Lecture 7

- Verilog
  - Structural constructs
  - Describing combinational circuits

- Additional references
  - Tutorial and reference manual are found in ActiveHDL help
  - “Starter’s Guide to Verilog 2001” by Michael Ciletti copies for borrowing in hardware lab

Combinational design

- Step 1: Understand the problem
  - Identify the inputs and outputs
  - Draw a truth table
- Step 2: Simplify the logic
  - Draw a K-map
  - Write a simplified Boolean expression
    - SOP or POS
    - Use don’t cares
- Step 3: Implement the design
  - Logic gates and/or
  - Verilog

Ways of specifying circuits

- Schematics
  - Structural description
  - Describe circuit as interconnected elements
    - Build complex circuits using hierarchy
      - Large circuits are unreadable
- Hardware description languages (HDLs)
  - Not programming languages
    - Parallel languages tailored to digital design
  - Synthesize code to produce a circuit

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
  - Models what a circuit does
    - Multiply is "*", ignoring implementation options
  - Allows you to test design options
  - "Execute" a design to verify correctness

- 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”

Specifying circuits in Verilog

- Three major styles
  - Instances and wires
  - Continuous assignments
  - “always” blocks

"Structural"  "Behavioral"

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

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

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

- **Values on a wire**
  - 0, 1, \( x \) (unknown or conflict), \( z \) (tristate or unconnected)

- **Vectors**
    - Unsigned integer value
    - Indices must be constants

### Manipulating vectors

- Concatenating bits/vectors, use \{ \}
  - e.g. sign extend

- Style: Use \( a[7:0] = b[7:0] + c[7:0] \)
  - Not \( a = b + c; \)

### Data types that do **NOT** exist

- Structures
- Pointers
- Objects
- Recursive types

Verilog is not C or Java or Lisp or ...!

### Numbers

- Format: \(<\text{sign}><\text{size}><\text{base}><\text{number}>\)
  - 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)
  - 12'h046
    - 3-digit (12-bit) hexadecimal number
Numbers are unsigned

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

Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>($)</td>
<td>bit-select or port-select</td>
<td></td>
</tr>
<tr>
<td>($)</td>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>(=)</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>(=)</td>
<td>sign or</td>
<td>Logical</td>
</tr>
<tr>
<td>(=)</td>
<td>reduction</td>
<td>Binary</td>
</tr>
<tr>
<td>(=)</td>
<td>AND</td>
<td>Reduction</td>
</tr>
<tr>
<td>(=)</td>
<td>OR</td>
<td>Reduction</td>
</tr>
<tr>
<td>(=)</td>
<td>NOD</td>
<td>Reduction</td>
</tr>
<tr>
<td>(=)</td>
<td>XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>(+)</td>
<td>unary (left shift)</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>(+)</td>
<td>unary (right shift)</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>(%)</td>
<td>modulo</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>(%)</td>
<td>divide</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>
</tbody>
</table>

Operators similar to Java operators

<table>
<thead>
<tr>
<th>relational operator</th>
<th>greater than or equal to</th>
<th>less than or equal to</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>==</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td></td>
<td></td>
<td>Equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise OR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>logical AND</td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
</tr>
<tr>
<td>%</td>
<td>conditional</td>
<td>Conditional</td>
<td></td>
</tr>
</tbody>
</table>

Two abstraction mechanisms

- Modules
  - More structural, but also behavioral
  - 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

```verilog
// first simple example module simple (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
```
Basic building blocks: modules

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

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 ⇔ |
- Keywords are in lowercase
- and, or, not

// previous example as a Boolean expression
module simple2 (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

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

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

Continuous assignment

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

```
assign A = X | (Y & ~Z);
assign B[3:0] = 4'b01XX;
assign C[15:0] = 16'h00ff;
```

Example: 4-bit 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
```

Invalid sequential assigns

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

"Reusing" a variable on the left side of several assign statements is not allowed

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

Cyclic dependencies also are bad
A depends on X
which depends on B
which depends on A

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

Starting with 1-bit comparator
- Top-down design and bottom-up design are both okay
- Module ordering doesn’t matter because modules execute in parallel
4-bit comparator

// 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, Al0, Al1, Al2, Al3, B10, B11, B12, B13;
    Compare1 cp0(e0, Al0, Bl0, A4[0], B4[0]);
    Compare1 cp1(e1, Al1, Bl1, A4[1], B4[1]);
    Compare1 cp2(e2, Al2, Bl2, A4[2], B4[2]);
    Compare1 cp3(e3, Al3, Bl3, A4[3], B4[3]);
    assign Equal = (e0 & e1 & e2 & e3);
    assign Alarger = (Al3 | (Al2 & e3) | (Al1 & e3 & e2) | (Al0 & e3 & e2 & e1));
    assign Blarger = (~Alarger & ~Equal);
endmodule

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

Always code blocks

- Variables that appear on the left hand side in an always block must be declared as 'reg's
  - Sensitivity list: block is executed each time one of them changes value
  - Statements in an always block are executed in sequence
  - BAD: All variables must be assigned on every control path!!

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 parallel assignment
    - Usual use: Register assignment
      - Registers simultaneously take new values after the clock edge
### Blocking vs. non-blocking

- **Example: Swap**

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

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