Lecture 7

- **Logistics**
  - Homework 2 due today
  - Homework 3 out today due next Wednesday
  - First midterm a week Friday: Friday Jan 30
    - Will cover up to the end of Multiplexers/DeMultiplexers

- **Last lecture**
  - K-Maps

- **Today**
  - Verilog
    - Structural constructs
    - Describing combinational circuits

Summary of two-level combinational-logic

- Logic functions and truth tables
  - AND, OR, Buf, NOT, NAND, NOR, XOR, XNOR
  - Minimal set

- Axioms and theorems of Boolean algebra
  - Proofs by re-writing
  - Proofs by truth table

- Gate logic
  - Networks of Boolean functions
  - NAND/NOR conversion and de Morgan’s theorem

- Canonical forms
  - Two-level forms
  - Incompletely specified functions (don’t cares)

- Simplification
  - Two-level simplification (K-maps)

Solving combinational design problems

- **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

- **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 (programmable logic devices)
  - 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 (Department of Defence) 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

- 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 later

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 $a[7:0] = b[7:0] + c[7:0]$
    - $a = b + c$;

Specifying circuits in Verilog

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

"Structural"

```
wire E;
and g1(E,A,B);
not g2(Y,C);
or  g3(X,E,Y);
wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
```

"Behavioral"

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

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)
- 12'h046
  - 3-digit (12-bit) hexadecimal number

Verilog values are unsigned

\[
\text{C}[4:0] = \text{A}[3:0] + \text{B}[3:0];
\]

- 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>Operator</th>
<th>Name</th>
<th>Relationship Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Logical exclusive OR (XOR)</td>
<td>Logical</td>
</tr>
<tr>
<td>&amp;</td>
<td>And</td>
<td>Logical</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Xor</td>
<td>Logical</td>
</tr>
<tr>
<td>~</td>
<td>Not</td>
<td>Logical</td>
</tr>
<tr>
<td>=</td>
<td>Equality</td>
<td>Relational</td>
</tr>
<tr>
<td>=&lt;</td>
<td>Greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>=&lt;&gt;</td>
<td>Notequal</td>
<td>Relational</td>
</tr>
<tr>
<td>=&lt;=&gt;</td>
<td>Bitwise AND</td>
<td>Bitwise</td>
</tr>
<tr>
<td>=&lt;&gt;&lt;&gt;=</td>
<td>Bitwise OR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>=&lt;&gt;&lt;&gt;</td>
<td>Bitwise XOR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>=&lt;&gt;&lt;&gt;&lt;&gt;=</td>
<td>Bitwise XNOR</td>
<td>Bitwise</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

- Instances into a design (like macros)
  - Never called
- Illegal to nest module defs.
- Modules execute in parallel

// first simple example
module xor_gate (out,a,b);
  input a,b;
  output out;
  wire  sbar, bbar, t1, t2;
  not   inva(sbar,a);
  not   invb(bbar,b);
  and   and1(t1,a,b);
  or    or1(out,t1,t2);
endmodule

Structural Verilog

module xor_gate (out,a,b);
  input a,b;
  output out;
  wire  sbar, bbar, t1, t2;
  not   inva(sbar,a);
  not   invb(bbar,b);
  and   and1(t1,a,b);
  or    or1(out,t1,t2);
endmodule
Behavioral Verilog

- Describe circuit behavior
  - Not implementation

module full_adder (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" Buses are implicitly connected
Bit 0 is the MSB
If you write BUS[3:2], BUS[1:0]
Bit 3 is the LSB
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);
assign B[3:0] = 4'b1100;
assign C[15:0] = 16'h00ff;

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

Example: A comparator (cont)

// 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, Bl1, Bl2, Bl3;
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

Comparator example (con't)

Sequential assigns don't make any sense

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

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

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

Always Blocks

```verilog
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
```

- Sensitivity list: block is executed each time one of them changes value
- Variables that appear on the left hand side in an always block must be declared as "reg's" (can be used to represent every value change)
- 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)

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 parallel assignment
    - Useful for Register assignment
- Example: Swap

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

**Sequential Verilog-- Assignments- watch out!**

- **Blocking versus Non-blocking**

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

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

- Don’t forget variables in this list!
- **Always @ * can be used to represent every value change**

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
    - copies for borrowing in hardware lab