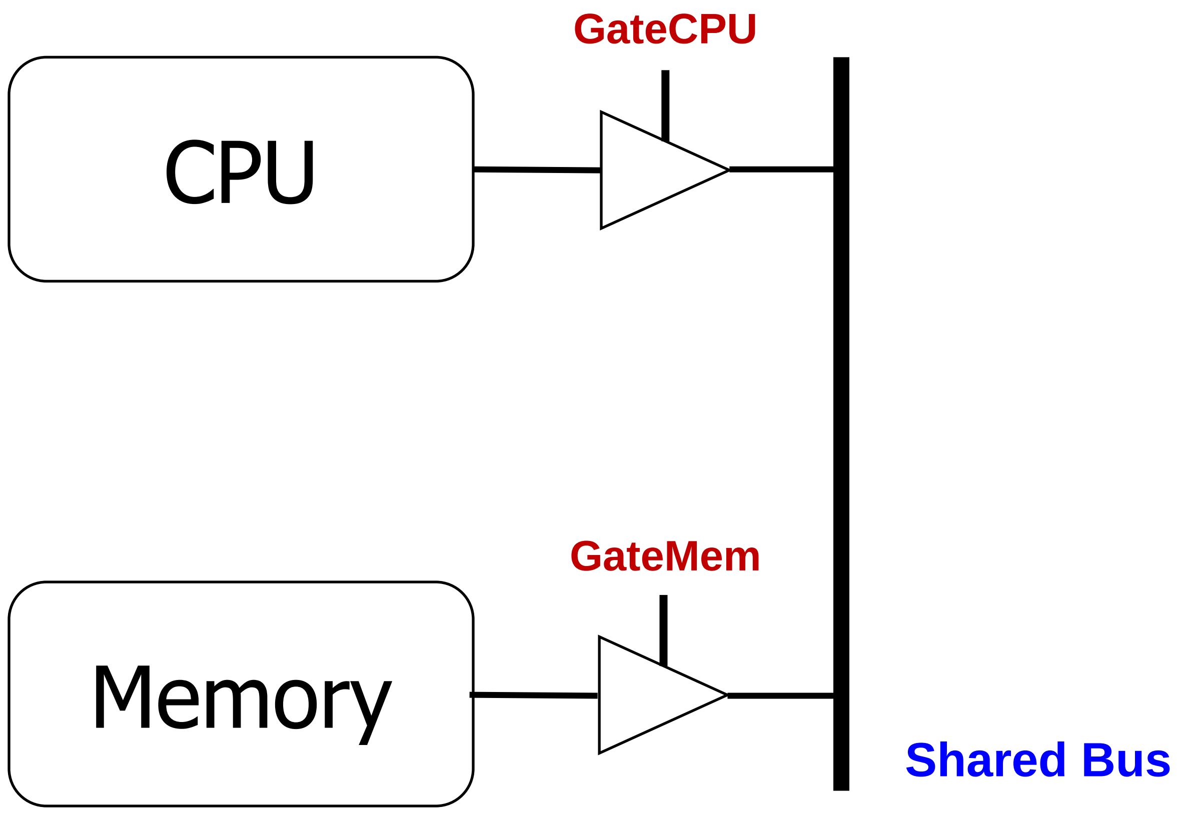
Figure 2.40 **Tristate buffer**

- **Floating signal (Z): Signal that is not driven by any circuit**
  - Open circuit, floating wire

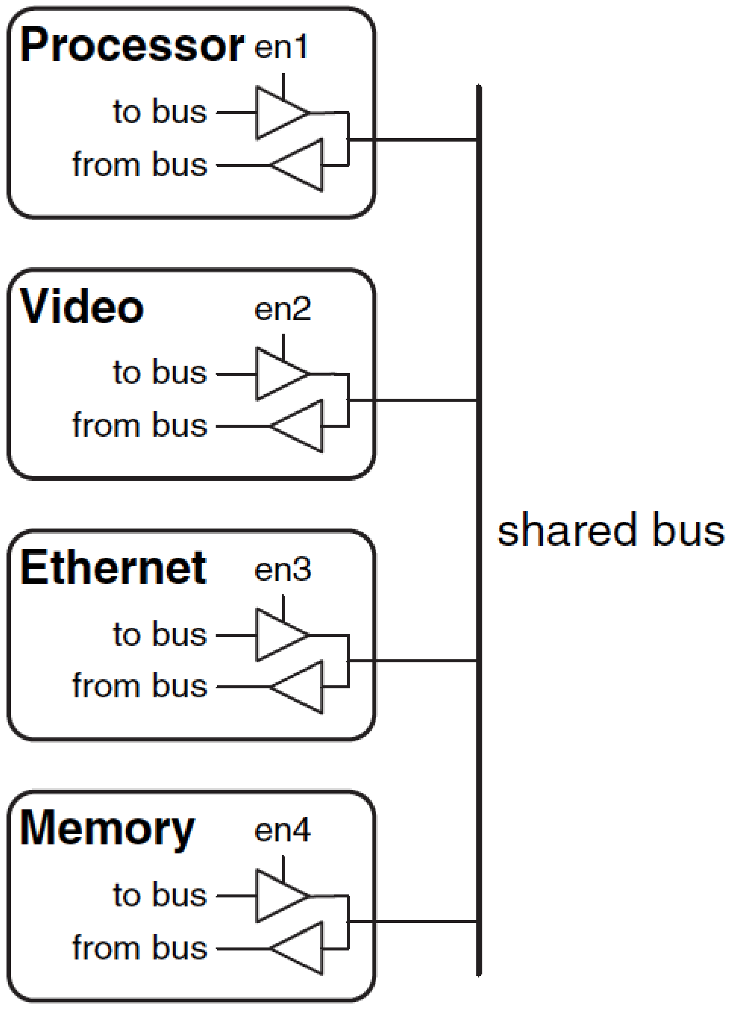
Example: Use of Tri-State Buffers

- **Imagine a wire connecting the CPU and memory**
  - At any time only the CPU or the memory can place a value on the wire, both not both
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

Example Design with Tri-State Buffers



Another Example



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

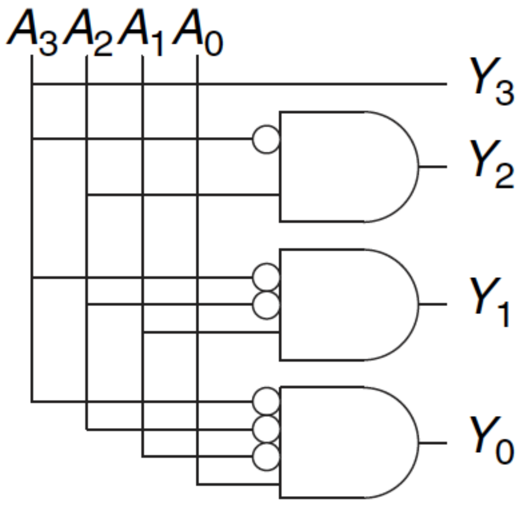
Recall: Simplified Priority Circuit

- **Priority Circuit**
  - Inputs: "Requestors" with priority levels
  - Outputs: "Grant" signal for each requestor
  - Example 4-bit priority circuit

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 **Priority circuit truth table with don't cares (X's)**



X (Don't Care) means *I don't care what the value of this input is*

What Happens with HDL Code?

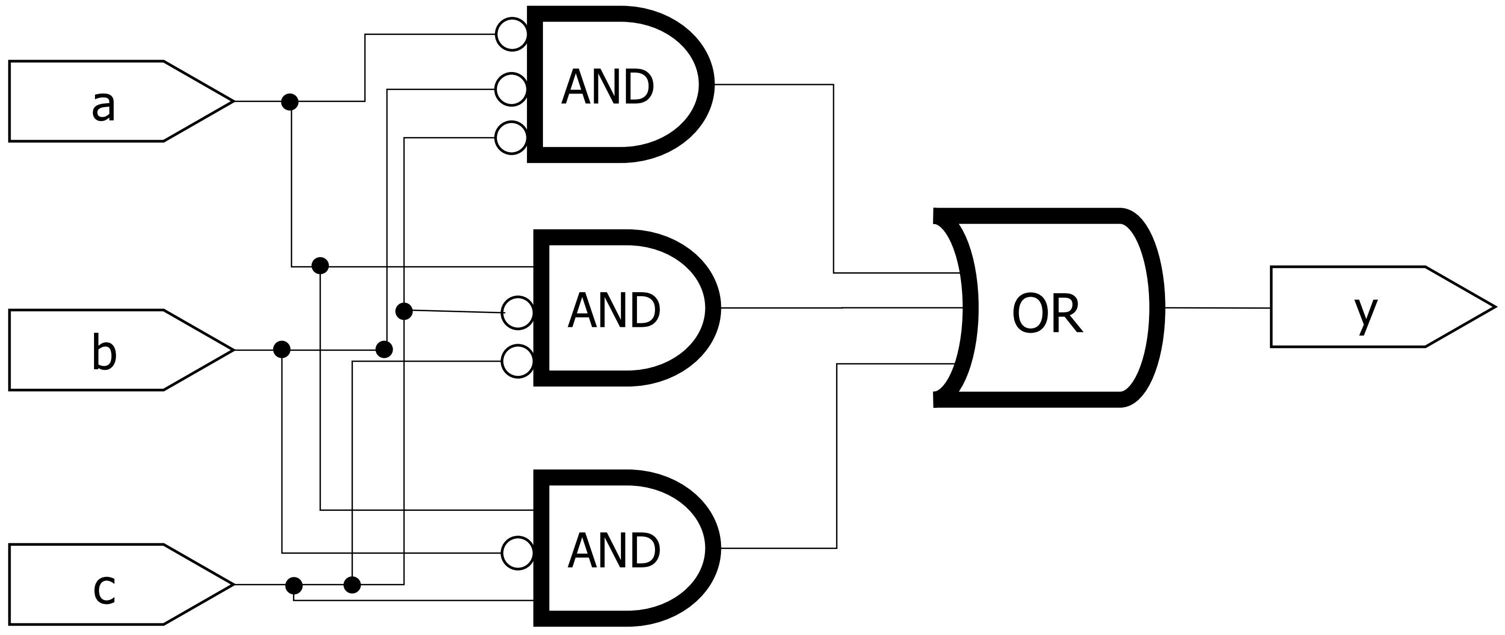
- **Synthesis (i.e., Hardware Synthesis)**
  - Modern tools are able to map ***synthesizable*** *HDL code* into low-level *cell libraries* → *netlist describing gates and wires*
  - They can perform many optimizations
  - … however they can **not** guarantee that a solution is optimal
    - Mainly due 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

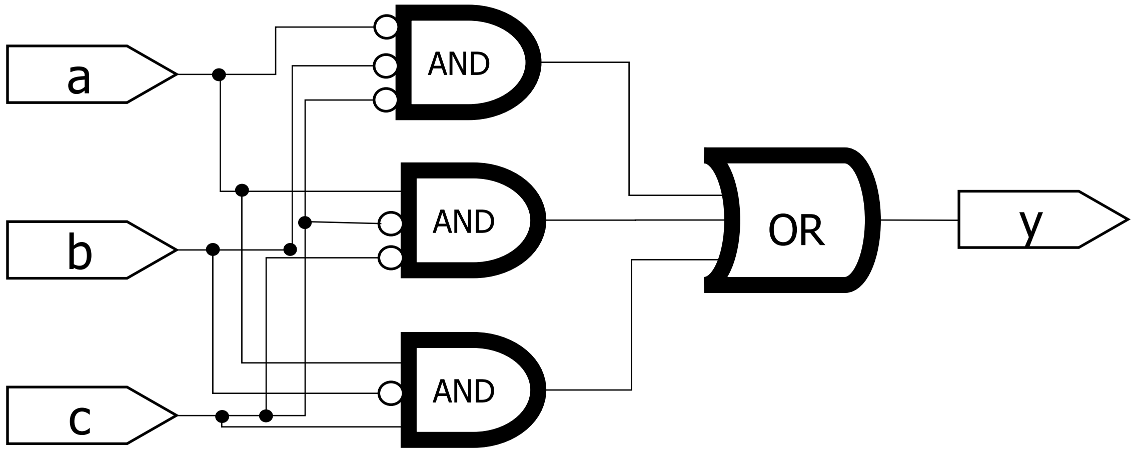
Recall This "example"

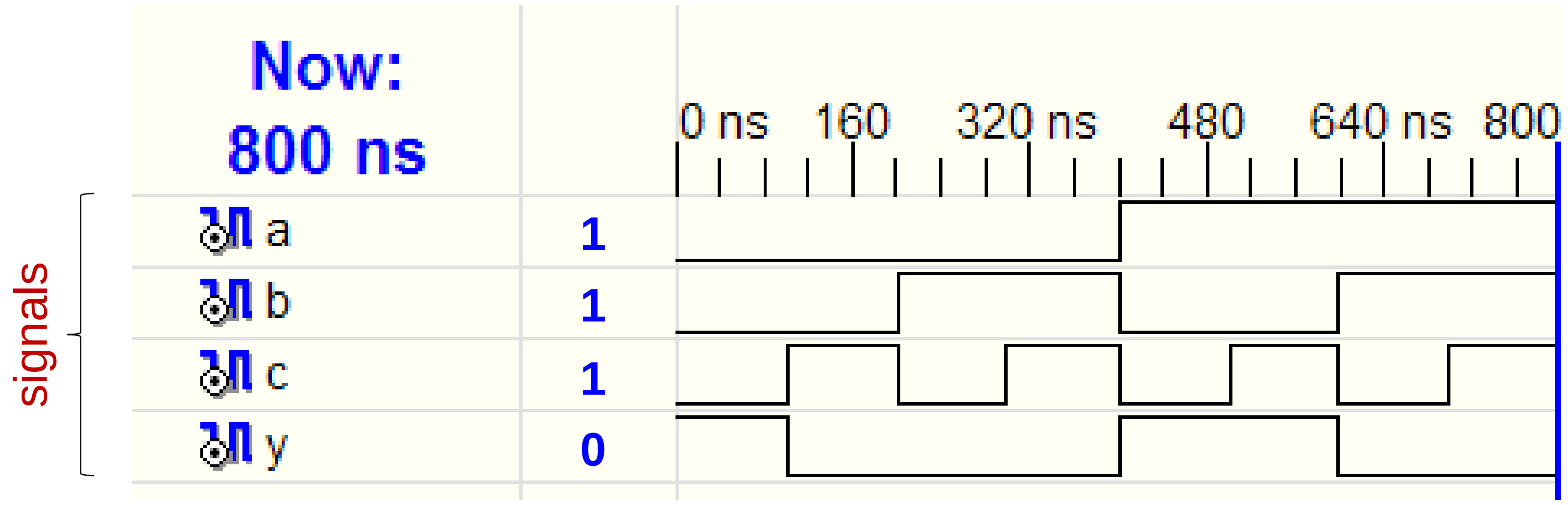
```
module example (a, b, c, y);
      input a;
      input b;
      input c;
      output y;

// here comes the circuit description
assign y = ~a & ~b & ~c |
            a & ~b & ~c |
            a & ~b &  c;

endmodule
```

Synthesizing the "example"



Simulating the "example"





Waveform Diagram

time

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.

What We Have Seen So Far

- **Describing structural hierarchy with Verilog**
  - Instantiate modules in an other module
- **Describing functionality using behavioral modeling**

- **Writing simple logic equations**
  - We can write AND, OR, XOR, …
- **Multiplexer functionality**
  - If … then … else

- **We can describe constants**

- **But there is more...**

More Verilog Examples

- **We can write Verilog code in many different ways**

- **Let's see how we can express the same functionality by developing Verilog code**

  - At a low-level of abstraction
    - Poor readability
    - More optimization opportunities (especially for low-level tools)

  - At a high-level of abstraction
    - Better readability
    - Limited optimization opportunities

Comparing Two Numbers

- Defining your own gates as new modules

- We will use our gates to show the different ways of implementing a 4-bit comparator (equality checker)

An XNOR gate

```
module MyXnor (input A, B,
               output Z);

   assign Z = ~(A ^ B); //not XOR

endmodule
```

An AND gate

```
module MyAnd (input A, B,
              output Z);

   assign Z = A & B;    // AND

endmodule
```

Writing More 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

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
```

What About Timing?

- **It is possible to define *timing relations* in Verilog. BUT:**
  - These are ***ONLY*** for simulation
  - They ***CAN NOT*** be synthesized
  - They are used for *modeling delays* in a circuit

```
‘timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a;  // output after 9ns

endmodule
```

More to come later today!

Good Practices

- **Develop/use a consistent naming style**
- **Use MSB to LSB ordering for buses**
  - Use "`a[31:0]`", **not** "`a[0:31]`"
- **Define one module per file**
  - Makes managing your design hierarchy easier
- **Use a file name that equals module name**
  - e.g., module TryThis is defined in a file called TryThis.v
- **Always keep in mind that Verilog describes hardware**

Summary (HDL for Combinational Logic)

- We have seen an overview of Verilog

- Discussed structural and behavioral modeling

- Studied combinational logic constructs