Overview

- Last lecture
  - Incompletely specified functions

- Today
  - 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)
  - Similar to C
  - Moved to public domain in 1990

- VHDL (~1987)
  - DoD sponsored
  - 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 versus synthesis

- **Simulation**
  - "Execute" a design to verify correctness

- **Synthesis**
  - Generate a netlist from HDL code

Simulation versus 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

- **Verilog and VHDL simulate and synthesize**
  - CSE370: Learn simulation
  - CSE467: Learn synthesis

Simulation

- **You provide an environment**
  - Using non-circuit constructs
    - Read files, print, control simulation
  - Using Verilog simulation code
    - A "test fixture"

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

Levels of abstraction

- **Verilog supports 4 description levels**
  - Switch
  - Gate
  - Dataflow
  - Algorithmic

- **Can mix & match levels in a design**

- **Designs that combine dataflow and algorithmic constructs and synthesis are called RTL**
  - Register Transfer Level
Structural versus behavioral Verilog

- **Structural**
  - Describe explicit circuit elements
  - Describe explicit connections between elements
    - Connections between logic gates
  - Just like schematics, but using text

- **Behavioral**
  - Describe circuit as algorithms/programs
    - What a component does
    - Input/output behavior
  - Many possible circuits could have same behavior
    - Different implementations of a Boolean function

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

Basic building blocks: Modules

- Instanced into a design
  - Never called
- Illegal to nest module def.
- 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
Structural Verilog

```
module xor_gate (out, a, b);
    input a, b;
    output out;
    wireabar, 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 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]

---

Data types

- Values on a wire
  - 0, 1, x (don’t care), 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 = 8A[6:7]; // logical and of bits 6 and 7 of A
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

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Packed Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>parens</td>
<td></td>
</tr>
<tr>
<td>$&gt;$</td>
<td>logical negative</td>
<td>Logical Negation</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>reduction AND</td>
<td>Reduction AND</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>reduction OR</td>
<td>Reduction OR</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>reduction XOR</td>
<td>Reduction XOR</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>reduction EXNOR</td>
<td>Reduction EXNOR</td>
</tr>
<tr>
<td>$+$</td>
<td>any logical</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>$+$</td>
<td>any logical</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>$+$</td>
<td>any logical</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>$+$</td>
<td>any logical</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>shift right</td>
<td>Shift</td>
</tr>
</tbody>
</table>

Verilog operators are similar to C operators:

Example: A comparator

```vhdl
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

Continuous assignment

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

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

Gate delay (used by simulator) multiple assignment (concatenation)
// 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, 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

module and_gate (out, in1, in2);
input         in1, in2;
output        out;
assign out = myfunction(in1, in2);
function
myfunction;
input in1, in2;
beg
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
```

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

Sequential Verilog-- Assignments- watch out!

- **Blocking** versus **Non-blocking**

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

```verilog
reg B, C, D;
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```
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 perfect induction (fill in 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