

Overview

- ◆ Last lecture
 - Incompletely specified functions
- ◆ Today (guest lecture by Benjamin Ylvisaker)
 - Verilog
 - ⇒ Structural constructs
 - ⇒ Describing combinational circuits

Ways of specifying circuits

◆ Schematics

- Structural description
- Describe circuit as interconnected elements
 - ⇒ Build complex circuits using hierarchy
 - ⇒ Large circuits are unreadable

◆ HDLs

- Hardware description languages
 - ⇒ **Not** programming languages
 - ⇒ Parallel languages tailored to digital design
- Synthesize code to produce a circuit

Hardware description languages (HDLs)

- ◆ Abel (~1983)
 - Developed by Data-I/O
 - Targeted to PLDs
 - Limited capabilities (can do state machines)
- ◆ Verilog (~1985)
 - Developed by Gateway (now part of Cadence)
 - **Syntax** similar to C
 - Moved to public domain in 1990
- ◆ VHDL (~1987)
 - DoD sponsored
 - **Syntax** similar to Ada

Verilog versus VHDL

- ◆ Both “IEEE standard” languages
- ◆ Most tools support both
- ◆ Verilog is “simpler”
 - Less syntax, fewer constructs
- ◆ VHDL is more structured
 - Can be better for large, complex systems
 - Better modularization

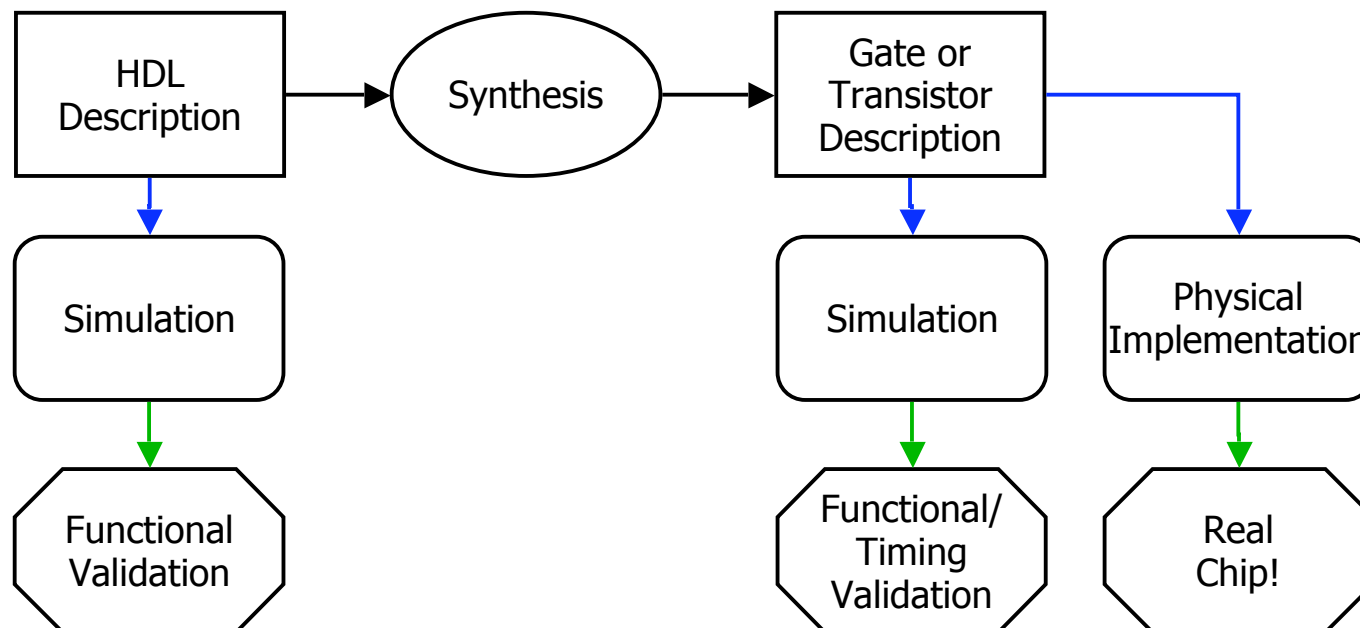
Simulation and synthesis

- ◆ Simulation

- “Execute” a design to verify correctness

- ◆ Synthesis

- Generate a physical implementation from HDL code



Simulation and synthesis (con't)

◆ Simulation

- Models what a circuit does
 - ⇒ Multiply is "*", ignoring implementation options
- Can include static timing
- Allows you to test design options

◆ Synthesis

- Converts your code to a netlist
 - ⇒ Can simulate synthesized design
- Tools map your netlist to hardware

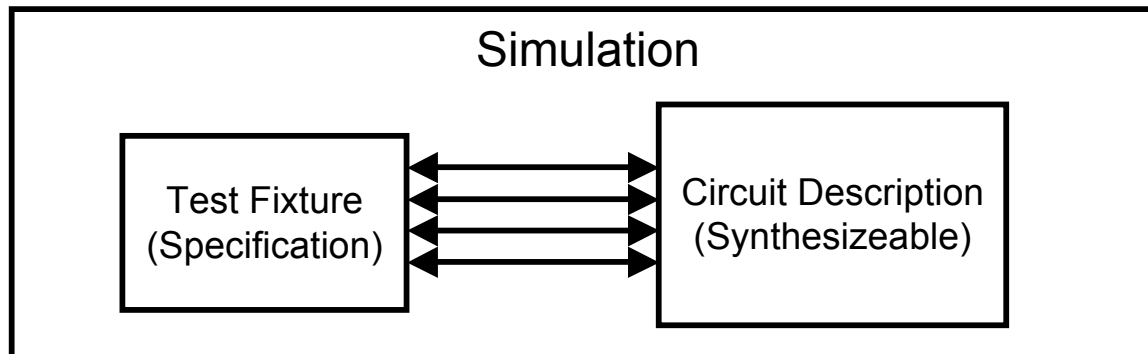
◆ Simulation and synthesis in the CSE curriculum

- CSE370: Learn simulation
- CSE467: Learn synthesis

Simulation

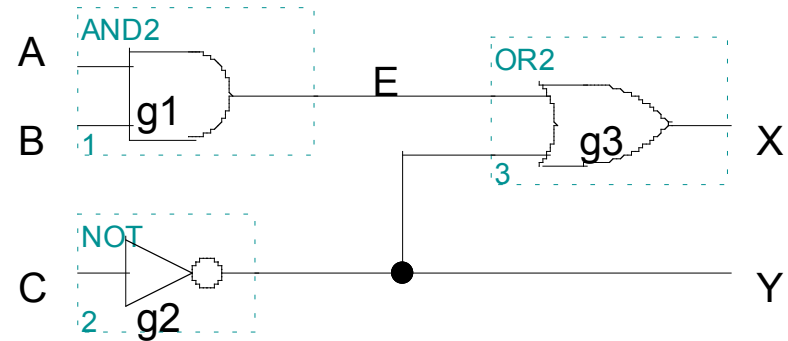
- ◆ You provide an environment
 - Using non-circuit constructs
 - ⇒ Active-HDL waveforms, Read files, print
 - Using Verilog simulation code
 - ⇒ A “test fixture”

Note: We will ignore timing and test benches until next Verilog lecture



Specifying circuits in Verilog

- ◆ There are three major styles
 - Instances `n wires
 - Continuous assignments
 - "always" blocks



"Structural"

```

wire E;
and g1(E,A,B);
not g2(Y,C);
or g3(X,E,Y);
    
```

"Behavioral"

```

wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
    
```

```

reg E, X, Y;
always @ (A or B or C)
begin
    E = A & B;
    Y = ~C;
    X = E | Y;
end
    
```


Data types

◆ Values on a wire

- 0, 1, **x** (unknown or conflict), **z** (tristate or unconnected)

◆ Vectors

- $A[3:0]$ vector of 4 bits: $A[3]$, $A[2]$, $A[1]$, $A[0]$

- ⇒ Unsigned integer value

- ⇒ Indices **must** be constants

- Concatenating bits/vectors

- ⇒ e.g. sign extend

- ◆ $B[7:0] = \{A[3], A[3], A[3], A[3], A[3:0]\};$

- ◆ $B[7:0] = \{4\{A[3]\}, A[3:0]\};$

- Style: Use $a[7:0] = b[7:0] + c;$
Not $a = b + c;$

- Legal syntax: $C = \&A[6:7];$ // logical *and* of bits 6 and 7 of A

Data types that do not exist

- ◆ Structures
- ◆ Pointers
- ◆ Objects
- ◆ Recursive types
- ◆ (Remember, Verilog is not C or Java or Lisp or ...!)

Numbers

- ◆ Format: <sign><size><base format><number>
- ◆ 14
 - Decimal number
- ◆ -4'b11
 - 4-bit 2's complement binary of 0011 (is 1101)
- ◆ 12'b0000_0100_0110
 - 12 bit binary number (_ is ignored)
- ◆ 3'h046
 - 3-digit (12-bit) hexadecimal number
- ◆ Verilog values are unsigned
 - $C[4:0] = A[3:0] + B[3:0];$
 - ⇒ if $A = 0110$ (6) and $B = 1010$ (-6), then $C = 10000$ (*not* 00000)
 - ⇒ B is zero-padded, *not* sign-extended

Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
! ~ & ~& ~ ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{}	concatenation	Concatenation
{{}}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

Similar to C operators

Two abstraction mechanisms

◆ Modules

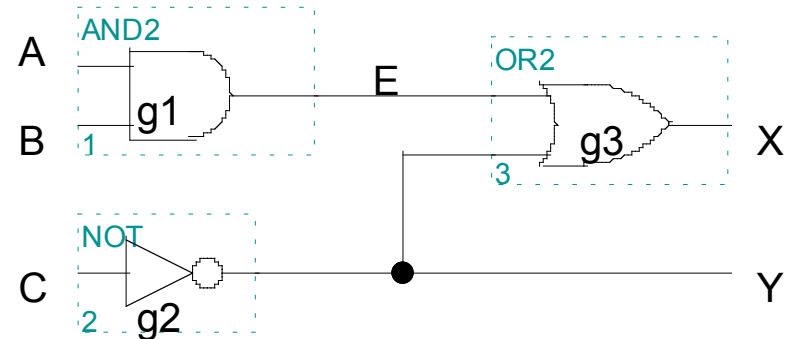
- More structural
- Heavily used in 370 and “real” Verilog code

◆ Functions

- More behavioral
- Used to some extent in “real” Verilog, but not much in 370

Basic building blocks: Modules

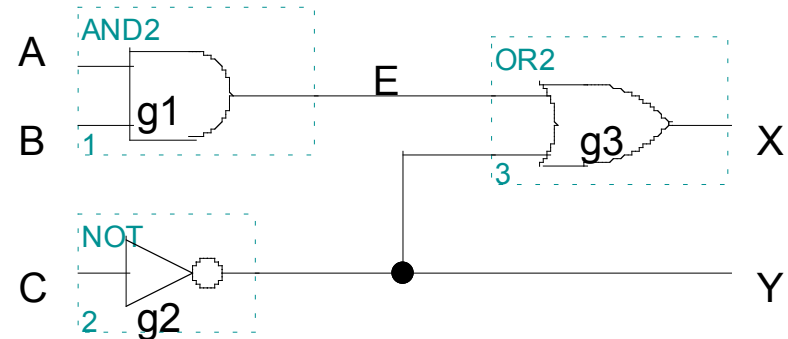
- Instanced into a design
 - ⇒ Never called
- Illegal to nest module defs.
- Modules execute in parallel
- Names are case sensitive
- // for comments
- Name can't begin with a number
- Use wires for connections
- *and*, *or*, *not* are keywords
- All keywords are lower case
- Gate declarations (*and*, *or*, etc)
 - ⇒ List outputs first
 - ⇒ Inputs second



```
// first simple example
module smp1 (X,Y,A,B,C);
    input A,B,C;
    output X,Y;
    wire E
    and g1 (E,A,B);
    not g2 (Y,C);
    or g3 (X,E,Y);
endmodule
```

Modules are circuit components

- Module has ports
 - ⇒ External connections
 - ⇒ A,B,C,X,Y in example
- Port types
 - ⇒ input
 - ⇒ output
 - ⇒ inout (tristate)
- Use assign statements for Boolean expressions
 - ⇒ and \Leftrightarrow &
 - ⇒ or \Leftrightarrow |
 - ⇒ not \Leftrightarrow ~



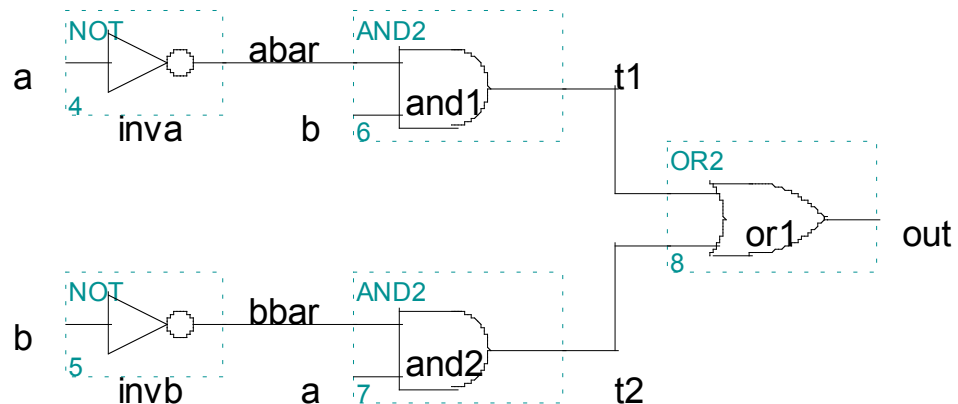
```
// previous example as a
// Boolean expression
module smpl2 (X,Y,A,B,C);
    input A,B,C;
    output X,Y;
    assign X = (A&B) | ~C;
    assign Y = ~C;
endmodule
```

Structural Verilog

```
module xor_gate (out,a,b);  
  input      a,b;  
  output     out;  
  wire      abar, bbar, t1, t2;  
  not       inva (abar,a);  
  not       invb (bbar,b);  
  and       and1 (t1,abar,b);  
  and       and2 (t2,bbar,a);  
  or        or1 (out,t1,t2);  
endmodule
```

8 basic gates (keywords):

and, or, nand, nor
buf, not, xor, xnor



Behavioral Verilog

- ◆ Describe circuit behavior
 - Not implementation



```
module full_addr (Sum,Cout,A,B,Cin) ;  
  input      A, B, Cin;  
  output     Sum, Cout;  
  assign     {Cout, Sum} = A + B + Cin;  
endmodule
```

{Cout, Sum} is a concatenation

Behavioral 4-bit adder

```
module add4 (SUM, OVER, A, B);  
  input [3:0] A;  
  input [3:0] B;  
  output [3:0] SUM;  
  output OVER;  
  assign {OVER, SUM[3:0]} = A[3:0] + B[3:0];  
endmodule
```

“[3:0] A” is a 4-wire bus labeled “A”

Bit 3 is the MSB

Bit 0 is the LSB

Can also write “[0:3] A”

Bit 0 is the MSB

Bit 3 is the LSB

Buses are implicitly connected

If you write BUS[3:2], BUS[1:0]

They become part of BUS[3:0]

Continuous assignment

- ◆ Assignment is continuously evaluated
 - Corresponds to a logic gate
 - Assignments execute in parallel

```
assign A = X | (Y & ~Z);
```

Boolean operators
(~ for bit-wise negation)

```
assign B[3:0] = 4'b01XX;
```

bits can assume four values
(0, 1, X, Z)

```
assign C[15:0] = 4'h00ff;
```

variables can be n-bits wide
(MSB:LSB)

```
assign #3 {Cout, Sum[3:0]} = A[3:0] + B[3:0] + Cin;
```

arithmetic operator

Gate delay (used by simulator)

multiple assignment (concatenation)

Example: A comparator

```
module Compare1 (Equal, Alarger, Blarger, A, B);  
  input      A, B;  
  output     Equal, Alarger, Blarger;  
  assign Equal = (A & B) | (~A & ~B);  
  assign Alarger = (A & ~B);  
  assign Blarger = (~A & B);  
endmodule
```

- Top-down design and bottom-up design are both okay
- ⇒ module ordering doesn't matter
- ⇒ because modules execute in parallel

Comparator example (con't)

```
// Make a 4-bit comparator from 4 1-bit comparators

module Compare4(Equal, Alarger, Blarger, A4, B4);
    input [3:0] A4, B4;
    output Equal, Alarger, Blarger;
    wire e0, e1, e2, e3, A10, A11, A12, A13, B10, B11, B12, B13;

    Compare1 cp0(e0, A10, B10, A4[0], B4[0]);
    Compare1 cp1(e1, A11, B11, A4[1], B4[1]);
    Compare1 cp2(e2, A12, B12, A4[2], B4[2]);
    Compare1 cp3(e3, A13, B13, A4[3], B4[3], );

    assign Equal = (e0 & e1 & e2 & e3);
    assign Alarger = (A13 | (A12 & e3) |
                    (A11 & e3 & e2) |
                    (A10 & e3 & e2 & e1));
    assign Blarger = (~Alarger & ~Equal);
endmodule
```

Sequential assigns don't make any sense

```
assign A = X | (Y & ~Z);
```

```
assign B = W | A;
```

```
assign A = Y & Z;
```



"Reusing" a variable in
several assign statements
is not allowed

Always Blocks

```
reg A, B, C;

always @ (W or X or Y or Z)
begin
    A = X | (Y & ~Z);
    B = W | A;
    A = Y & Z;
    if (A & B) begin
        B = Z;
        C = W | Y;
    end
end
end
```

Variables that appear on the left hand side in an always block must be declared as "reg"s

Sensitivity list

Statements in an always block are executed in sequence

All variables must be assigned on every control path!!! (otherwise you get the dreaded "inferred latch")

Functions

- ◆ Use functions for complex combinational logic

```
module and_gate (out, in1, in2);  
    input        in1, in2;  
    output       out;  
  
    assign out = myfunction(in1, in2);  
  
    function myfunction;  
        input in1, in2;  
        begin  
            myfunction = in1 & in2;  
        end  
    endfunction  
  
endmodule
```

Benefit:

Functions force a result
⇒ Compiler will fail if function
does not generate a result

Sequential Verilog-- Blocking and non-blocking assignments

- ◆ Blocking assignments ($Q = A$)
 - Variable is assigned immediately
 - ⇒ New value is used by subsequent statements
- ◆ Non-blocking assignments ($Q <= A$)
 - Variable is assigned after all scheduled statements are executed
 - ⇒ Value to be assigned is computed but saved for later
 - Usual use: Register assignment
 - ⇒ Registers simultaneously take new values after the clock edge
- ◆ Example: Swap

```
always @(posedge CLK)
begin
    temp = B;
    B = A;
    A = temp;
end
```

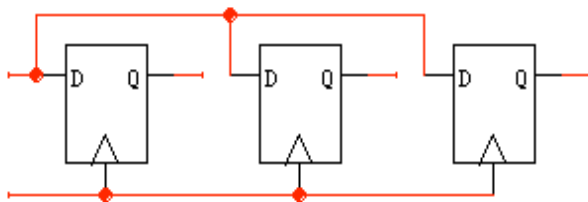
```
always @(posedge CLK)
begin
    A <= B;
    B <= A;
end
```

Sequential Verilog-- Assignments- watch out!

◆ Blocking versus Non-blocking

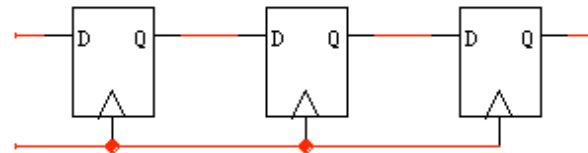
```
reg B, C, D;
```

```
always @(posedge clk)
begin
    B = A;
    C = B;
    D = C;
end
```



```
reg B, C, D;
```

```
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```



Verilog tips

- ◆ **Do not** write C-code
 - Think hardware, not algorithms
 - ⇒ Verilog is **inherently parallel**
 - ⇒ Compilers don't map algorithms to circuits well
- ◆ **Do** describe hardware circuits
 - First draw a dataflow diagram
 - Then start coding
- ◆ References
 - Tutorial and reference manual are found in ActiveHDL help
 - And in today's reading assignment
 - "Starter's Guide to Verilog 2001" by Michael Ciletti
 - copies for borrowing in hardware lab