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

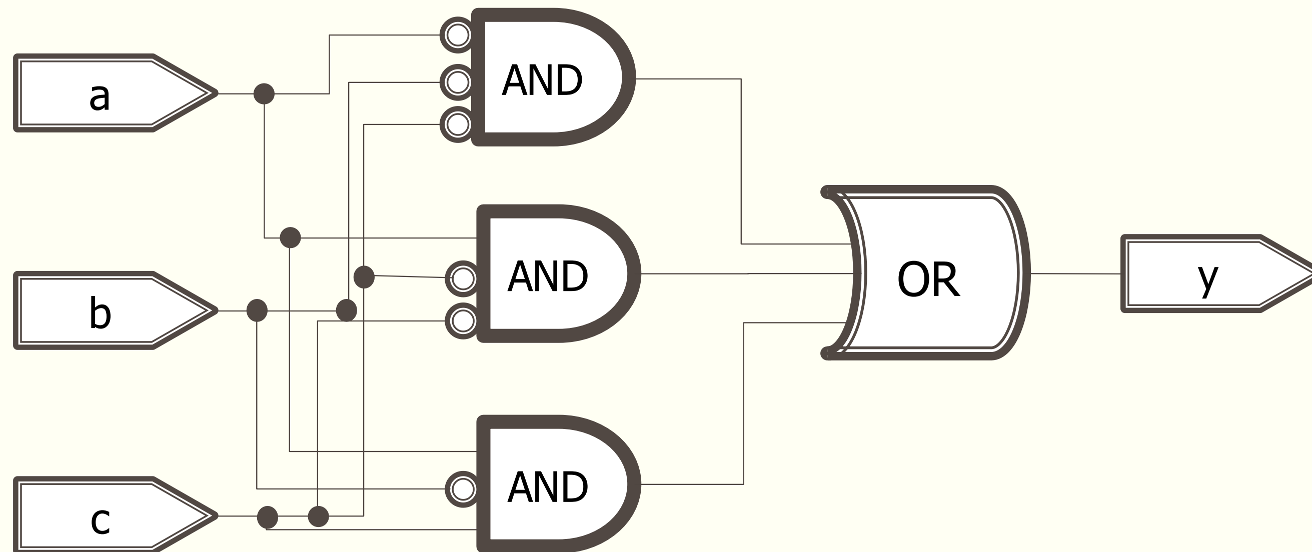
- **Many practical designs use a combination of both**

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

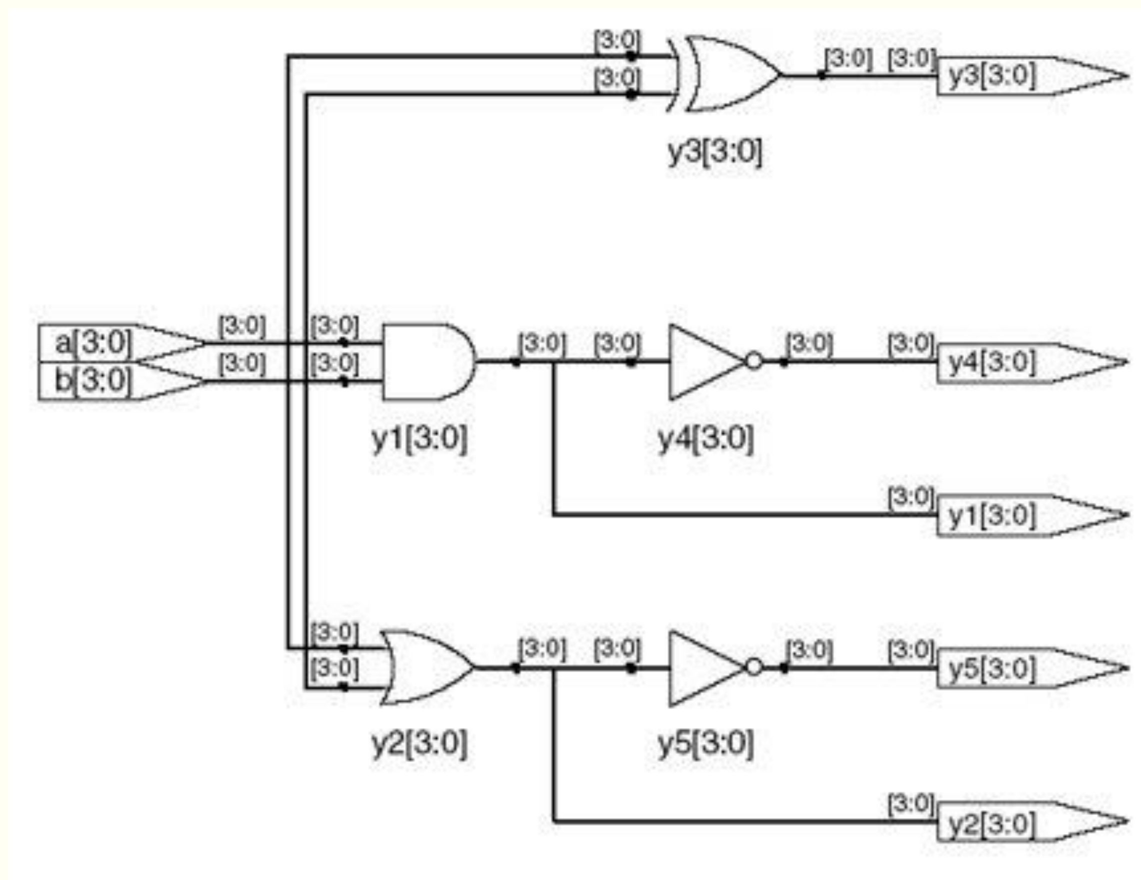
A behavioral implementation still models a hardware circuit!



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
```

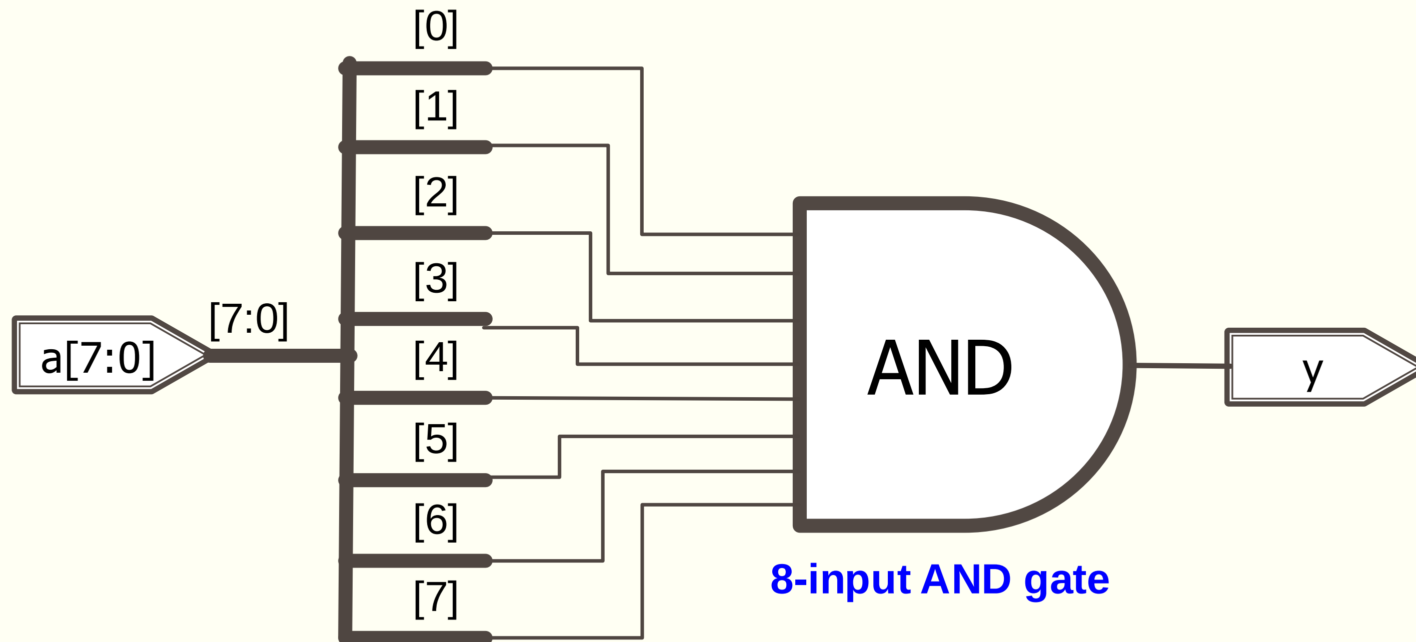
Bitwise Operators: Schematic View



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

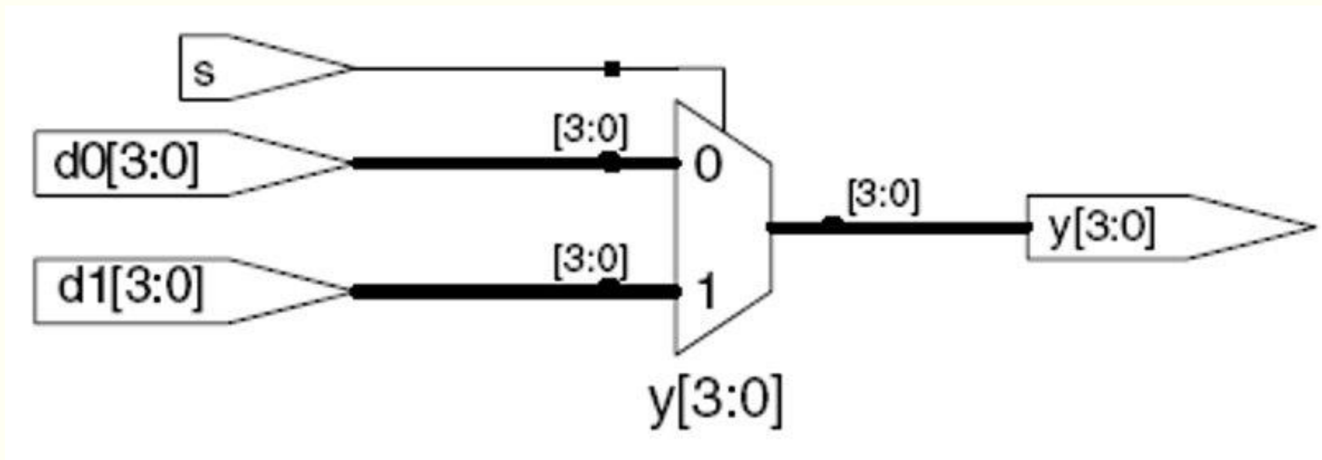


Conditional Assignment in Behavioral Verilog

```
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 Assignment

```
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

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

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

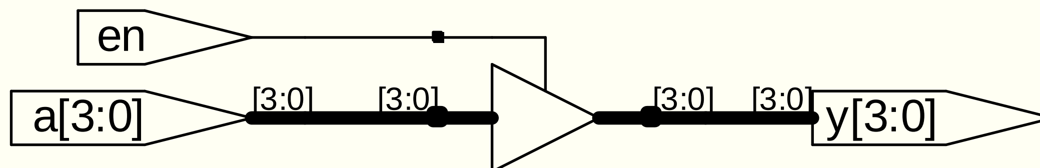
Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1010 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001	12'h0	0000 0000 0000

32 bits (default)

Floating Signals (Z)

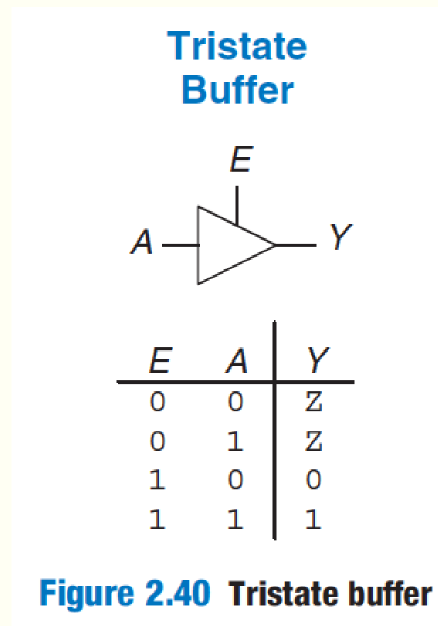
- Floating signal: Signal that is not driven by any circuit
  - Open circuit, floating wire
- Also known as: high impedance, hi-Z, tri-stated signals

```
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



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

Figure 2.40 **Tristate buffer**

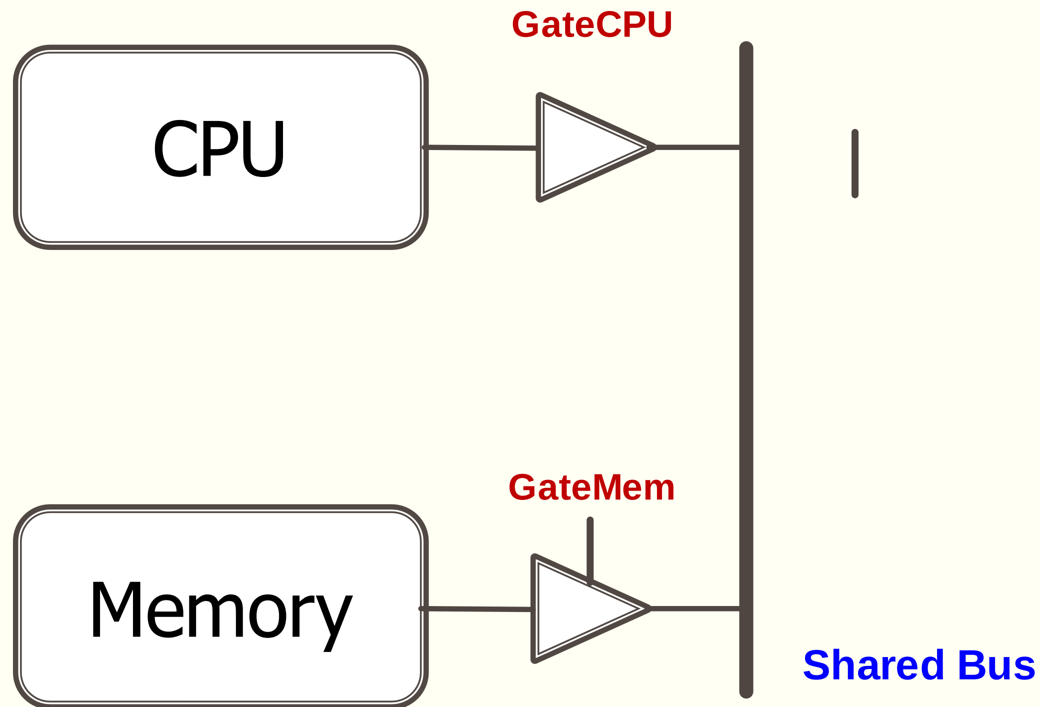
A tri-state buffer acts like a switch

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

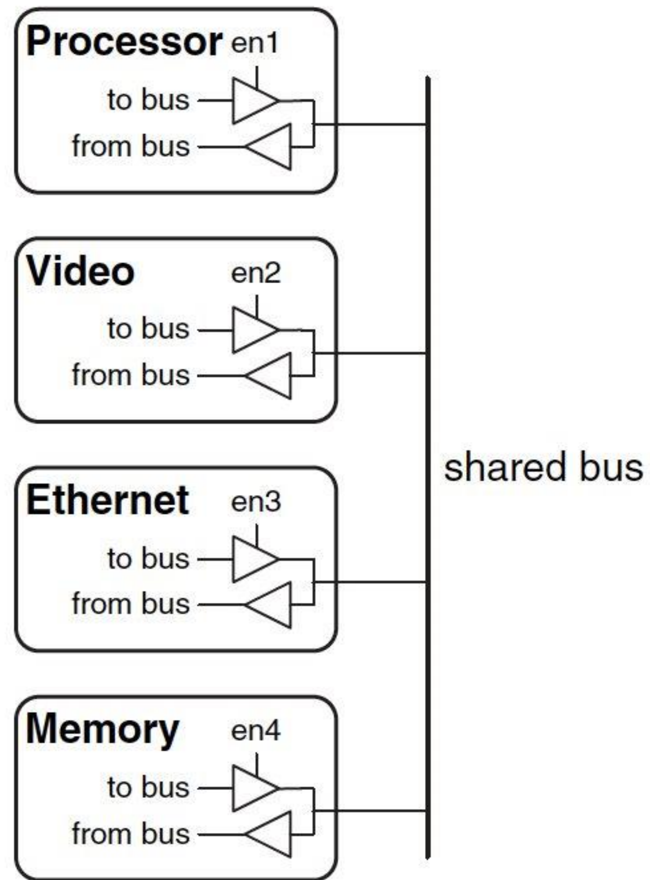
Example: Use of Tri-State Buffer

- 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 Buffer



Another Example



What Happen with HDL Code?

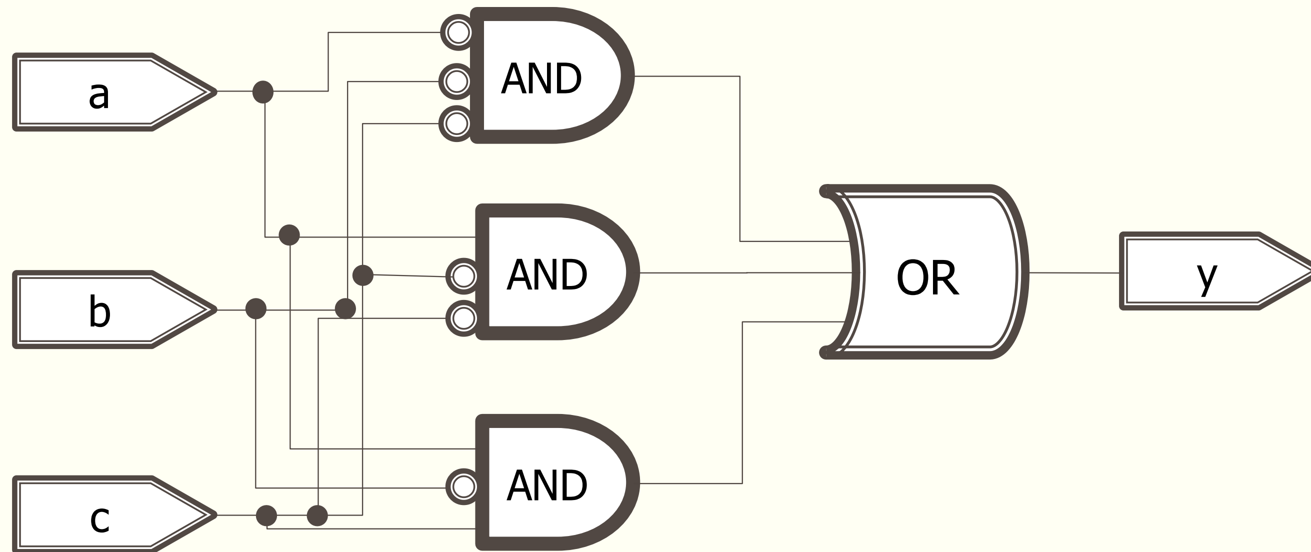
- **Synthesis (i.e., Hardware Synthesis)**
  - Modern tools are able to map ***synthesizable*** *HDL code* into low-level *cell libraries* → *netlist describing gates and wires*
  - They can perform many optimizations
  - … however they can **not** guarantee that a solution is optimal
    - Mainly due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

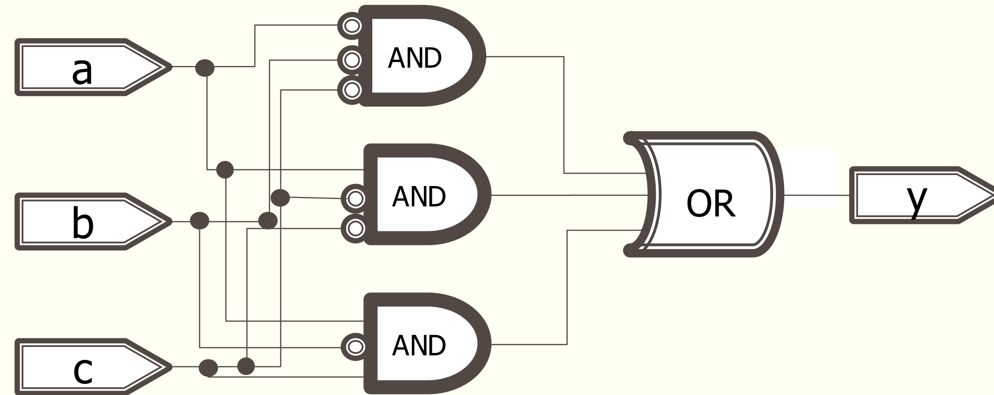
Recall "This Example"

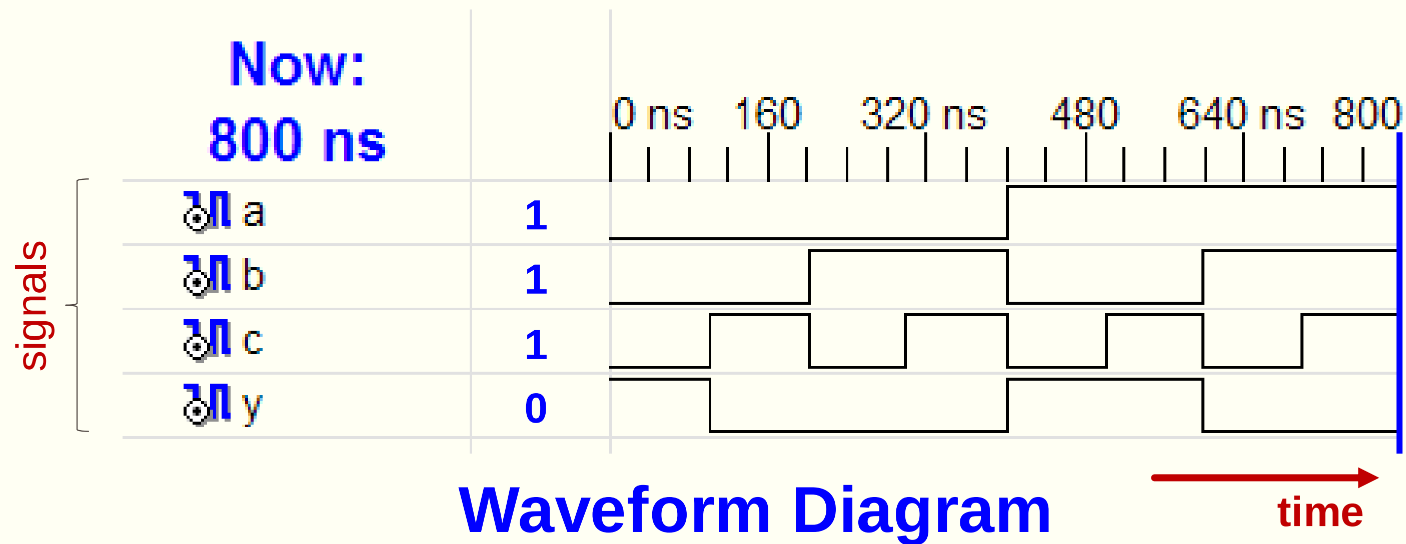
```
module example (a, b, c, y);
       input a;
       input b;
       input c;
       output y;

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

endmodule
```

Synthesizing the "Example"



Simulating the "example"





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
    - Easier automated optimization (especially for low-level tools)

  - At a high-level of abstraction
    - Better readability
    - More difficult automated optimization (large search space)

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

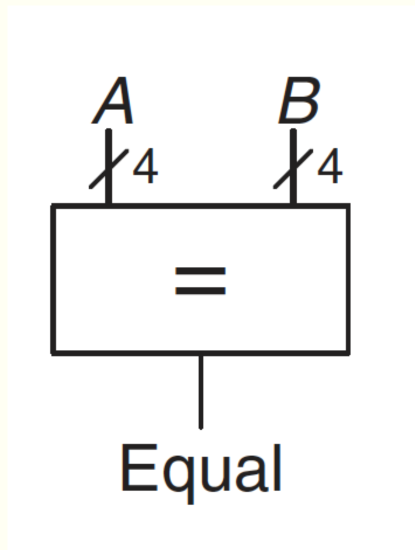
```
module MyXnor (input A, B,
               output Z);

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

endmodule
```

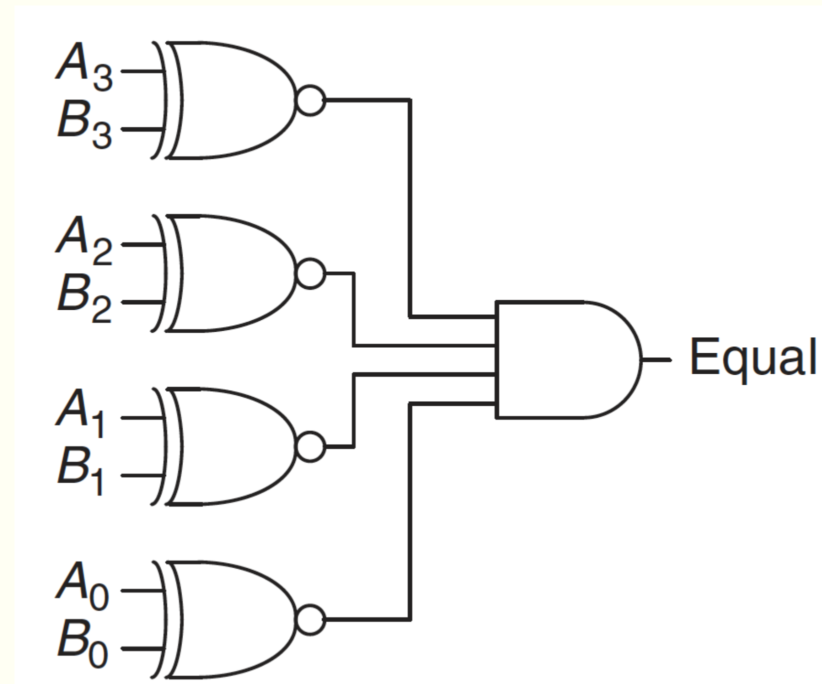
A 2-input AND gate

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

   assign Z = A & B;    // AND

endmodule
```

Equality Checker (Compare if Equal)

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





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
```

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 Module

```
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 soon

Good Practises

- 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