Verilog Synthesis Examples

CS/EE 3710
Fall 2010

Mostly from CMOS VLSI Design by Weste and Harris

Behavioral Modeling
- Using continuous assignments
  - ISE can build you a nice adder
  - Easier than specifying your own

```verilog
module adder(input [31:0] a, input [31:0] b, output [31:0] y);
    assign y = a + b;
endmodule
```

Bitwise Operators
- Bitwise operations act on vectors (buses)

```verilog
module inv(input [3:0] a, output [3:0] y);
    assign y = ~a;
endmodule
```

More bitwise operators

```verilog
module prolog(input [15:0] a, b, output [3:0] y1, y2, y3, y4);

// Some different two input logic
// Same netting on a 4-bit version
assign y1 = a & b; // AND
assign y2 = a | b; // OR
assign y3 = a ^ b; // XOR
assign y4 = ~(a | b); // Nand
endmodule
```

Reduction Operators
- Apply operator to a single vector
  - Reduce to a single bit answer

```verilog
module sum(input [7:0] a, output y);
    assign y = a;
endmodule
```

Conditional Operator
- Classic mux
  - Can be confusing if you get crazy

```verilog
module mux2(input [3:0] d0, d1, input w, output [3:0] y);
    assign y = w ? d1 : d0;
endmodule
Using internal signals

- Internal wires and regs can be used inside a module

```
module fulladder(input a, b, cin, output sum, carry);
wire prop;
assign prop = a & b;
assign sum = prop + cin;
assign carry = (a & b) | (cin & (a | b));
endmodule
```

Using internal signals

- Internal wires and regs can be used inside a module

```
module fulladder(input a, b, cin, output sum, carry);
wire prop;
assign prop = a & b;
assign sum = prop + cin;
assign carry = (a & b) | (cin & (a | b));
endmodule
```

Operator Precedence

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
<td>Highest</td>
</tr>
<tr>
<td>*, /, %</td>
<td>MUL, DIV, MODULO</td>
<td></td>
</tr>
<tr>
<td>+, -</td>
<td>PLUS, MINUS</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td>Logical Left/Right Shift</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>Arithmetic Left/Right Shift</td>
<td></td>
</tr>
<tr>
<td>=, !=</td>
<td>Assign Value</td>
<td></td>
</tr>
<tr>
<td>&lt;, &lt;=</td>
<td>True Comparison</td>
<td></td>
</tr>
<tr>
<td>&gt;, &gt;=</td>
<td>False Comparison</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>AND, NAND</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR, NXOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>, ~</td>
<td>OR, NOR</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

Constants

- Specified in binary, octal, decimal, or hex
- Note use of underscore in long binary numbers

<table>
<thead>
<tr>
<th>Number</th>
<th>Base</th>
<th>Decimal Equivalent</th>
<th>Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b11</td>
<td>Binary</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>'b111</td>
<td>Binary</td>
<td>3</td>
<td>0000011</td>
</tr>
<tr>
<td>'b1101</td>
<td>Binary</td>
<td>5</td>
<td>1010</td>
</tr>
<tr>
<td>'b1010</td>
<td>Binary</td>
<td>10</td>
<td>1010101</td>
</tr>
<tr>
<td>'b1010</td>
<td>Decimal</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>'b1010</td>
<td>Octal</td>
<td>34</td>
<td>100340</td>
</tr>
<tr>
<td>'b1010</td>
<td>Hexadecimal</td>
<td>171</td>
<td>4101011</td>
</tr>
<tr>
<td>'b1010</td>
<td>Unspecified</td>
<td>42</td>
<td>000...00101010</td>
</tr>
</tbody>
</table>

Hierarchy

- Instantiate other modules in your module

```
module adder(input a, b, output sum, carry);
wire prop1, prop2;
assign prop1 = a & b;
assign prop2 = cin & (a | b);
assign sum = prop1 + prop2 + cin;
assign carry = (a & b) | (prop1 & prop2) | (prop1 & cin) | (prop2 & cin);
endmodule
```

Tristates

- Assign the value z
- Just say NO!
- No on-board tri-states on Spartan3e FPGAs
- Use MUXs instead!
**Bit Swizzling**

- Sometimes useful to work on part of a bus, or combine different signals together
  - Use bus (vector) notation

**Registers**

- Edge-triggered flip flops
  - Always use reset of some sort!

**Counters**

- Behavioral

**Bitswizzling**

- Sometimes useful to work on part of a bus, or combine different signals together
  - Use concatenation \{\} operator

**Registers**

- Can also add an enable signal
  - Only capture new data on clock and en

**Counters**

- Structural
Comb Logic with Always blocks
- Always blocks are often sequential
  - But, if you have all RHS variables in the sensitivity list it can be combinational
  - Remember that you still must assign to a reg

```
module latching
  ( [n] x, output reg [y] y)
always @(x)
  y <= #1 x;
endmodule
```

Decoder example (combinational)
```
module decoder ( [n] x, output reg [y] y);
  // a 3:8 decoder
  always @(x)
    case (x)
      2'0:  y <= 8'00000001;
      2'1:  y <= 8'00000000;
      2'2:  y <= 8'00000100;
      2'3:  y <= 8'00000101;
      ...
      2'6:  y <= 8'00000000;
      2'7:  y <= 8'00000000;
    endcase
endmodule
```

Continuous assignment version is not as readable
Same circuit though…

Seven Segment Decoder
```
module seven_segment
  ( [n] x, output reg [y] y);
  // seven segment decoder
  always @(x)
    case #(50ns) x
      2'0:  y <= 8'00000001;
      2'1:  y <= 8'00000000;
      2'2:  y <= 8'00000100;
      2'3:  y <= 8'00000101;
      ...
      2'6:  y <= 8'00000000;
      2'7:  y <= 8'00000000;
    endcase
endmodule
```

Memories
- Generally translates to block RAMs on the Spartan3e FPGA
Shift Register?

Blocking vs. Non-Blocking

Finite State Machines

Example

Example
Mealy vs. Moore

![Diagram of Mealy vs. Moore machines](image)

**Fig A.3** Moore and Mealy machines

Mealy example

![Diagram of Mealy FSM](image)

Output is true if input is the same as it was on the last two cycles.

Mealy Example

```verilog
module history54(input clk, input reset, input x, output y);

reg [3:0] state, nextstate;
parameter S0 = 3'd0000;
parameter S1 = 3'd0010;
parameter S2 = 3'd0101;
parameter S3 = 3'd0110;
parameter S4 = 3'd1001;

// State Register
always @(posedge clk, posedge reset)
if (reset) state <= S0;
else state <= nextstate;

// Output Logic
assign y = (state[1] & x) ? (state[2] & ~x) :
                           ~x;
endmodule
```

Parameterized Modules

```verilog
module dff (parameter width = 8)
(input [width-1:0] a, output [width-1:0] y);
assign y = ~a;
endmodule
```

Verilog Style Guide

- Use only non-blocking assignments in **always** blocks
- Define combinational logic using **assign** statements whenever practical
  - Unless **if** or **case** makes things more readable
  - When modeling combinational logic with **always** blocks, if a signal is assigned in one branch of an **if** or **case**, it needs to be assigned in all branches
- Include default statements in your **case** statements
- Use **parameters** to define state names and constants
- Properly indent your code
- Use comments liberally
- Use meaningful variable names
- Do NOT ignore synthesis warnings unless you know what they mean!
Verilog Style Guide

- Be very careful if you use both edges of the clock
  - It’s much safer to stick with one
  - i.e. @(posedge clock) only
- Be certain not to imply latches
  - Watch for synthesis warnings about implied latches
- Provide a reset on all registers

Verilog Style Guide

- Provide a common clock to all registers
  - Avoid gated clocks
  - Use enables instead