

Verilog Synthesis Examples

CS/EE 3710
Fall 2008

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

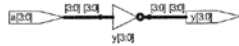
```
module adder(input [31:0] a,  
            input [31:0] b,  
            output [31:0] y);  
  
    assign y = a + b;  
endmodule
```



Bitwise Operators

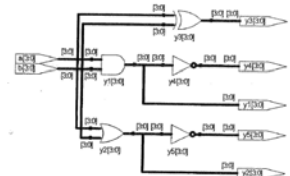
- Bitwise operations act on vectors (buses)

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



More bitwise operators

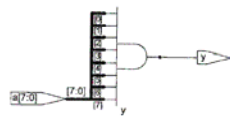
```
module gates(input [3:0] a, b,  
            output [3:0] y1, y2, y3, y4, y5);  
  
    /* Five different two-input logic  
    gates acting on 4 bit buses */  
    assign y1 = a & b; // AND  
    assign y2 = a | b; // OR  
    assign y3 = a ^ b; // XOR  
    assign y4 = ~(a & b); // NAND  
    assign y5 = ~(a | b); // NOR  
endmodule
```



Reduction Operators

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

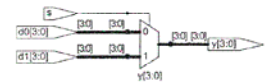
```
module and8(input [7:0] a,  
          output y);  
  
    assign y = &a;  
endmodule
```



Conditional Operator

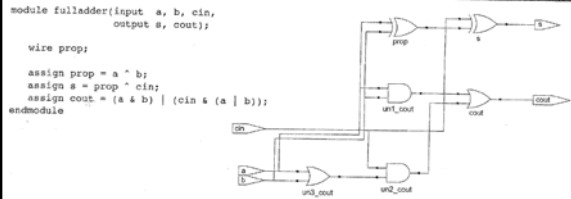
- Classic mux
 - Can be confusing if you get crazy

```
module mux2(input [3:0] d0, d1,  
          input s,  
          output [3:0] y);  
  
    assign y = s ? d1 : d0;  
endmodule
```



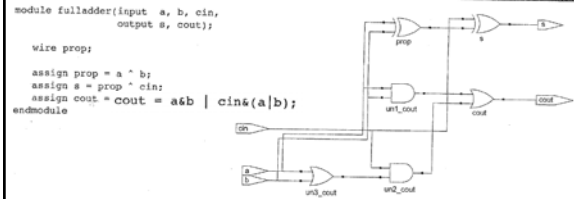
Using internal signals

- Internal wires and regs can be used inside a module



Using internal signals

- Internal wires and regs can be used inside a module



Operator Precedence

Table A.1 Operator Precedence

Symbol	Meaning	Precedence
~	NOT	Highest
*, /, %	MUL, DIV, MODULO	
+, -	PLUS, MINUS	
<<, >>	Logical Left/Right Shift	
<<<, >>>	Arithmetic Left/Right Shift	
<, <=, >, >=	Relative Comparison	
==, !=	Equality Comparison	
&, ~&	AND, NAND	
^, ~^	XOR, XNOR	
, ~	OR, NOR	
?:	Conditional	Lowest

Constants

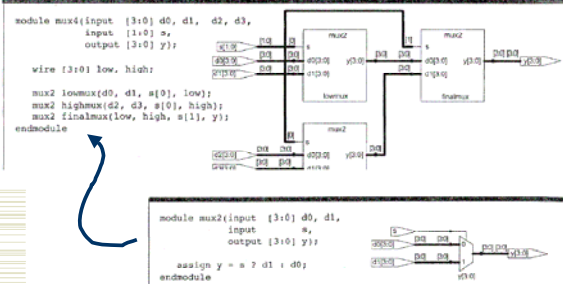
- Specified in binary, octal, decimal, or hex
 - Note use of underscore in long binary numbers

Table A.2 Constants

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	Binary	5	101
'b11	unsized	Binary	3	000000...00011
8'b11	8	Binary	3	00000011
8'b1010_1011	8	Binary	171	10101011
3'd6	3	Decimal	6	110
6'o42	6	Octal	34	100010
8'hAB	8	Hexadecimal	171	10101011
42	unsized	Decimal	42	0000...00101010

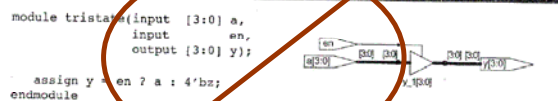
Hierarchy

- Instantiate other modules in your module



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



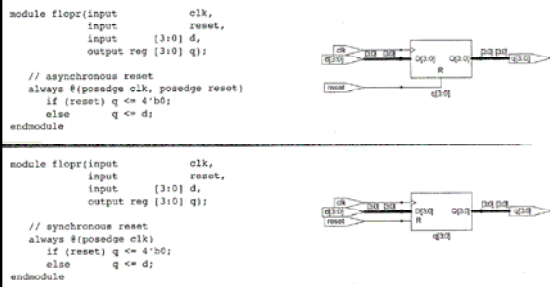
Bit Swizzling

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



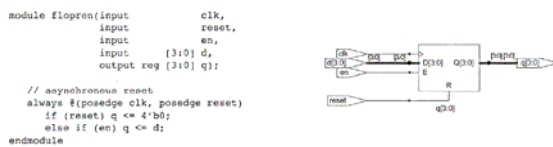
Registers

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



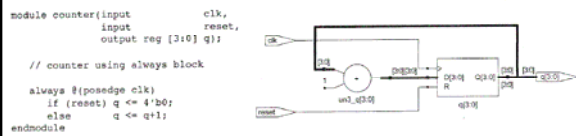
Registers

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



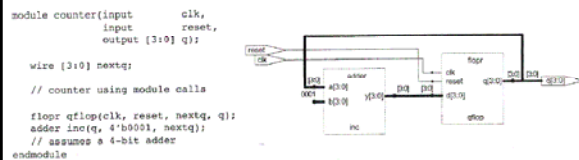
Counters

- ◆ Behavioral



Counters

- ◆ Structural

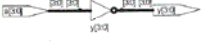


Comb Logic with Always blocks

- 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 type

```

module inv(input [3:0] a,
           output reg [3:0] y);
    always @(*)
        y <= ~a;
endmodule
    
```

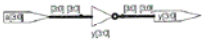


Comb Logic with Always blocks

- 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 type

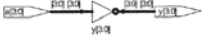
```

module inv(input [3:0] a,
           output reg [3:0] y);
    always @(*)
        y <= ~a;
endmodule
    
```



```

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

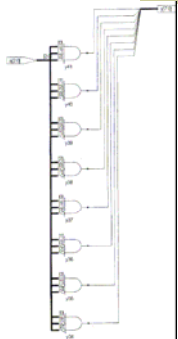


Decoder example (combinational)

```

module decoder_always(input [2:0] a,
                    output reg [7:0] y);

// a 3:8 decoder
always @(*)
    case (a)
        3'b000: y <= 8'b00000001;
        3'b001: y <= 8'b00000010;
        3'b010: y <= 8'b00000100;
        3'b011: y <= 8'b00001000;
        3'b100: y <= 8'b00010000;
        3'b101: y <= 8'b00100000;
        3'b110: y <= 8'b01000000;
        3'b111: y <= 8'b10000000;
    endcase
endmodule
    
```

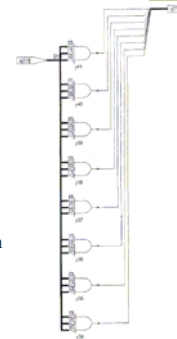


Decoder example (combinational)

```

module decoder_assign(input [2:0] a,
                    output [7:0] y);

    assign y[0] = ~a[0] & ~a[1] & ~a[2];
    assign y[1] = a[0] & ~a[1] & ~a[2];
    assign y[2] = ~a[0] & a[1] & ~a[2];
    assign y[3] = a[0] & a[1] & ~a[2];
    assign y[4] = ~a[0] & ~a[1] & a[2];
    assign y[5] = a[0] & ~a[1] & a[2];
    assign y[6] = ~a[0] & a[1] & a[2];
    assign y[7] = a[0] & a[1] & a[2];
endmodule
    
```



Continuous assignment version is not as readable
Same circuit though...

Seven Segment Decoder

```

module sevenseg(input [3:0] data,
               output reg [6:0] segments);

// Segment #
parameter BLANK = 7'b000_0000;
parameter ZERO = 7'b111_1110;
parameter ONE = 7'b011_0000;
parameter TWO = 7'b110_1101;
parameter THREE = 7'b111_1001;
parameter FOUR = 7'b011_0011;
parameter FIVE = 7'b101_1011;
parameter SIX = 7'b101_1111;
parameter SEVEN = 7'b111_0000;
parameter EIGHT = 7'b111_1111;
parameter NINE = 7'b111_1011;

always @(*)
    case (data)
        0: segments <= ZERO;
        1: segments <= ONE;
        2: segments <= TWO;
        3: segments <= THREE;
        4: segments <= FOUR;
        5: segments <= FIVE;
        6: segments <= SIX;
        7: segments <= SEVEN;
        8: segments <= EIGHT;
        9: segments <= NINE;
        default: segments <= BLANK;
    endcase
endmodule
    
```



Memories

- Generally translates to block RAMs on the Spartan3e FPGA

```

module ram(input clk,
          input [5:0] addr,
          input wrb,
          input [15:0] din,
          output [15:0] dout);

    reg [15:0] mem[63:0]; // the memory

    always @(posedge clk)
        if (~wrb) mem[addr] <= din;

    assign dout = mem[addr];
endmodule
    
```

Shift Register?

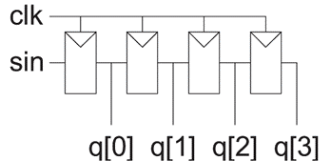


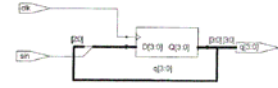
FIG A.2 Intended shift register

Blocking vs. Non-Blocking

◆ Shift Register?

```

module shiftreg(input      clk,
               input      sin,
               output reg [3:0] q);
// This is a correct implementation
// using nonblocking assignment
always @(posedge clk)
begin
  q[0] <= sin; // nonblocking <=
  q[1] <= q[0];
  q[2] <= q[1];
  q[3] <= q[2];
  // even better to write
  // q <= {q[2:0], sin};
end
endmodule
    
```

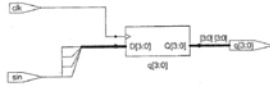


Blocking vs. Non-Blocking

◆ Shift Register?

```

module shiftreg(input      clk,
               input      sin,
               output reg [3:0] q);
// This is a bad implementation
// using blocking assignment
always @(posedge clk)
begin
  q[0] = sin; // blocking =
  q[1] = q[0];
  q[2] = q[1];
  q[3] = q[2];
end
endmodule
    
```



Finite State Machines

◆ Divide into three sections

- State register
- Next state logic
- output logic

◆ Use parameters for state encodings

Example

- ◆ Three states, no inputs, one output, two state bits

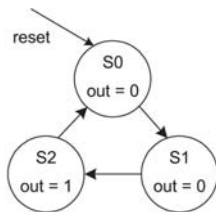
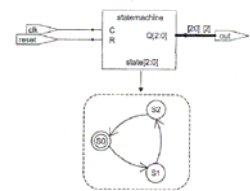


FIG A.4 Divide-by-3 counter state transition diagram

Example

```

module divideby3FSM(input clk,
                  input reset,
                  output out);
  reg [1:0] state, nextstate;
  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  // State Register
  always @(posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
  // Next State Logic
  always @(*)
  case (state)
    S0: nextstate <= S1;
    S1: nextstate <= S2;
    S2: nextstate <= S0;
    default: nextstate <= S0;
  endcase
  // Output Logic
  assign out = (state == S2);
endmodule
    
```



Mealy vs. Moore

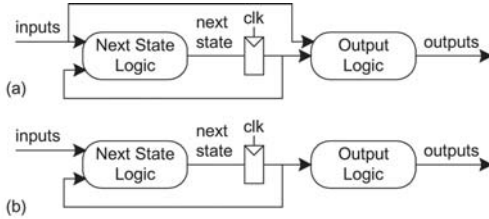
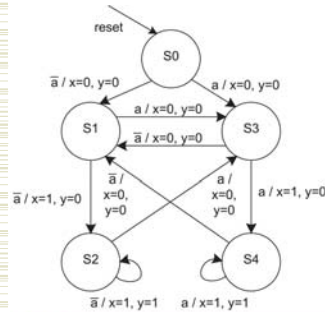


FIG A.3 Moore and Mealy machines

Mealy example



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

FIG A.5 History FSM state transition diagram

Mealy Example

```

module historyFSM(input clk,
                 input reset,
                 input a,
                 output x, y);

    reg [2:0] state, nextstate;

    parameter S0 = 3'b000;
    parameter S1 = 3'b010;
    parameter S2 = 3'b011;
    parameter S3 = 3'b100;
    parameter S4 = 3'b101;

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

    // Next State Logic
    always @(*)
        case (state)
            S0: if (a) nextstate <= S3;
                else nextstate <= S1;
            S1: if (a) nextstate <= S3;
                else nextstate <= S2;
            S2: if (a) nextstate <= S3;
                else nextstate <= S2;
            S3: if (a) nextstate <= S4;
                else nextstate <= S1;
            S4: if (a) nextstate <= S4;
                else nextstate <= S1;
            default: nextstate <= S0;
        endcase

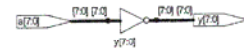
    // Output Logic
    assign x = (state[1] & ~a) |
              (state[2] & a);
    assign y = (state[1] & state[0] & ~a) |
              (state[2] & state[0] & a);
endmodule
    
```

Parameterized Modules

```

module inv
    #(parameter width = 8)
    (input [width-1:0] a,
     output [width-1:0] y);

    assign y = ~a;
endmodule
    
```



```

module buffer
    #(parameter numbits = 12)
    (input [numbits-1:0] a,
     output [numbits-1:0] y);

    wire [numbits-1:0] x;
    inv #(numbits) i1(a, x);
    inv #(numbits) i2(x, y);
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

Verilog Style Guide

- ◆ 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