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

Diagram:

- HDL Description → Synthesis → Gate or Transistor Description
- Simulation → Functional Validation
- Functional/Timing Validation → Physical Implementation → Real Chip!
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
Specify circuits in Verilog

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

"Structural"

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

"Behavioral"

```verilog
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
    ☞ Unsigned integer value
    ☞ Indices must be constants

  ■ Concatenating bits/vectors
    ☞ e.g. sign extend

  ■ 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
  - if A = 0110 (6) and B = 1010(–6), then C = 10000 (not 00000)
  - B is zero-padded, not sign-extended
## Operators

### Verilog Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>reduction NOR</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^ or ^~</td>
<td>reduction XNOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>{}</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>{}[]</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^~ or ^~</td>
<td>bit-wise XNOR</td>
<td>Bit-wise</td>
</tr>
</tbody>
</table>

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 smpl (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 ⇔ \&
  - or ⇔ |
  - not ⇔ ~

// 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
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

```verilog
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;
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

```markdown
assign A = X | (Y & ~Z);
assign B[3:0] = 4'b01XX;
assign C[15:0] = 4'h00ff;
```

- Boolean operators (\(~\) for bit-wise negation)
- Bits can assume four values (0, 1, X, Z)
- Variables can be n-bits wide (MSB:LSB)
- Gate delay (used by simulator)
- Arithmetic operator
- Multiple assignment (concatenation)
Example: A comparator

```verilog
define 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 assignments 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
Functions

- Use functions for complex combinational logic

```verilog
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

```verilog
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**

```verilog
define reg B, C, D;

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

```verilog
define 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
    - copies for borrowing in hardware lab