# Digital Design & Computer Arch.

## Lecture 7: Hardware Description Languages and Verilog

Prof. Onur Mutlu

ETH Zürich

Spring 2022

17 March 2022

# Readings (This Week)

- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full

- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By tomorrow, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

# Readings (Next Week)

- Von Neumann Model, LC-3, and MIPS
  - P&P, Chapter 4, 5
  - H&H, Chapter 6
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)

- Programming
  - P&P, Chapter 6

- **Recommended:** Digital Building Blocks
  - H&H, Chapter 5

# **Assignment:** Lecture Video (April 1)

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities

- **Required Assignment**
  - **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
  - https://www.youtube.com/watch?v=kgiZlSOcGFM (May 2017)
  - https://www.youtube.com/watch?v=mskTeNnf-i0 (Feb 2021)

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page summary** of one of the lectures and email us
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle by April 1

# **Extra Assignment:** Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: April 7

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)

# **Extra Assignment 2:** Moore's Law (II)

- **Guidelines on how to review papers critically**
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c
  - Example reviews on "Main Memory Scaling: Challenges and Solution Directions" (link to the paper)
    - Review 1
    - Review 2
  - Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" (link to the paper)
    - Review 1

# Agenda for Today

- Hardware Description Languages
- Implementing Combinational Logic (in Verilog)
- Implementing Sequential Logic (in Verilog)

- The Verilog slides constitute a tutorial. We may not cover all.
- All slides will be beneficial for your labs.

# What We Will Cover Soon: LC-3 Processor

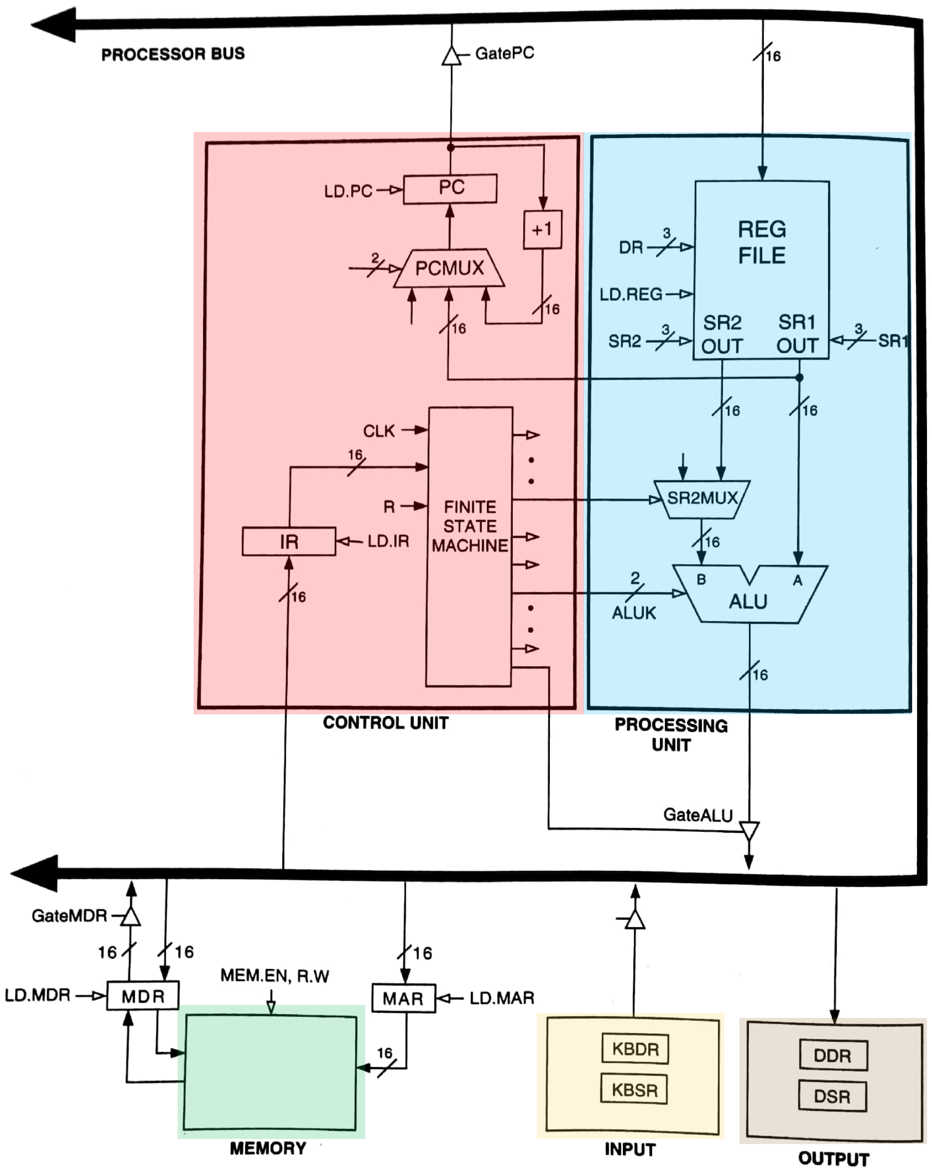
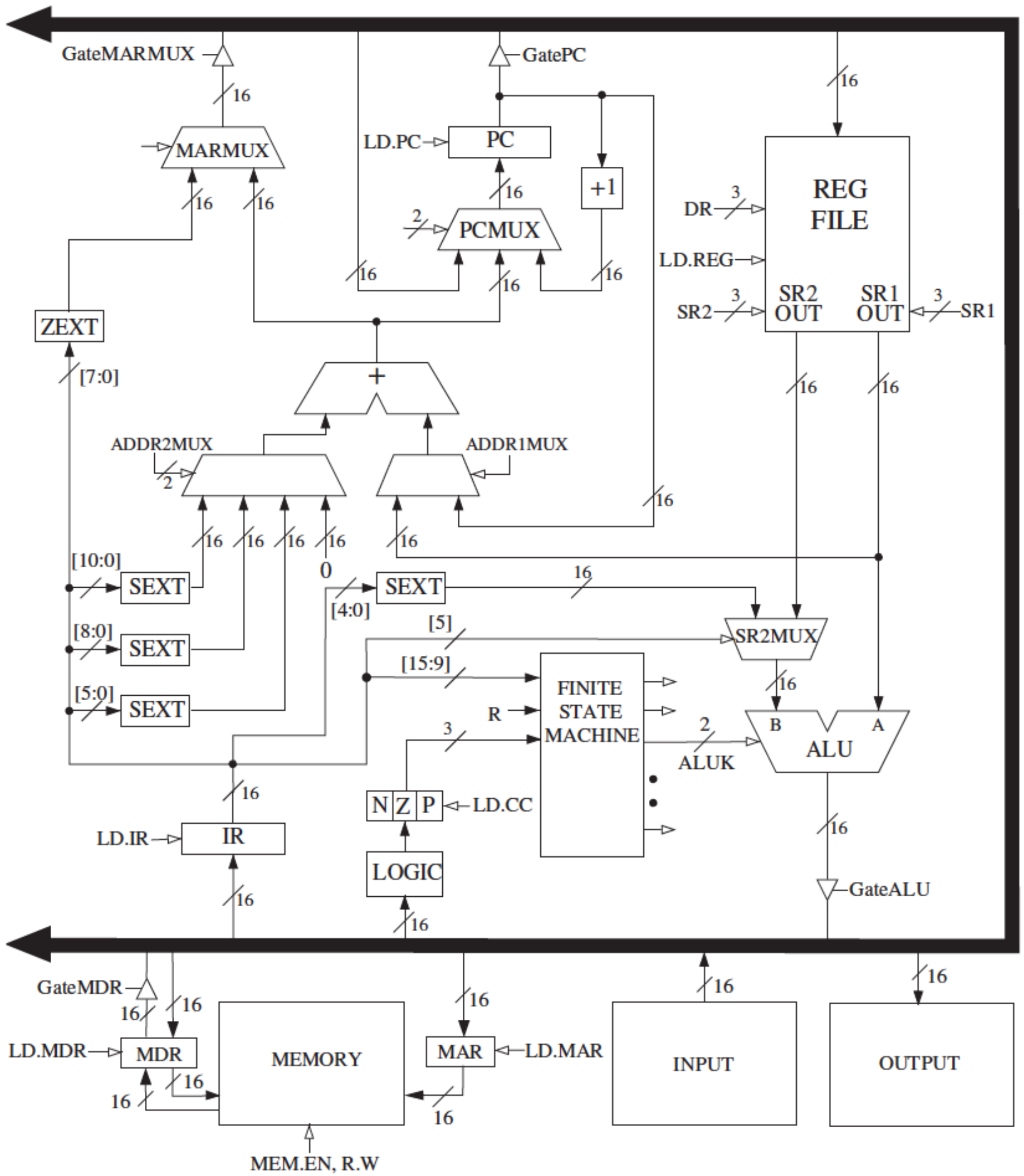


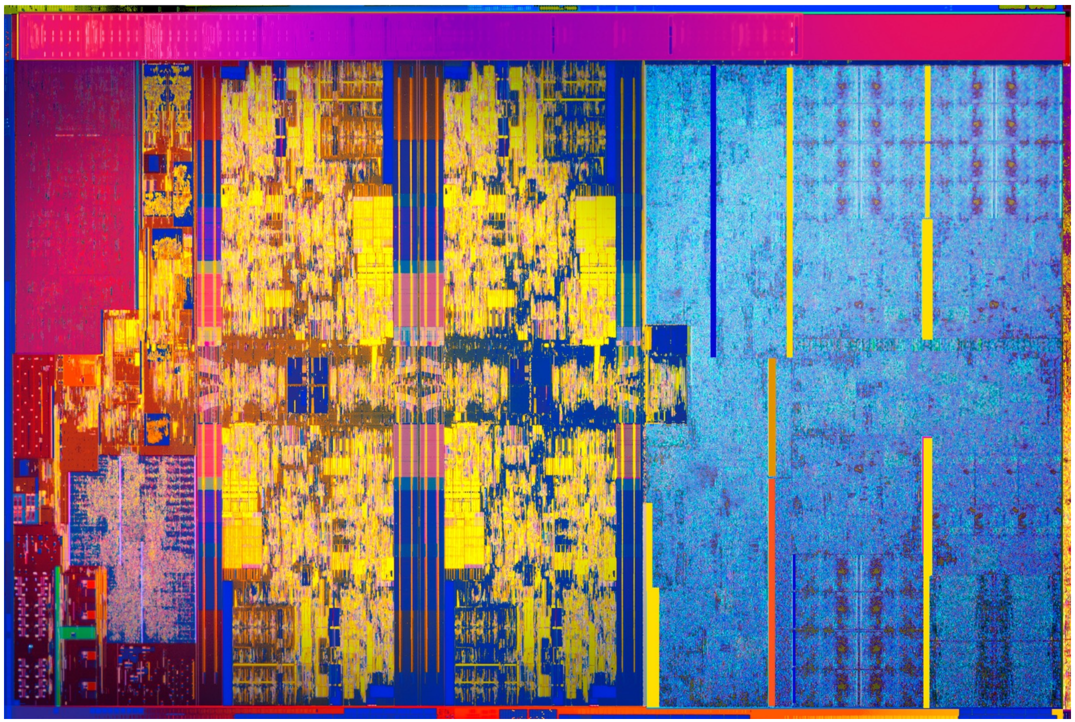

**Figure 4.3** The LC-3 as an example of the von Neumann model

# What We Will Cover Soon: LC-3 Datapath



# Hardware Description Languages & Verilog

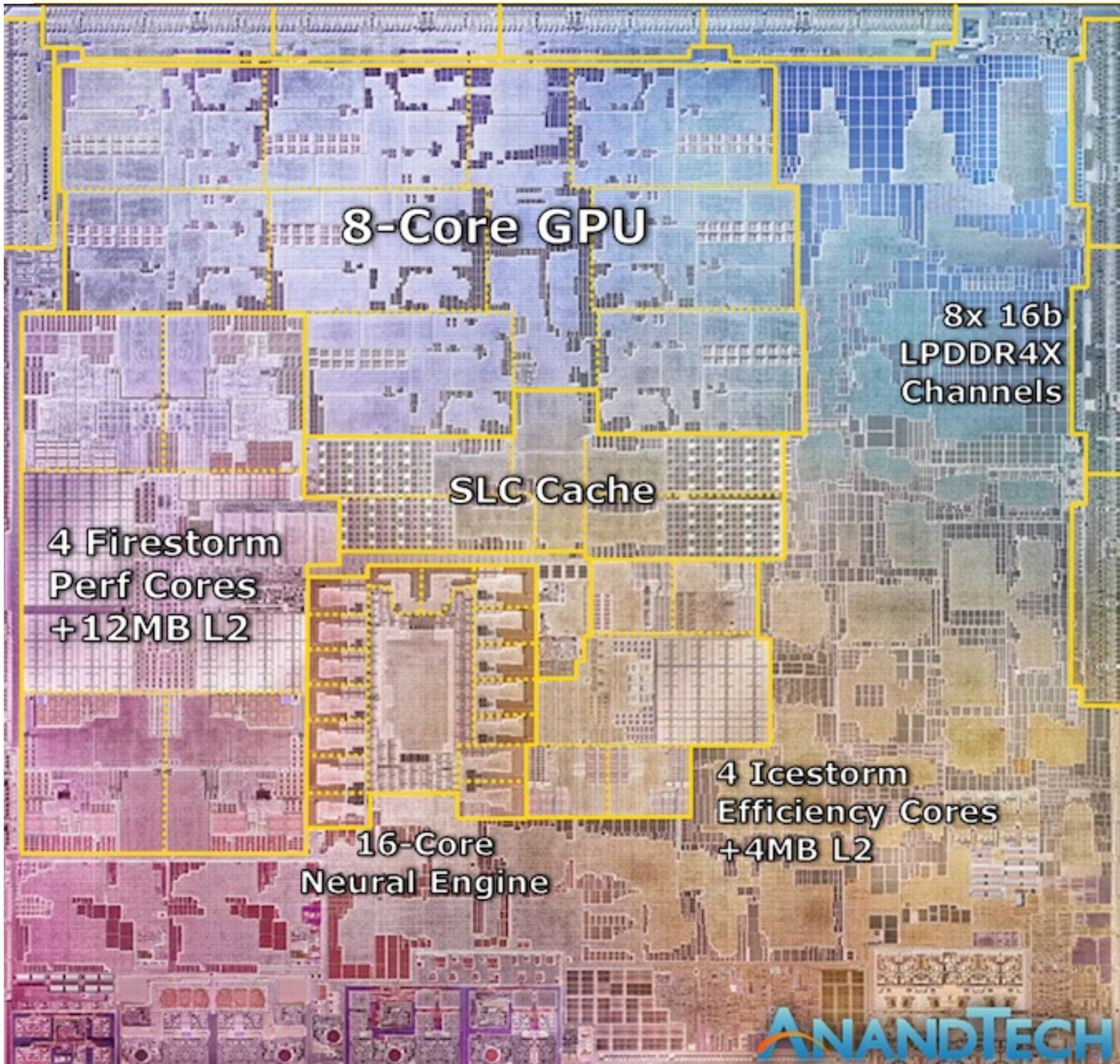
# 2017: Intel Kaby Lake



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

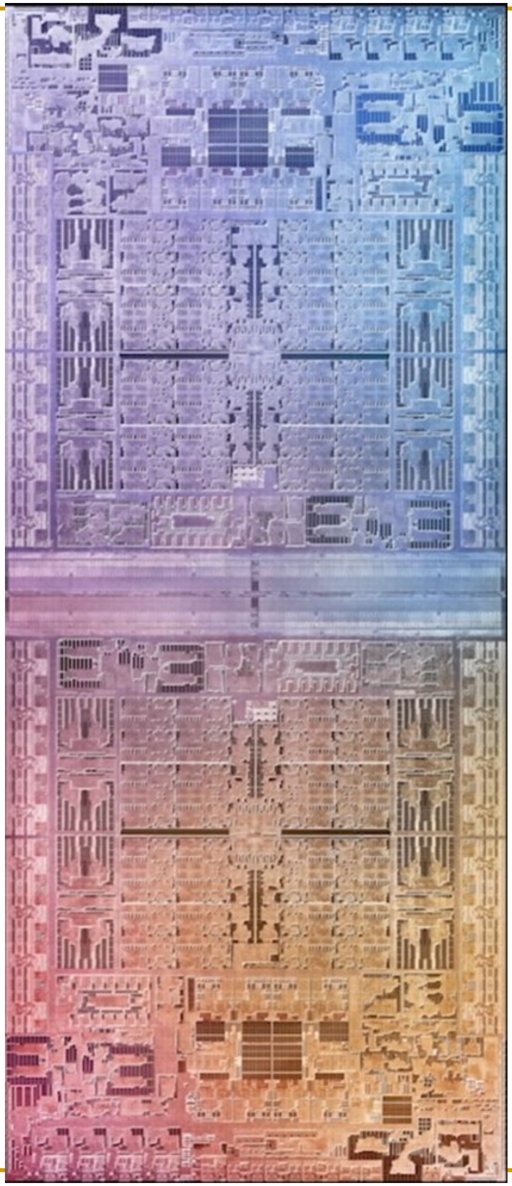



- 4 High-Perf GP Cores
- 4 Efficient GP Cores
- 8-Core GPU
- 16-Core Neural Engine
- Lots of Cache
- Many Caches
- 8x Memory Channels

- 16B transistors

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

# 2022: Apple M1 Ultra

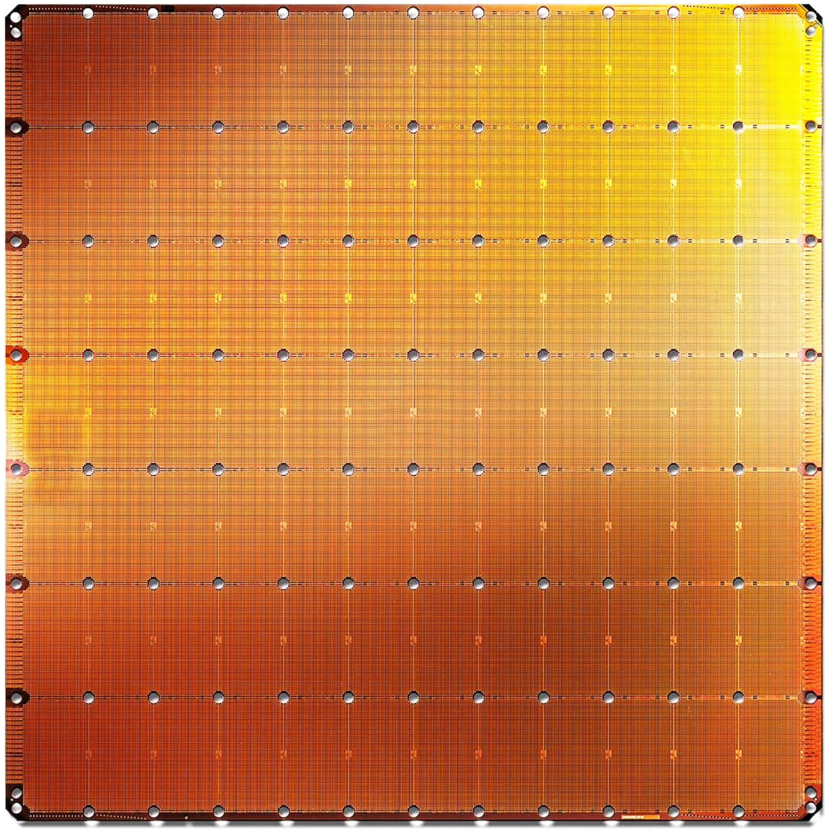


- 16 High-Perf GP Cores
- 4 Efficient GP Cores
- 64-Core GPU
- 32-Core Neural Engine
- Lots of Cache
- Many Caches
- 32x Memory Channels
- 128 GB DRAM

- 114B transistors

https://www.theverge.com/2022/3/9/22968611/apple-m1-ultra-gpu-nvidia-rtx-3090-comparison

# 2019: Cerebras Wafer Scale Engine



- The largest ML accelerator chip (2019)
- 400,000 cores

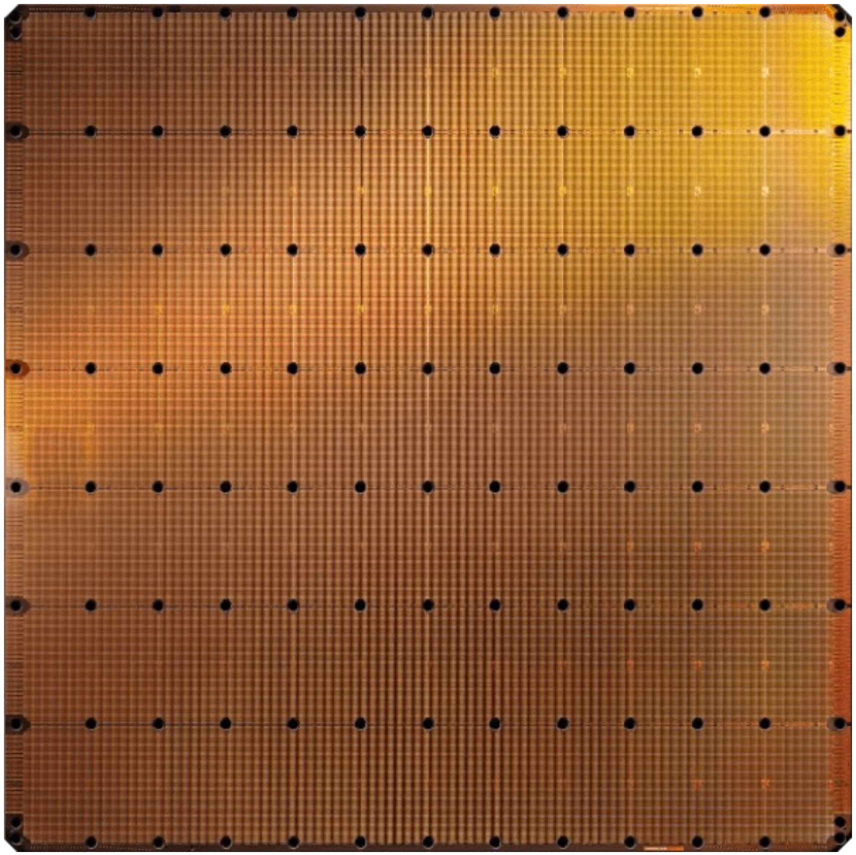

**Cerebras WSE**
**1.2 Trillion transistors**
**46,225 mm²**

**Largest GPU**
**21.1 Billion transistors**
**815 mm²**
**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/

# 2021: Cerebras Wafer Scale Engine 2



- The largest ML accelerator chip (2021)
- 850,000 cores


**Cerebras WSE-2**
**2.6 Trillion transistors**
**46,225 mm²**

**Largest GPU**
**54.2 Billion transistors**
**826 mm²**
**NVIDIA** Ampere GA100

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

- What we need for hardware design:
  - 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

# Why Specialized Languages for Hardware?

- 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

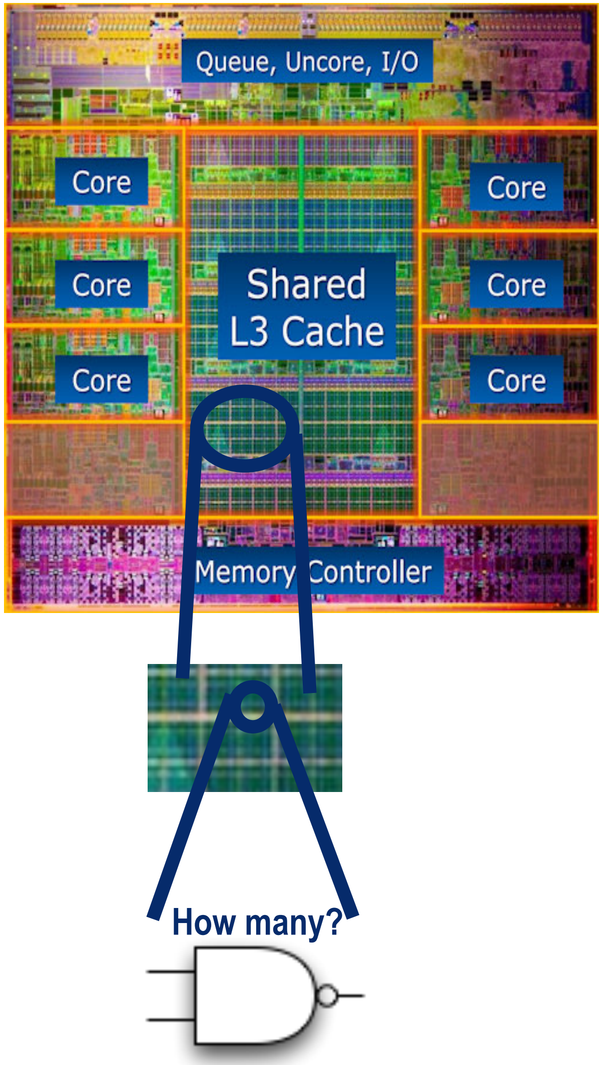
- Both of the above ease **specification, simulation & synthesis**

# Hardware Design Using HDL

# Key Design Principle: Hierarchical Design

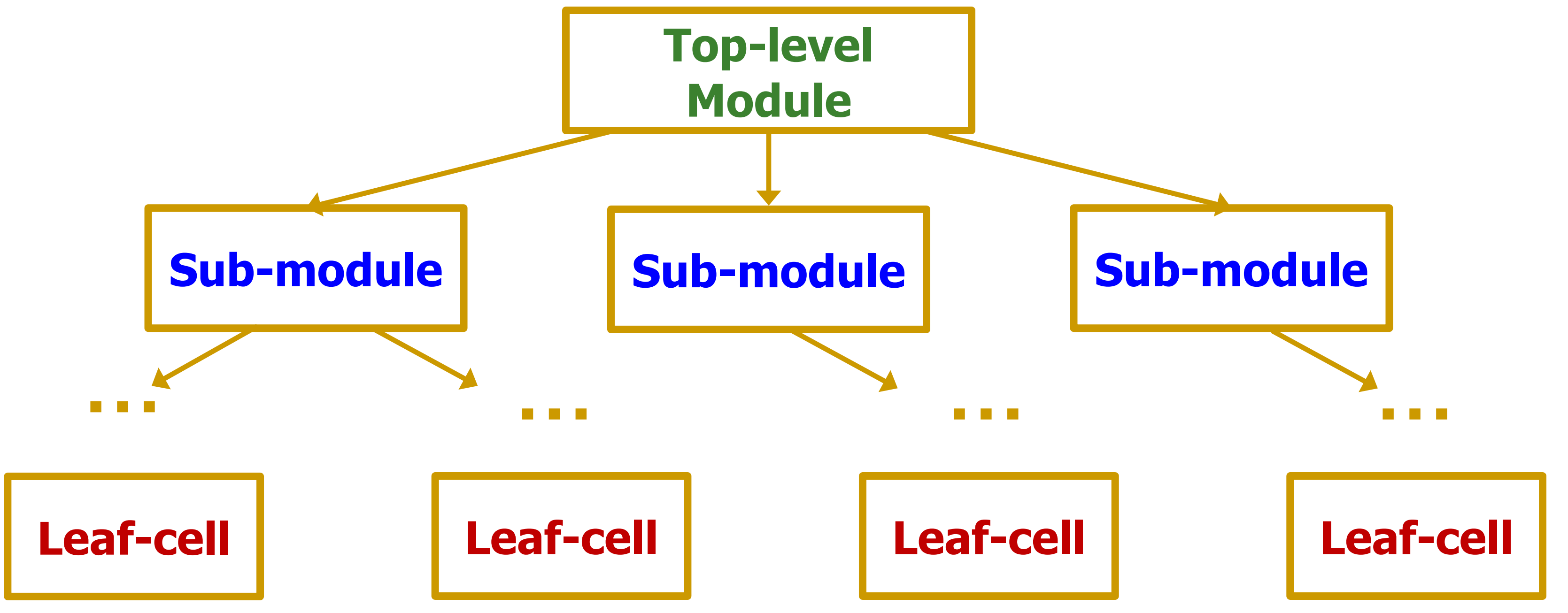
- **Design a hierarchy of modules**
  - Predefined "primitive" gates (AND, OR, …)
  - Simple modules are built by instantiating these gates (e.g., components like MUXes)
  - Complex modules are built by instantiating simple modules, …
- **Hierarchy controls complexity**
  - Analogous to the use of function/method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

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



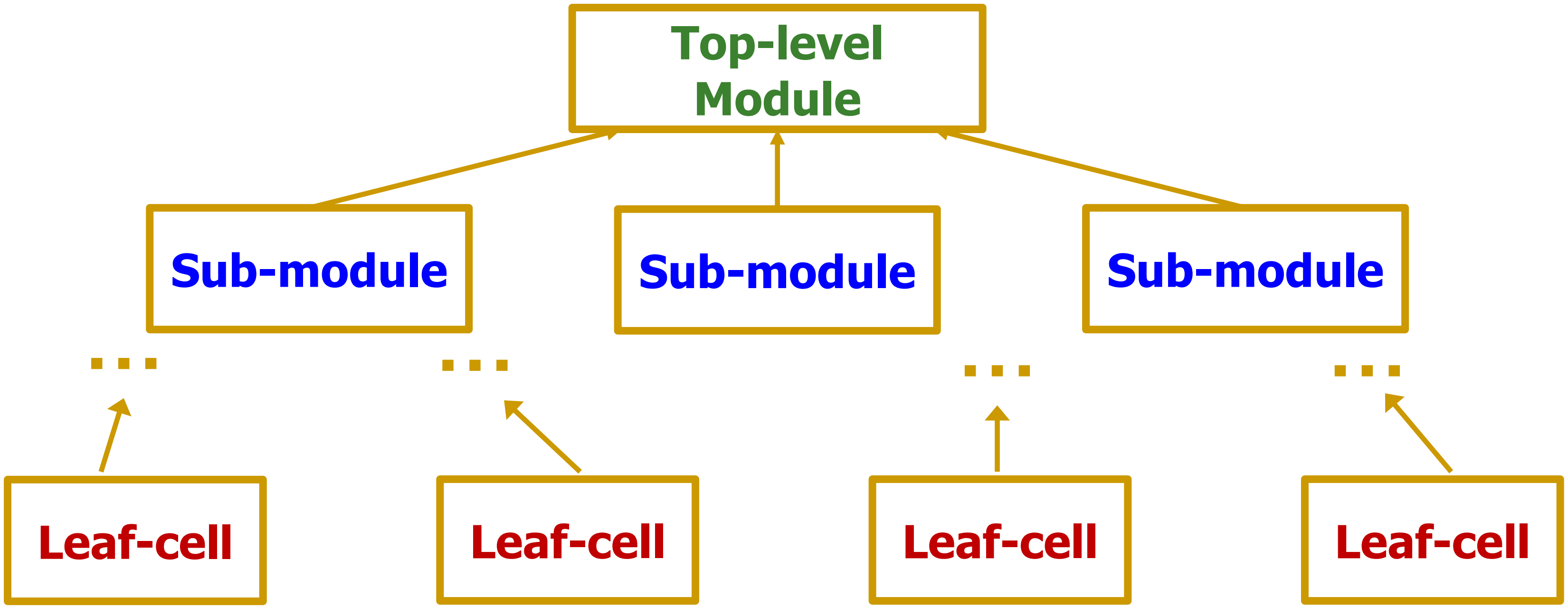
# Top-Down Design Methodology

- 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, primitive cell library elements*)



# Bottom-Up Design Methodology

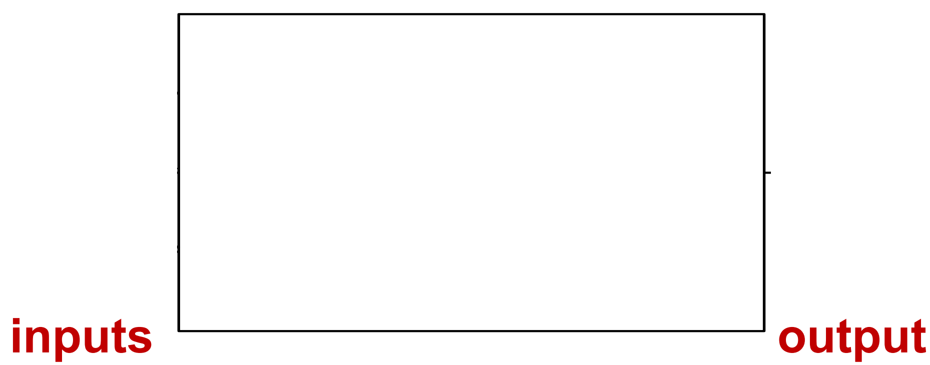
- We first identify the building blocks that are available to us
- Build bigger modules, using these building blocks
- These modules are then used for higher-level modules until we build the top-level module in the design



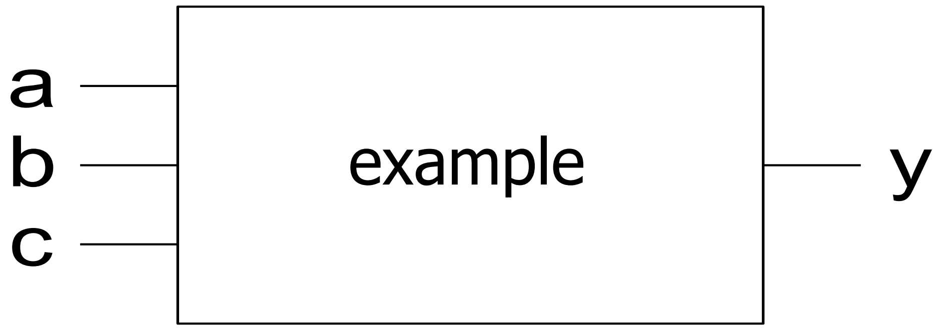
# Defining a Module in Verilog

- 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



# Implementing a Module in Verilog




name of module

Port list (inputs and outputs)

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

// here comes the circuit description

endmodule
```

ports have a declared type

a module definition

# A Question of Style (and Consistency)

- **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

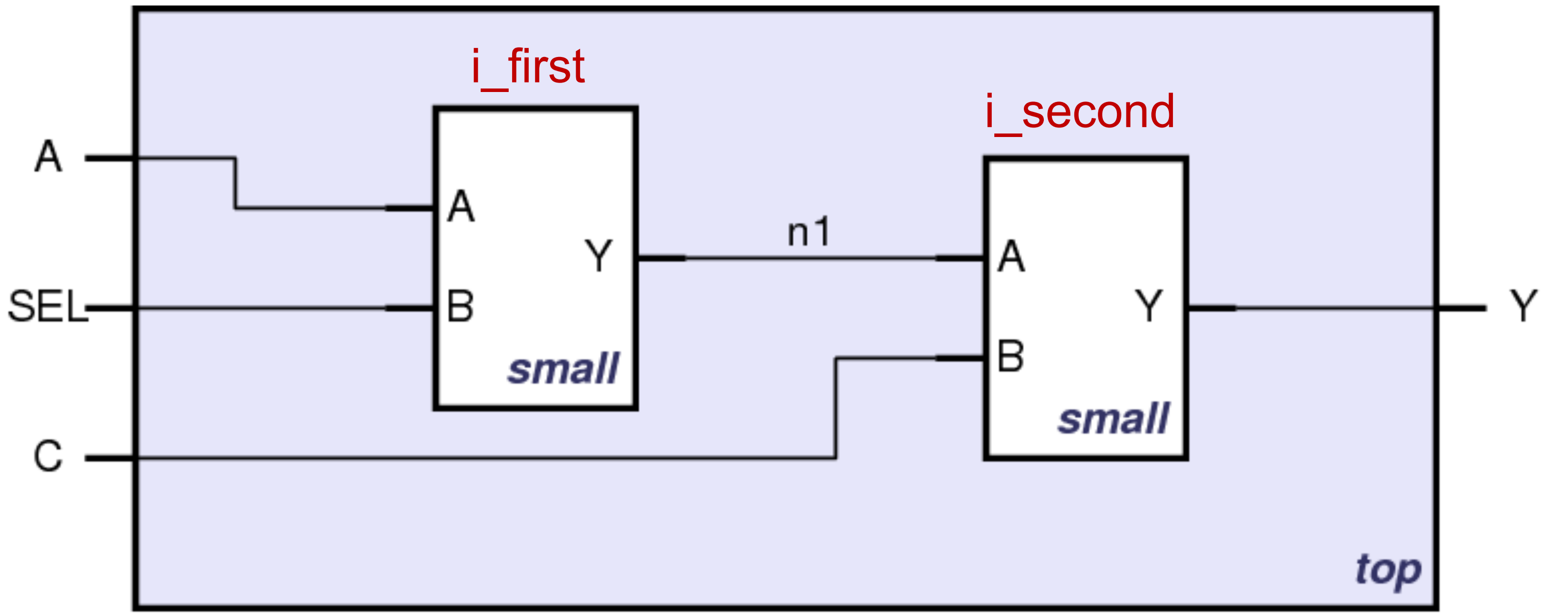
- **Structural (Gate-Level)**
  - The module body contains gate-level description of the circuit
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - … and interconnections between those modules
  - Describes a hierarchy of modules defined as gates

- **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

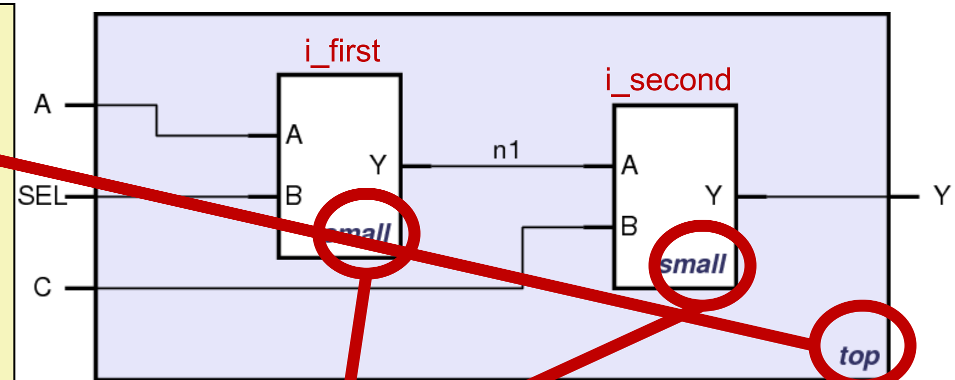



**Schematic of module "top" that is built from two instances of module "small"**

# Structural HDL Example

## Module Definitions in Verilog

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



endmodule
```

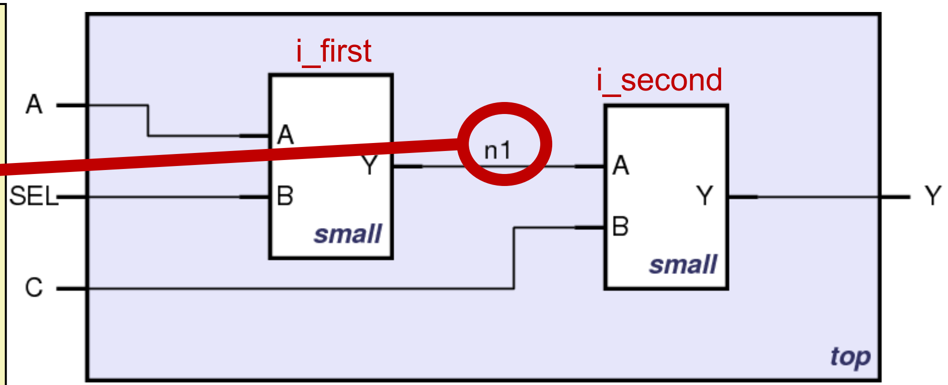



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// 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
```

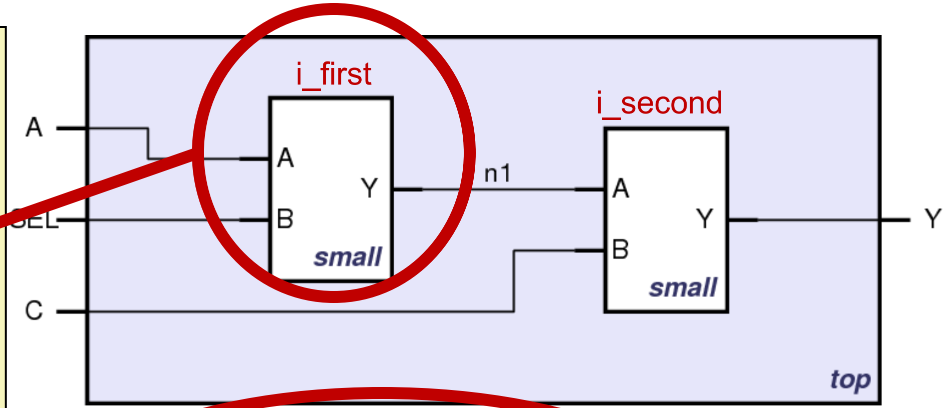



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// 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
```

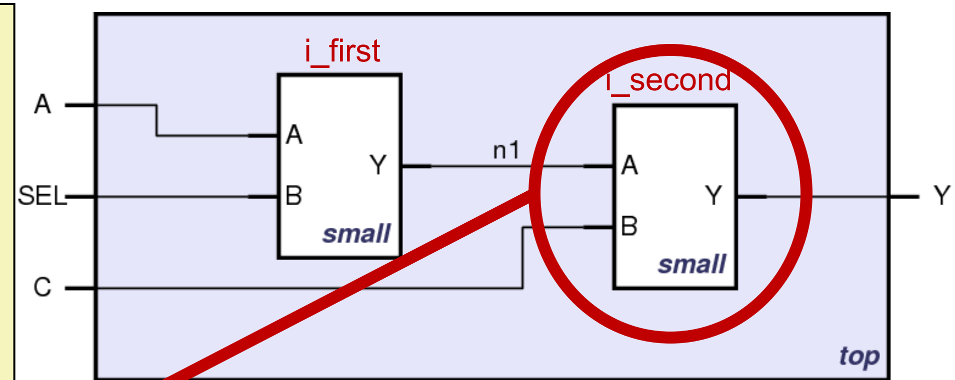



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// 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
```

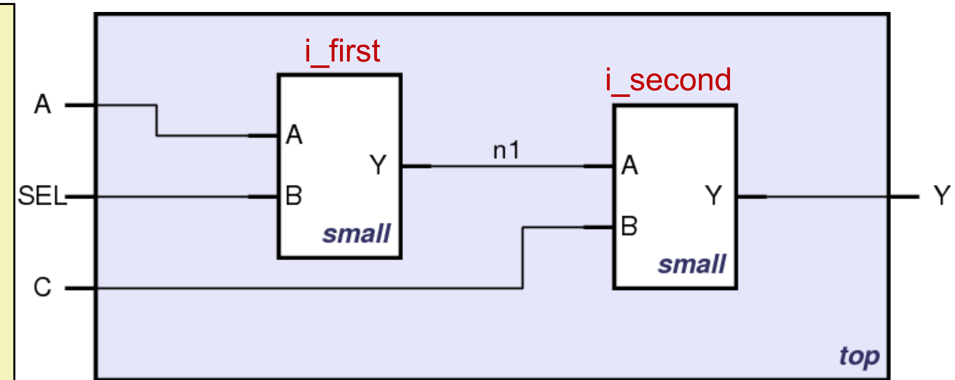



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// 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 short form
small i_first ( A, SEL, n1 );

/* In short form above,
   pin order very important */

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

endmodule
```



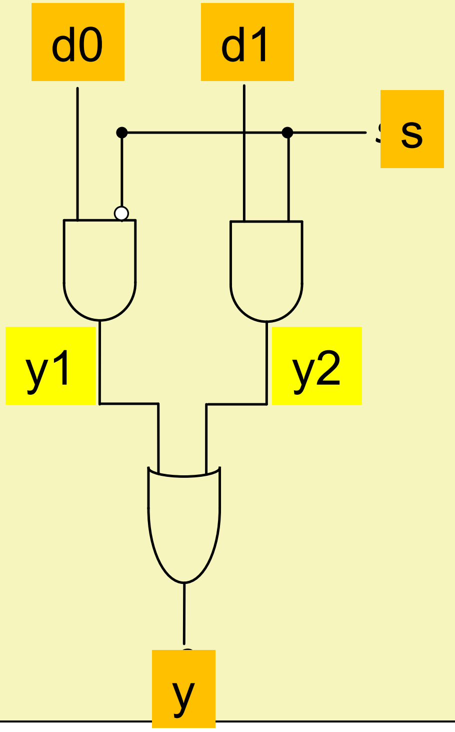

```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

**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

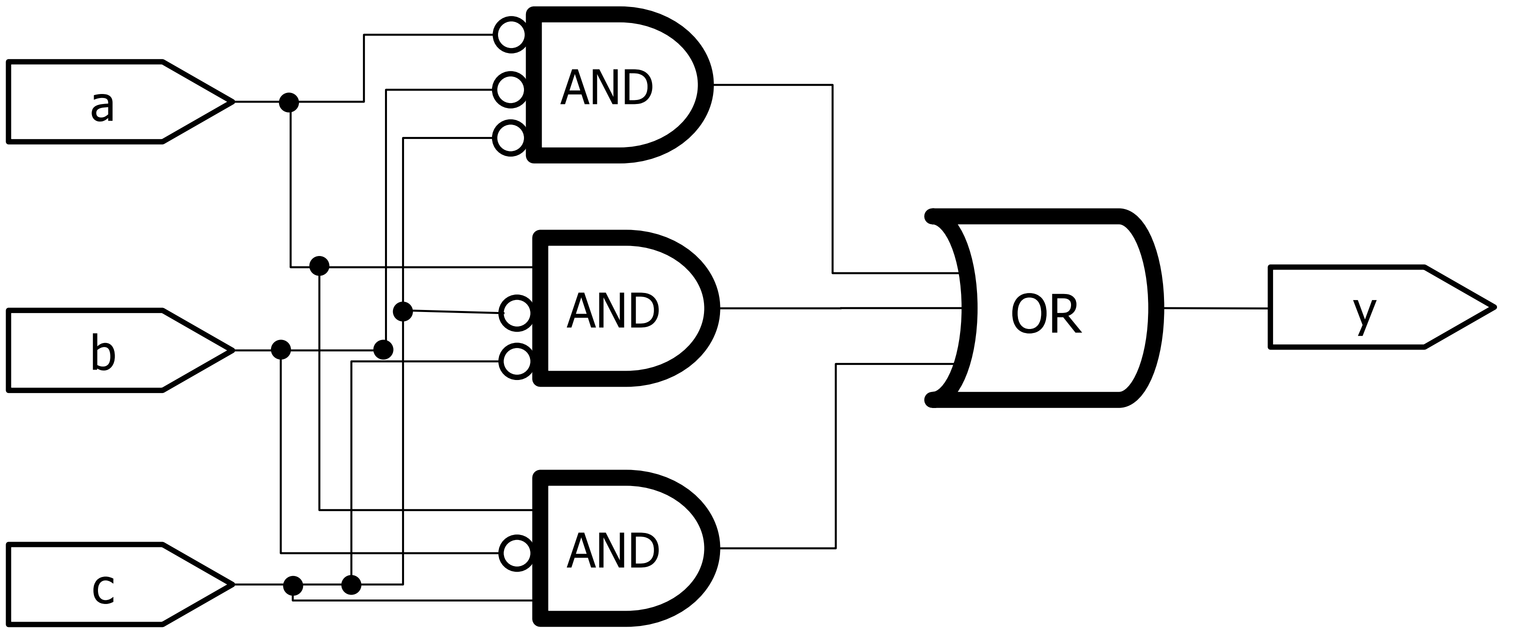
- **Many practical designs use a combination of both**

# 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

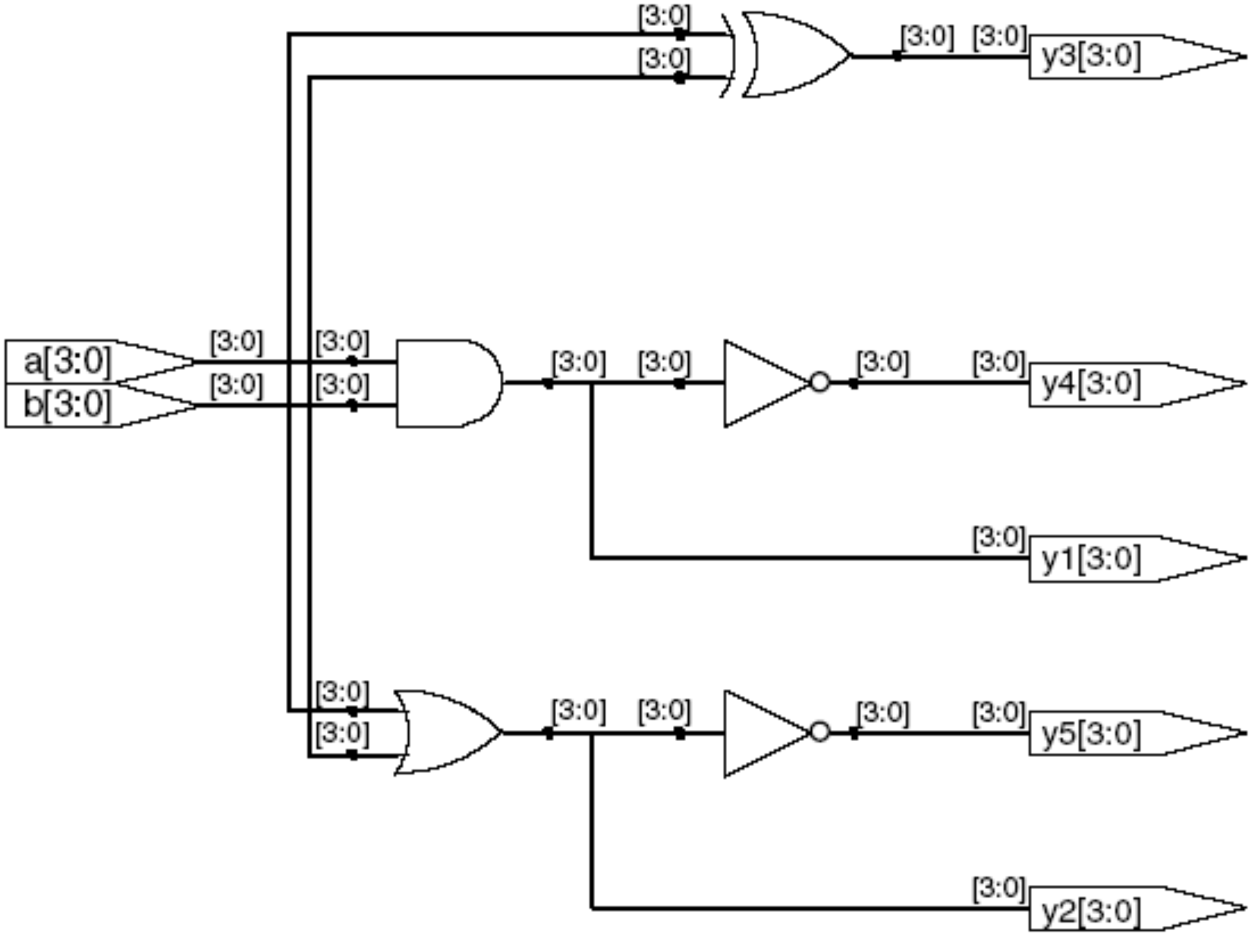
**A behavioral implementation still models a hardware circuit!**



# 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
```

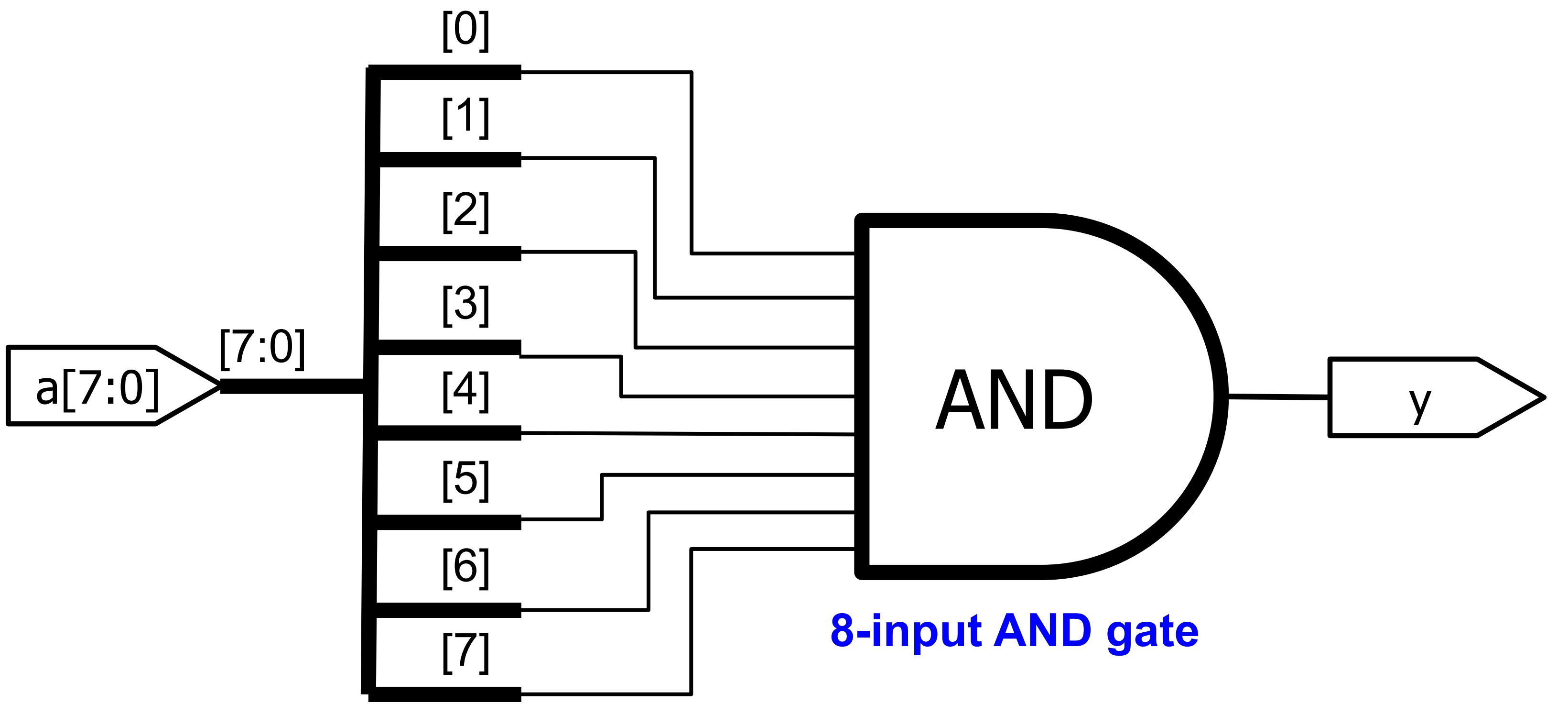
# Bitwise Operators: Schematic View



# 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



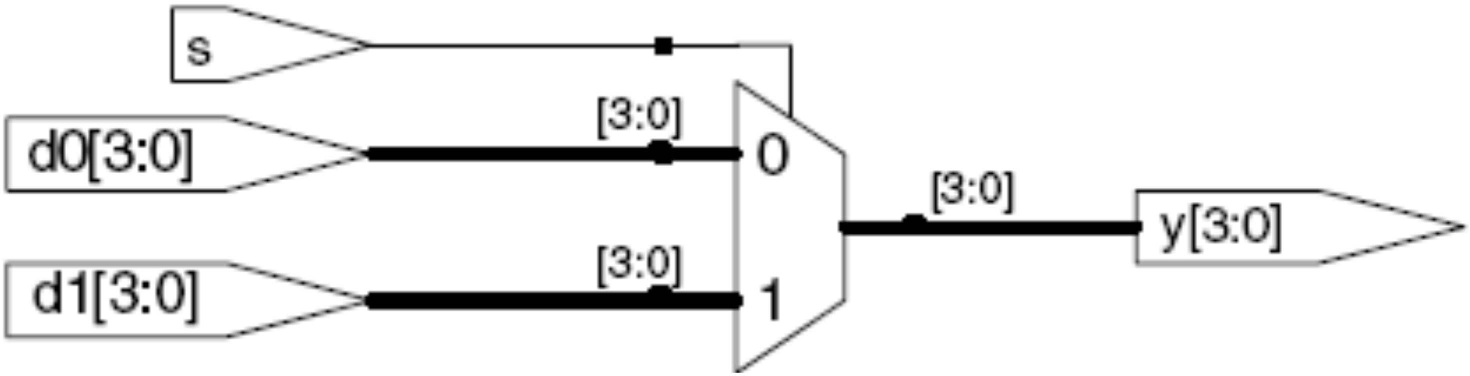

8-input AND gate

# 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

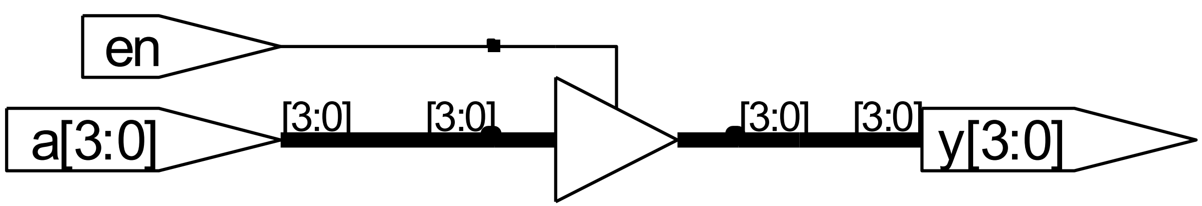
| Verilog | Stored Number | Verilog | Stored Number |
|---|---|---|---|
| 4'b1001 | 1001 | 4'd5 | 0101 |
| 8'b1001 | 0000 1001 | 12'hFA3 | 1111 1010 0011 |
| 8'b0000_1001 | 0000 1001 | 8'o12 | 00 001 010 |
| 8'bxX0X1zZ1 | XX0X 1ZZ1 | 4'h7 | 0111 |
| 'b01 | 0000 .. 0001 | 12'h0 | 0000 0000 0000 |

**32 bits (default)**

# 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

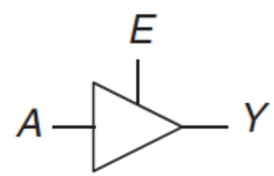
```
module tristate_buffer(input  [3:0] a,
                       input        en,
                       output [3:0] y);

   assign y = en ? a : 4'bz;

endmodule
```



# Recall: 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**

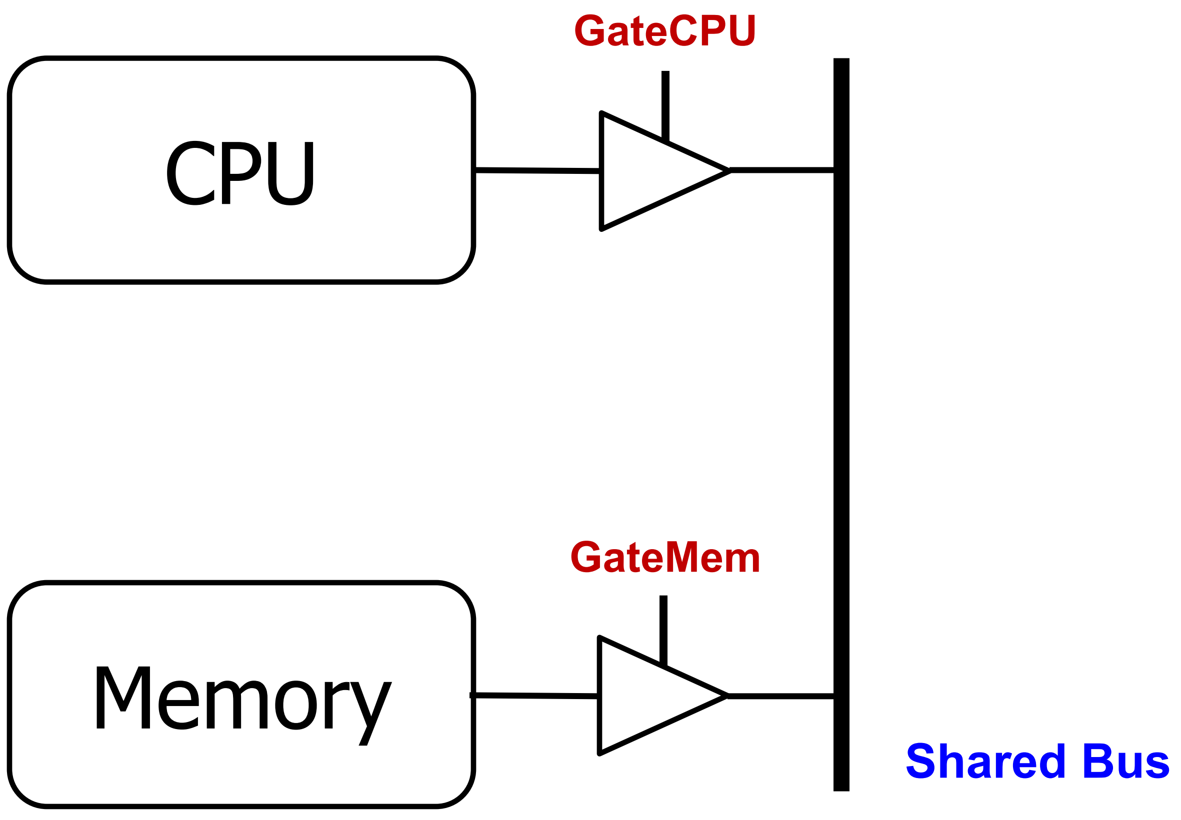
**A tri-state buffer acts like a switch**

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

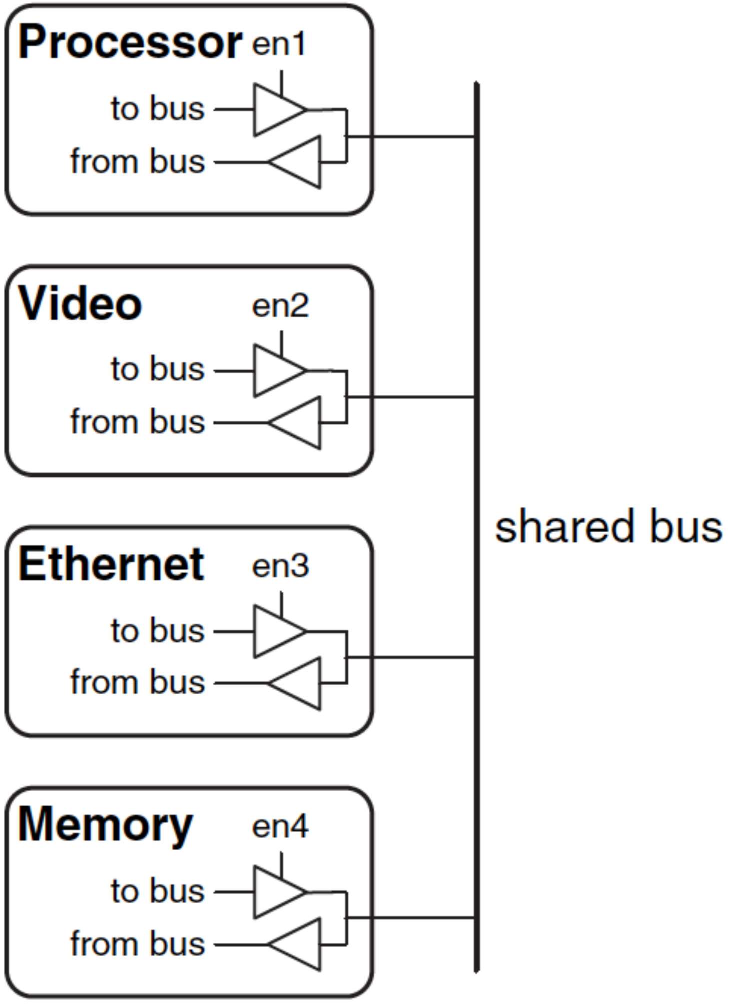
# Recall: Example: Use of Tri-State Buffers

- 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

# Recall: Example Design with Tri-State Buffers



# Recall: 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 |

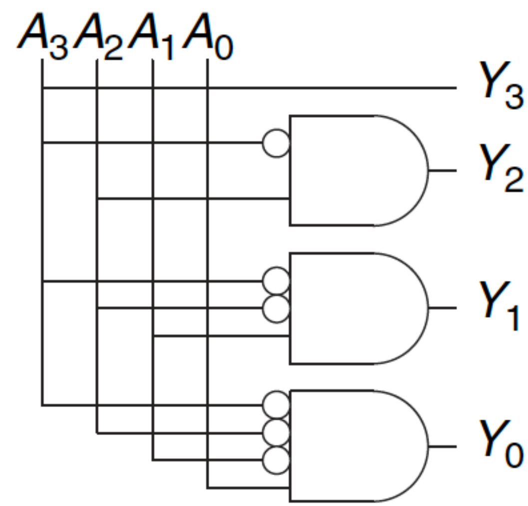
# Recall: Simplified Priority Circuit

- 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?

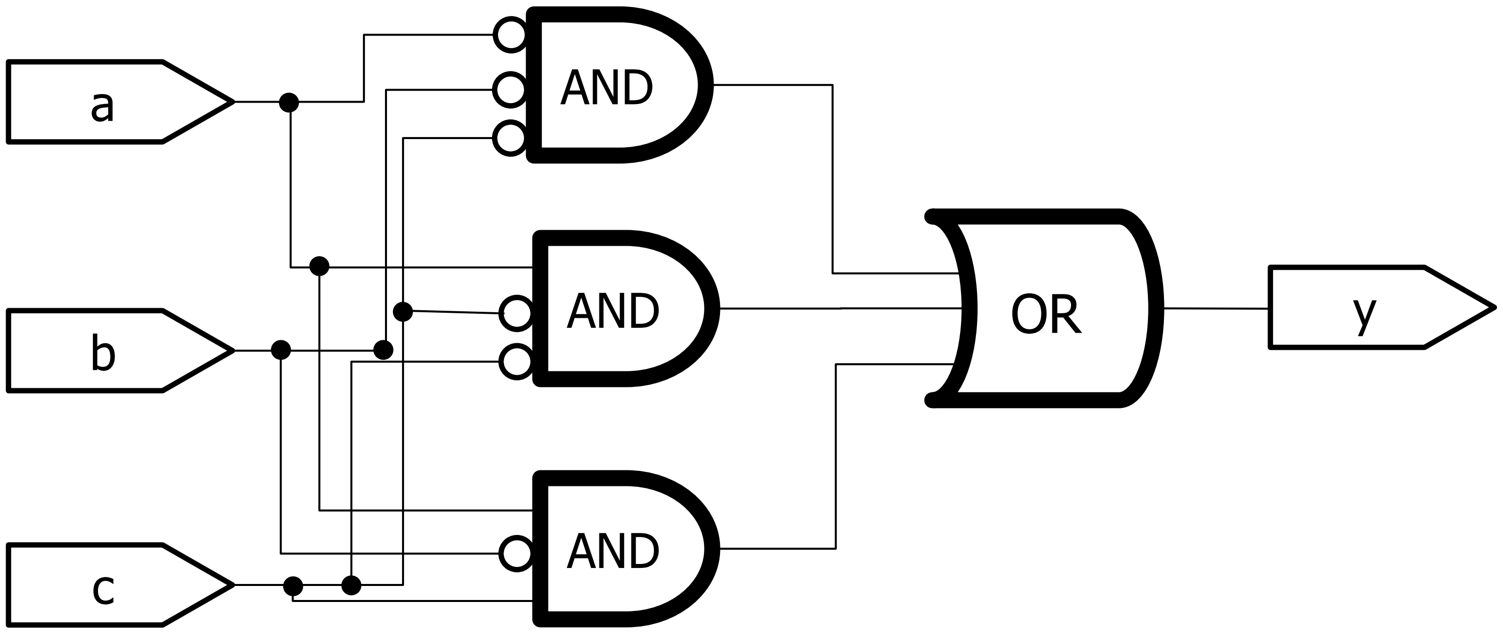
- **Synthesis (i.e., Hardware Synthesis)**
  - Modern tools are able to map ***synthesizable*** *HDL code* into low-level *cell libraries → netlist describing gates and wires*
  - They can perform many optimizations
  - … however they can **not** guarantee that a solution is optimal
    - Mainly due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

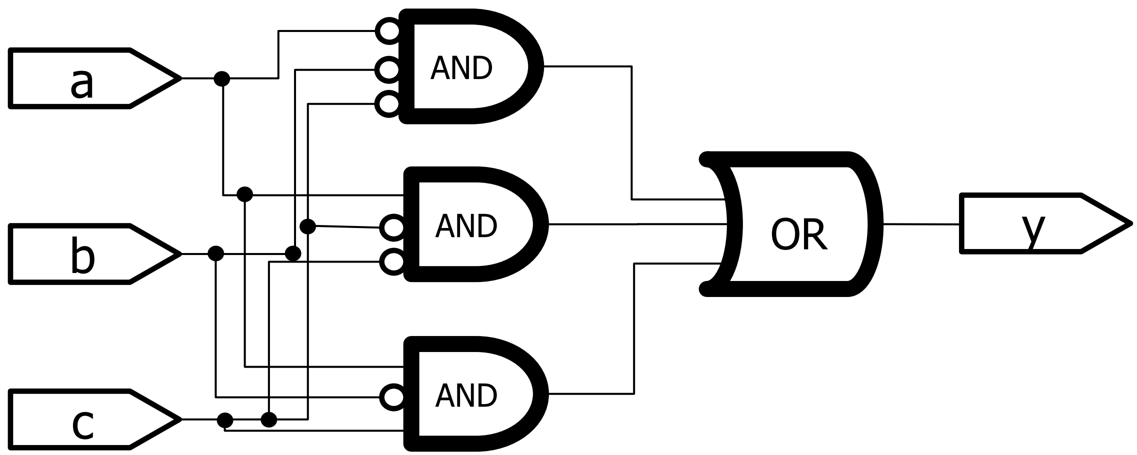
# Recall This “example”

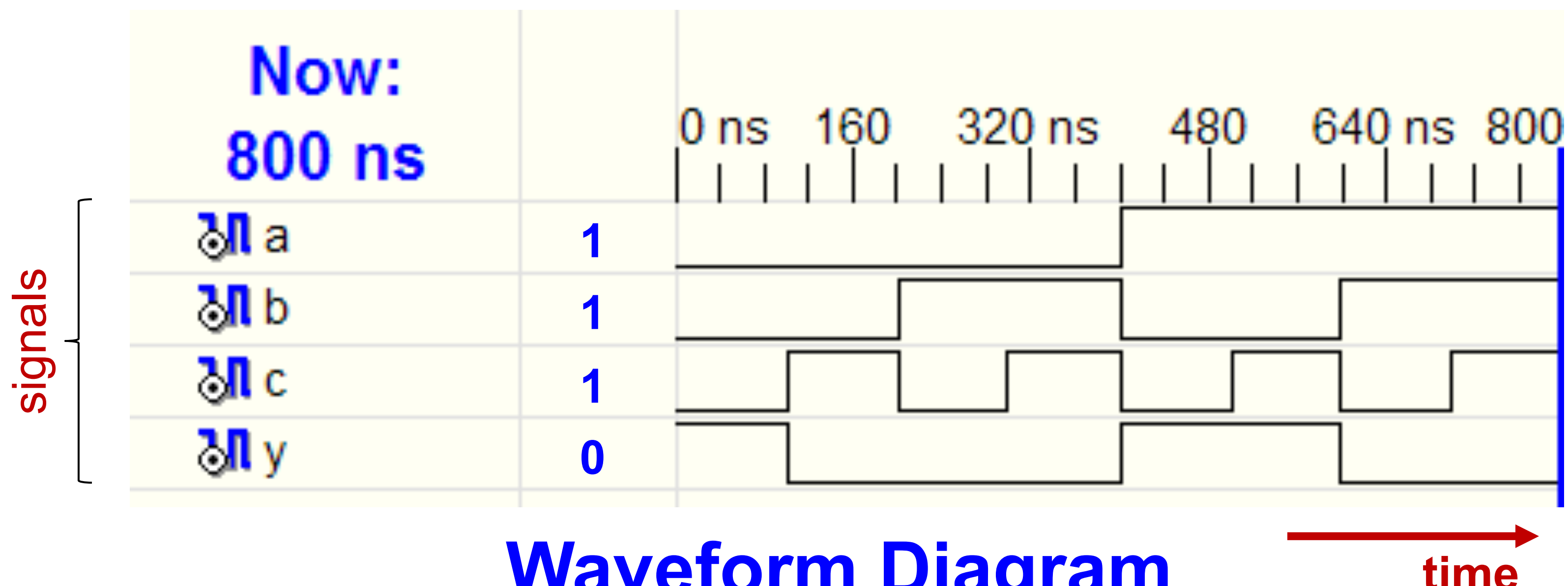
```
module example (a, b, c, y);
       input a;
       input b;
       input c;
       output y;

// 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**

# 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.

**Read H&H Chapter 4.1**

# 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***

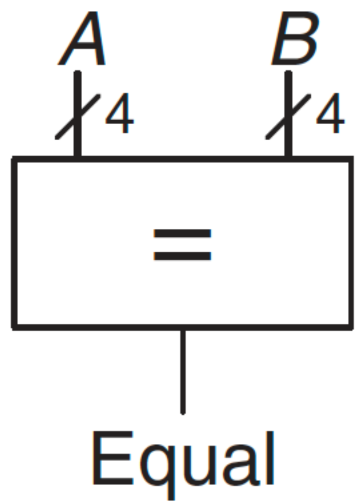
```
module MyAnd (input A, B,
              output Z);

   assign Z = A & B;    // AND

endmodule
```

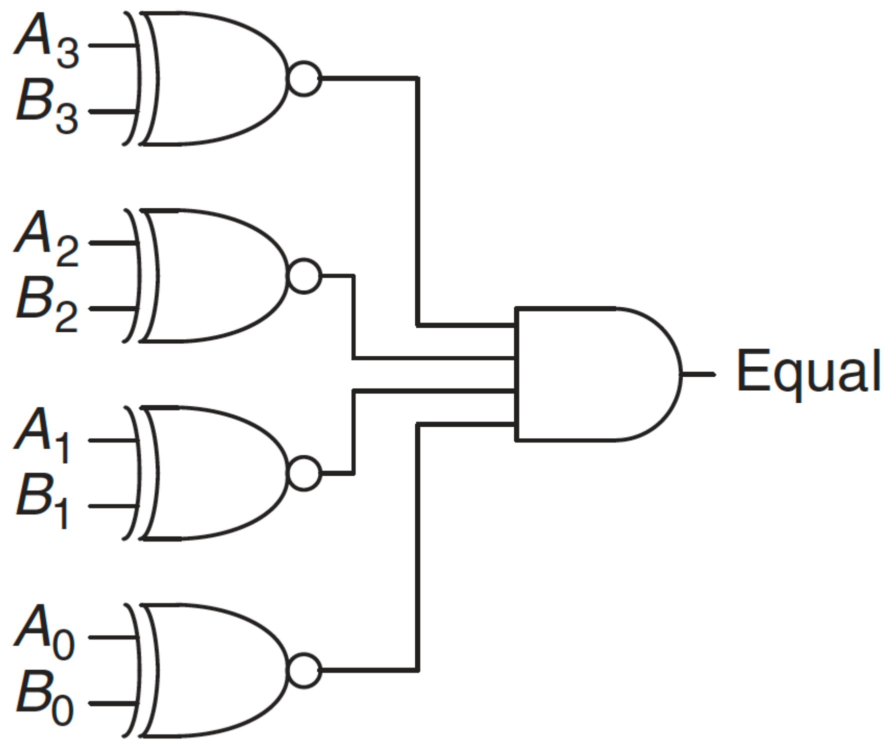
# Gate-Level Implementation

```
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
```

# Recall: Equality Checker (Compare if Equal)

- Checks if two N-input values are exactly the same
- Example: 4-bit Comparator



# 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
```

# 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
```

# 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
```

# Bitwise Operations

```
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
```

# Highest Abstraction Level: Comparing Two Numbers

```
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
```

# 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 on this later today**

# Good Practices

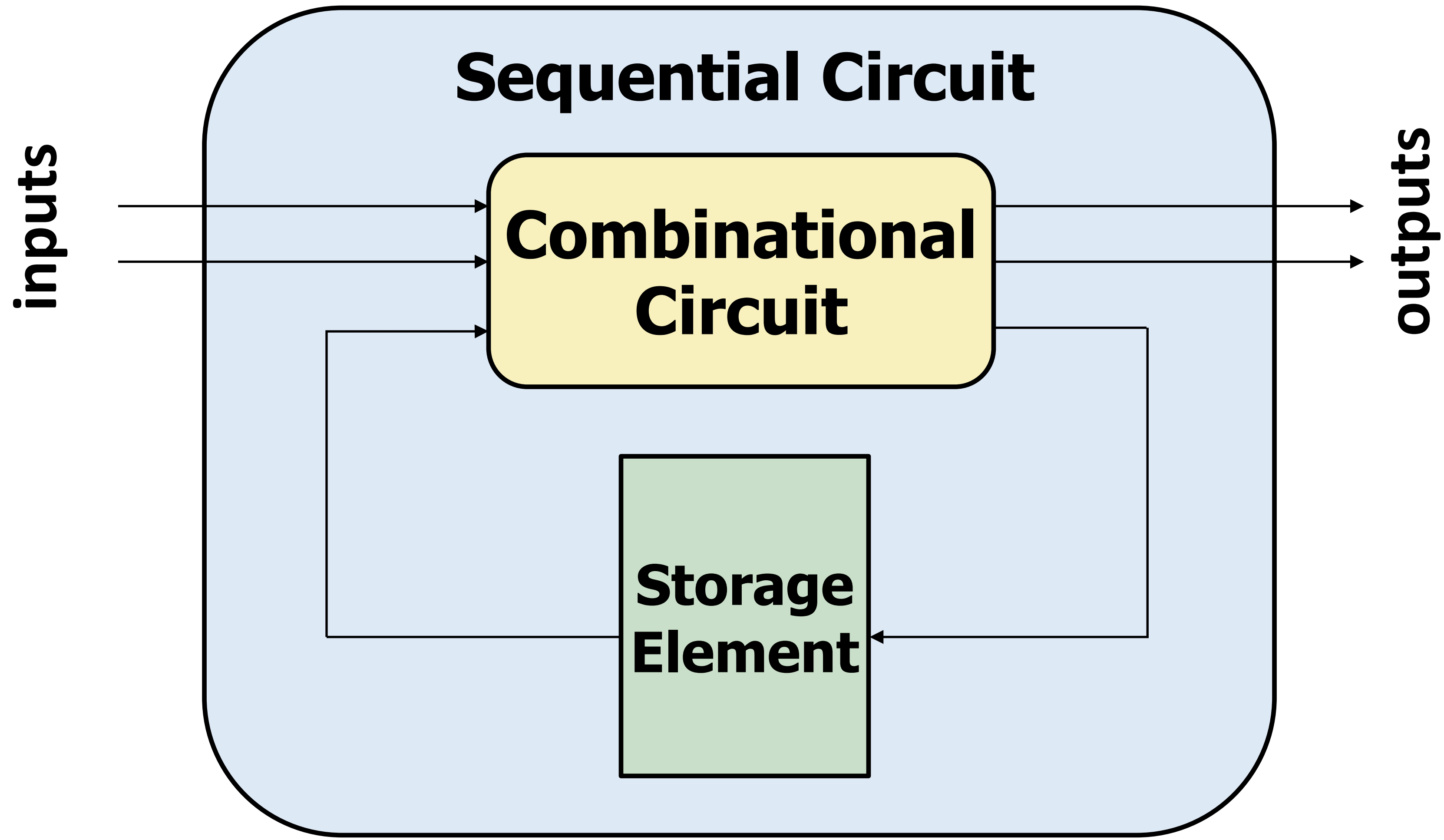
- Develop/use a consistent naming style
- Use MSB to LSB ordering for buses
  - Use "`a[31:0]`", **not** "`a[0:31]`"
- Define one module per file
  - Makes managing your design hierarchy easier
- Use a file name that matches module name
  - e.g., module TryThis is defined in a file called TryThis.v
-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

# Implementing Sequential Logic Using Verilog

# Combinational + Memory = Sequential



# 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`

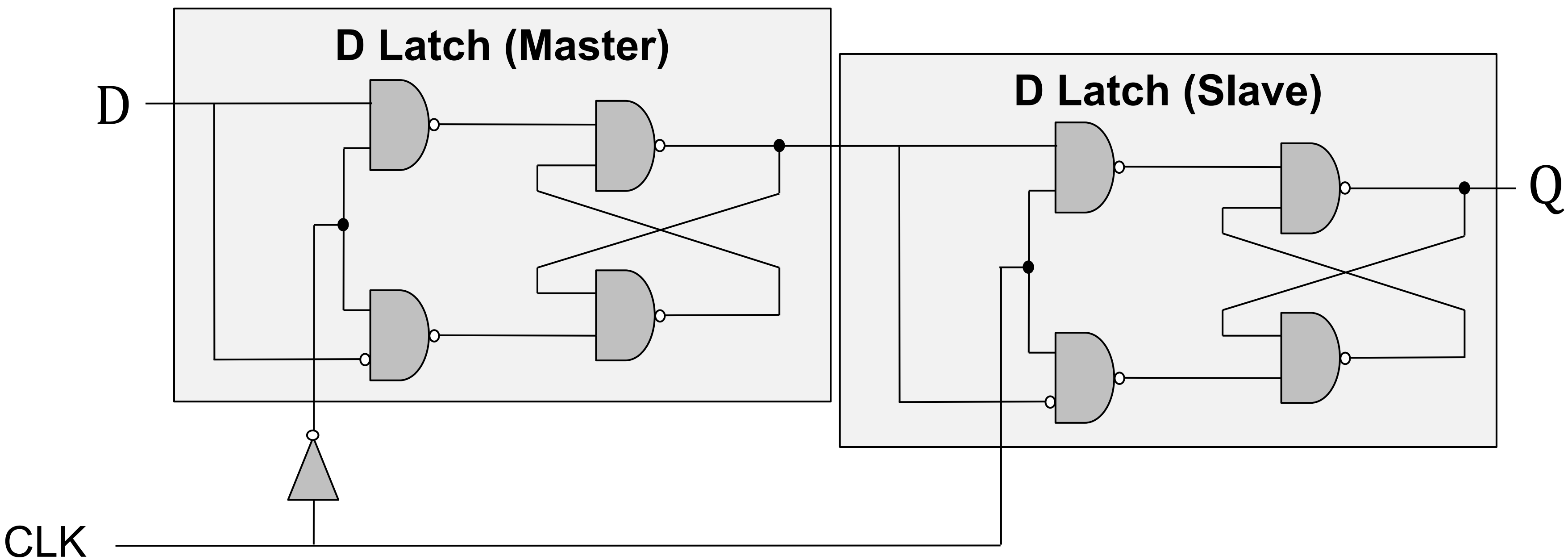
# The "always" Block

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

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

# Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

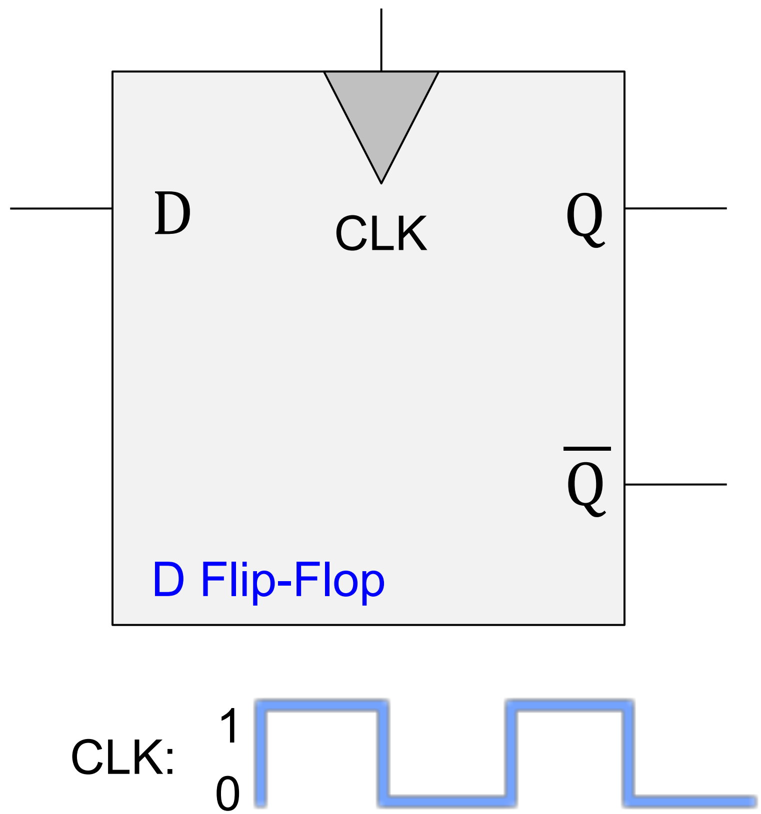



CLK:


- When the clock is low, master propagates **D** to the input of slave (Q unchanged)
- Only when the clock is high, slave latches **D (Q stores D)**
  - At the rising edge of clock (clock going from 0->1), Q gets assigned D

# Recall: The D Flip-Flop

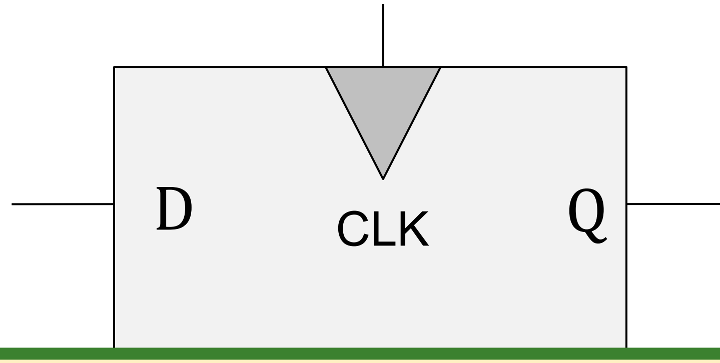
- 1) state change on clock edge, 2) data available for full cycle




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

# Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle




We can use **D Flip-Flops**
to implement the state register

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

# Example: 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
```

- 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

# Example: 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
```

- 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

# Example: 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
```

- 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)
- We will see examples later

# Asynchronous and Synchronous 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
    - Will be discussed in the Timing & Verification Lecture

- **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

# Recall: Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an asynchronous “machine”
  - **State transitions occur when they occur**
  - There is nothing that synchronizes when each state transition must occur

- Most modern computers are synchronous “machines”
  - **State transitions take place after fixed units of time**
  - Controlled in part by a clock, as we will see soon

- These are two different design paradigms, with tradeoffs
  - Synchronous control can be easier to get correct when the system consists of many components and many states
  - Asynchronous control can be more efficient (no clock overheads)

**We will assume synchronous systems in most of this course**

# D Flip-Flop with Asynchronous 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
```

- In this example: two events can trigger the process:
  - A ***rising edge*** on clk
  - A ***falling edge*** on reset

# D Flip-Flop with Asynchronous 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
```

- 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

# D Flip-Flop with **Asynchronous** 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
```

- 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

# D Flip-Flop with **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
```

- The process is sensitive to only clock
  - Reset ***happens only*** when the ***clock rises***. This is a synchronous reset

# D Flip-Flop with Enable and Reset

```
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
```

- 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

# Example: 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
```

# Summary: Sequential Statements So Far

- Sequential statements are within an `always` block

- The sequential block is triggered with a change in the sensitivity list

- Signals assigned within an **always** must be declared as `reg`

- We use `<=` for (non-blocking) assignments and do not use `assign` within the always block.

# Basics of **always** Blocks

```
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
```

You can have as many always blocks as needed

Assignment to the same signal in different always blocks is not allowed!

# 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)

# An **always** Block Does **NOT** Always 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

  1. All right-hand side signals are in the sensitivity list
     - You can use `always @*` for short
  2. 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?

```
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*

**Sequential**

**Sequential**

# The **always** Block is **NOT** Always Practical/Nice

```
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** Block for Case Statements (Handy!)

```
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
```

# Summary: **always** Block

- `if .. else` can only be used in always blocks

- The always block is combinational only if all regs within the block are always assigned to a signal
  - Use the `default` case to make sure you do not forget an unimplemented case, which may otherwise result in a latch

- Use `casex` statement to be able to check for don't cares

# Non-Blocking and Blocking Assignments

## Non-blocking (<=)

```
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

# Why Use (Non)-Blocking Statements

- 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, blocks with non-blocking statements will always evaluate to the same result
  - This may require some additional iterations

# Example: Blocking Assignment

- Assume all inputs are initially '0'

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

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

# The Same Example: Non-Blocking Assignment

- Assume all inputs are initially '0'

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

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

# The Same Example: Non-Blocking Assignment

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

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

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

# We Covered Until This Point in the Lecture

# 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 (Cont.)

- Use `always @ (*)` and blocking assignments (`=`) to model more complicated combinational logic.

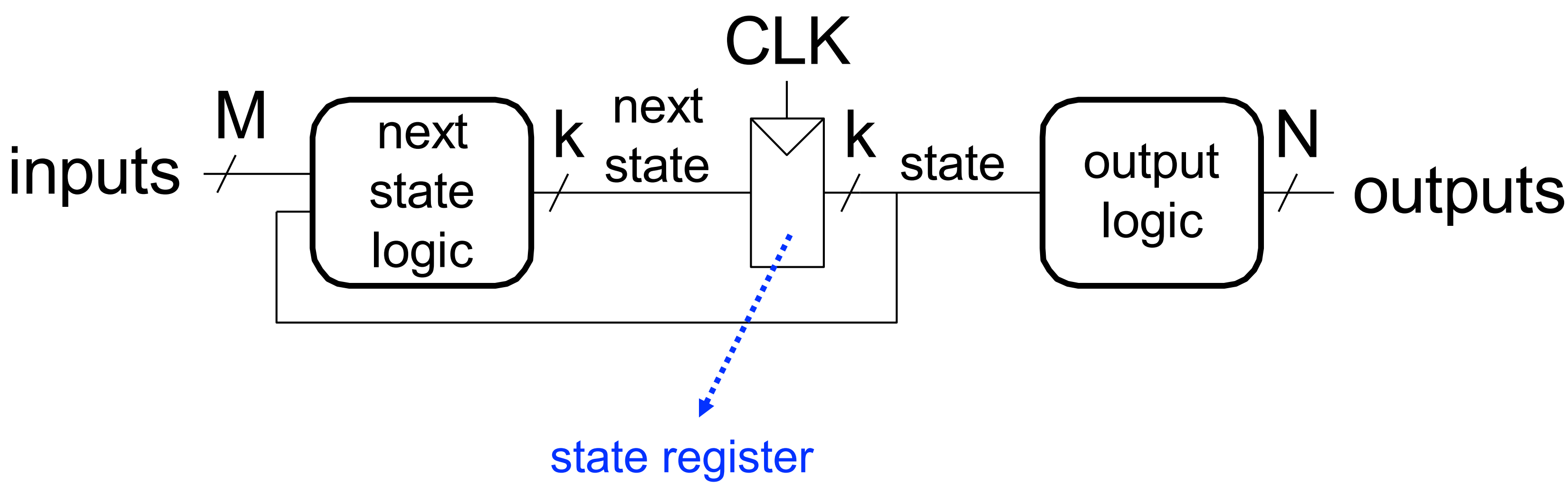
- You cannot make assignments to the same signal in more than one always block or in a *continuous assignment*

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

always @ (*)
      a = c;
```

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

assign a = c;
```

# Finite State Machines (FSMs)

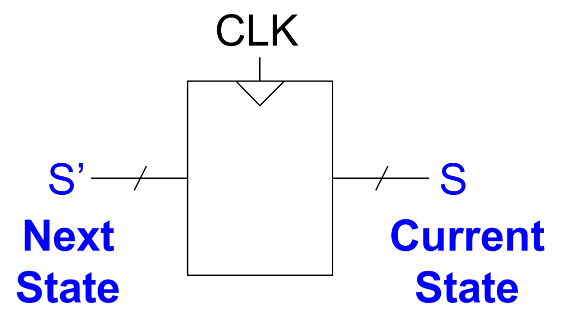
- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic



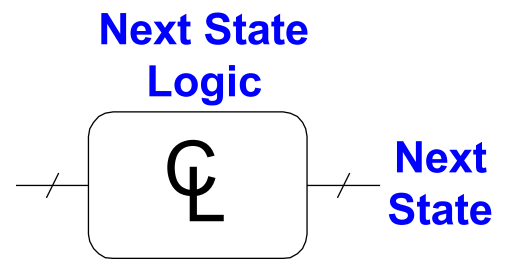

At the beginning of the clock cycle, next state is latched into the state register

# Finite State Machines (FSMs) Consist of:

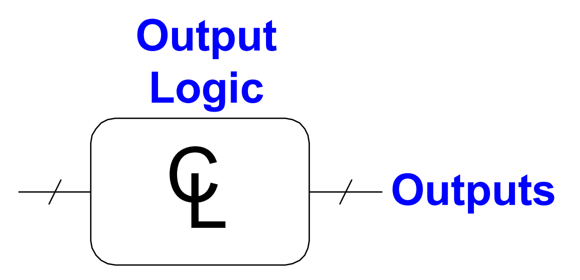
- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge



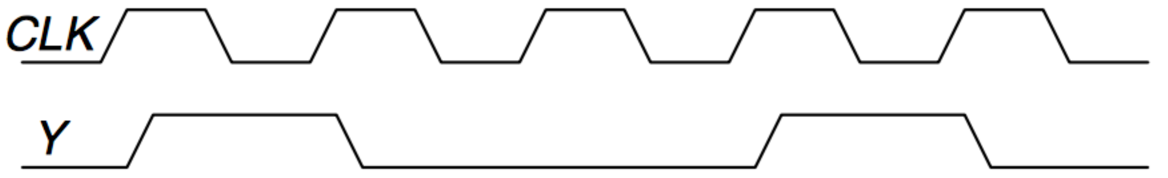

- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be



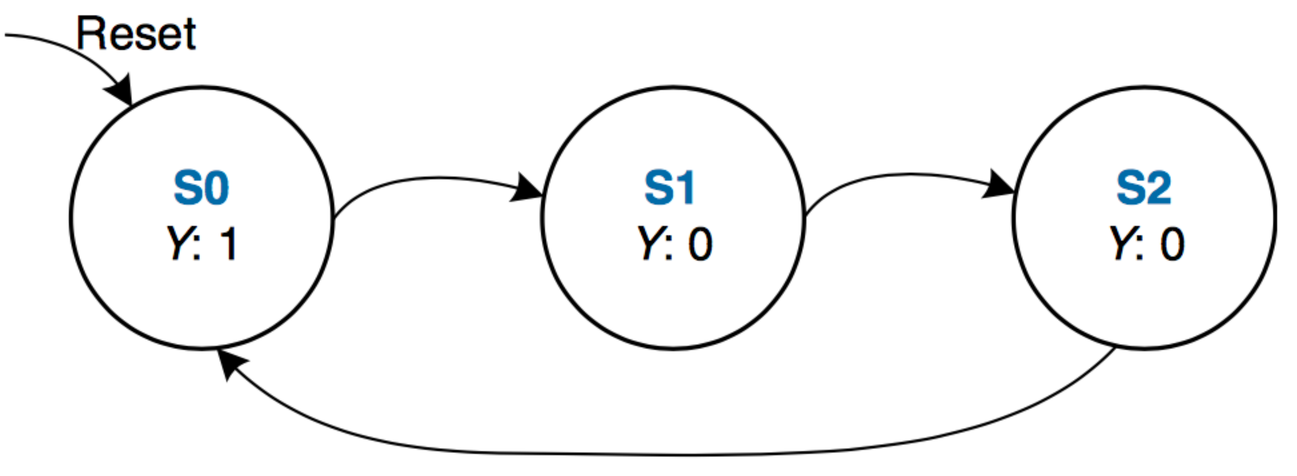

  - Output logic
    - Generates the outputs



# FSM Example 1: 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***.

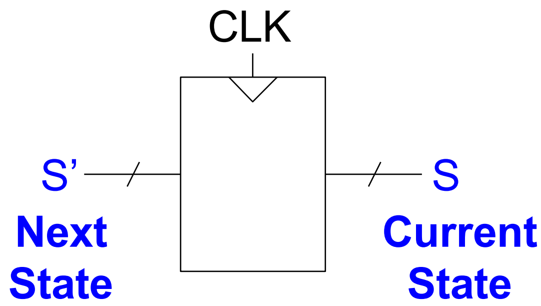


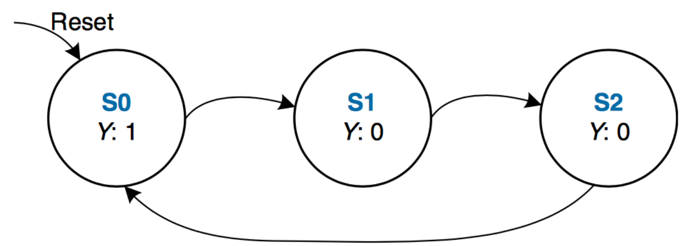
# Implementing FSM Example 1: Definitions

```
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

# Implementing FSM Example 1: State Register

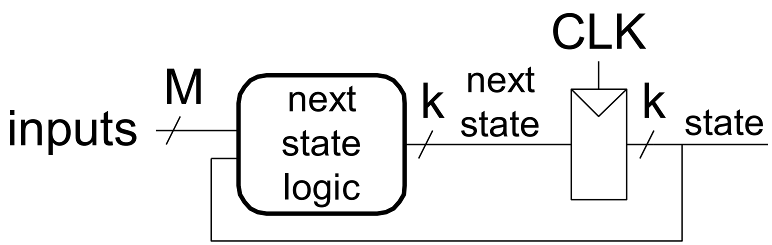


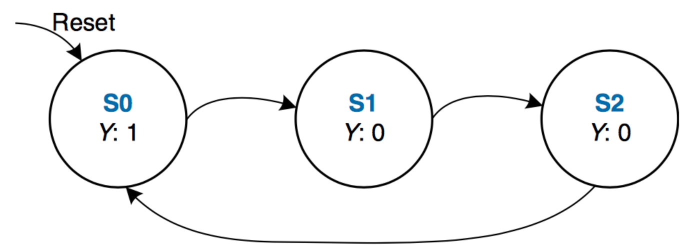




```
// state register
   always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
```

- 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)

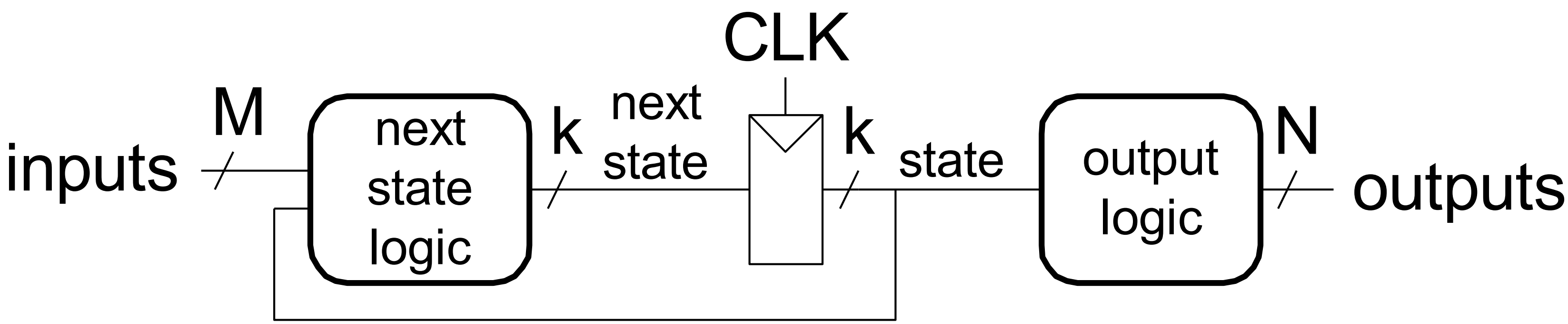
# Implementing FSM Example 1: Next State Logic

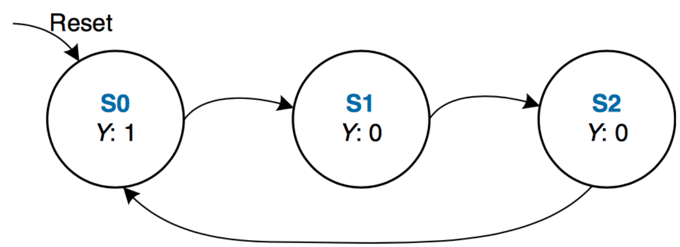



```
// next state logic
  always @ (*)
     case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
     endcase
```

# Implementing FSM Example 1: Output Logic




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

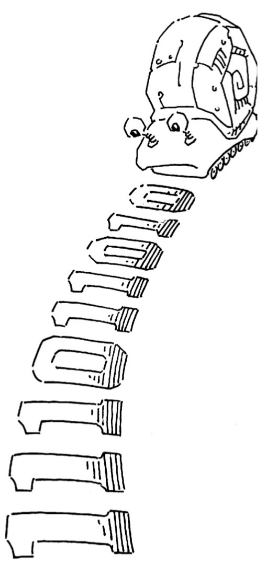
- In this example, output depends only on state
  - **Moore type FSM**

# Implementation of FSM Example 1

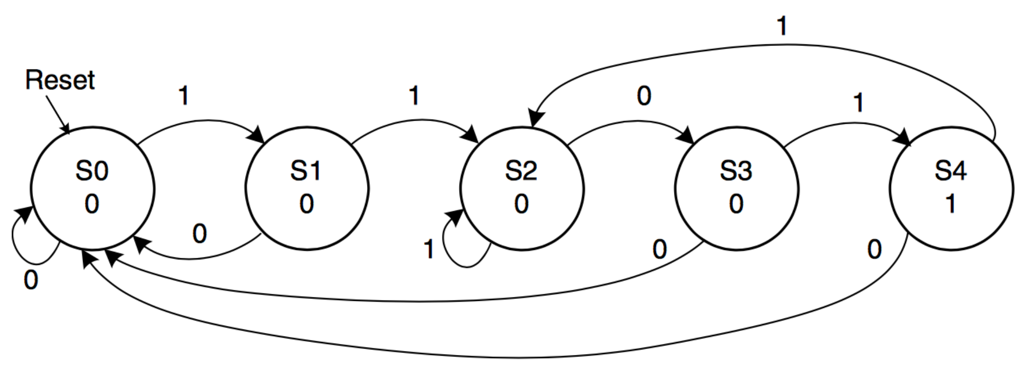
```
module divideby3FSM (input clk, input reset, output q);
   reg  [1:0] state, nextstate;

   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 @ (*)                          // next state logic
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);            // output logic
endmodule
```

# FSM Example 2: Smiling Snail

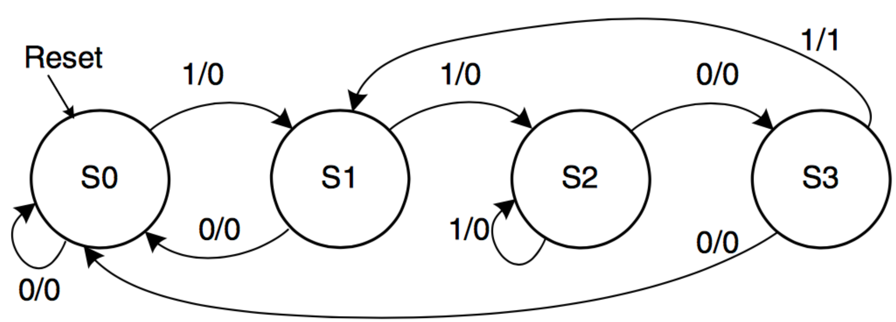
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it
- The snail smiles whenever the last four digits it has crawled over are 1101
- Design Moore and Mealy FSMs of the snail's brain



**Moore**

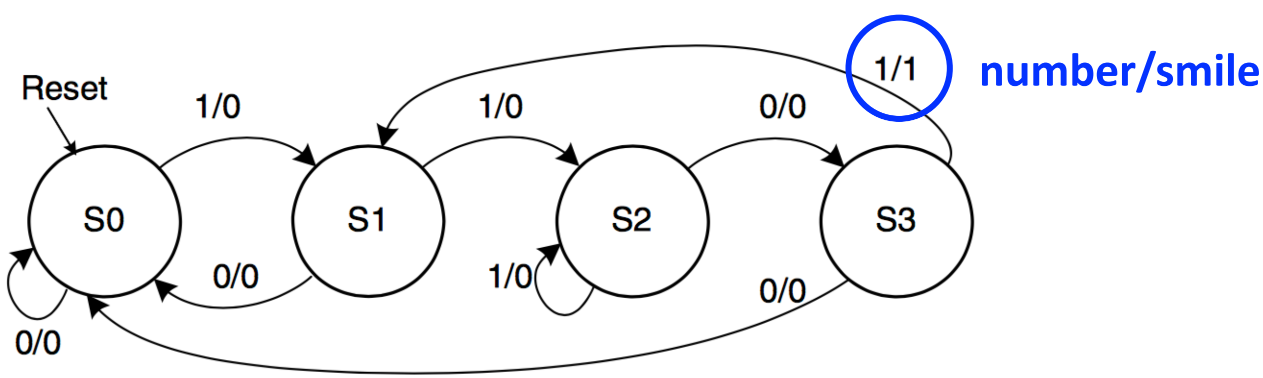



**Mealy**



# Implementing FSM Example 2: Definitions

```
module SmilingSnail (input clk,
                     input reset,
                     input number,
                     output smile);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
   parameter S3 = 2’b11;
```



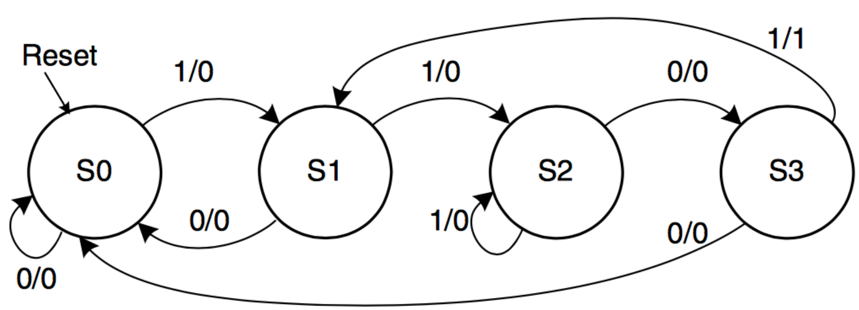
# Implementing FSM Example 2: State Register

```
// 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 ‘1’ (active-high)

# Implementing FSM Example 2: Next State Logic

```
// next state logic
  always @ (*)
     case (state)
        S0: if (number) nextstate = S1;
            else   nextstate = S0;
        S1: if (number) nextstate = S2;
            else   nextstate = S0;
        S2: if (number) nextstate = S2;
            else   nextstate = S3;
        S3: if (number) nextstate = S1;
            else   nextstate = S0;
        default:   nextstate = S0;
     endcase
```



# Implementing FSM Example 2: Output Logic

```
// output logic
   assign smile = (number & state == S3);
```

- In this example, output depends on state and input
  - **Mealy type FSM**
- We used a simple combinational assignment

# Implementation of FSM Example 2

```
module SmilingSnail (input clk,
                     input reset,
                     input number,
                     output smile);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
   parameter S3 = 2'b11;

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

```
always @ (*) // next state logic
      case (state)
         S0: if (number)
                  nextstate = S1;
             else nextstate = S0;
         S1: if (number)
                  nextstate = S2;
             else nextstate = S0;
         S2: if (number)
                  nextstate = S2;
             else nextstate = S3;
         S3: if (number)
                  nextstate = S1;
             else nextstate = S0;
         default: nextstate = S0;
      endcase
   // output logic
assign smile = (number & state==S3);

endmodule
```

# What Did We Learn?

- Basics of describing sequential circuits in Verilog

- The always statement
  - Needed for describing memorizing elements (flip-flops, latches)
  - Can also be used to describe combinational circuits

- Blocking vs Non-blocking statements
  - = assigns the value immediately
  - <= assigns the value at the end of the block

- Describing FSMs in Verilog
  - Next state logic
  - State assignment
  - Output logic

# Next Lecture:

Timing and Verification

# Digital Design & Computer Arch.

## Lecture 7: Hardware Description Languages and Verilog

Prof. Onur Mutlu

ETH Zürich

Spring 2022

17 March 2022