Verilog

Verilog for Synthesis
Verilog background

- 1983: Gateway Design Automation released Verilog HDL “Verilog” and simulator
- 1985: Verilog enhanced version – “Verilog-XL”
- 1987: Verilog-XL becoming more popular (same year VHDL released as IEEE standard)
- 1989: Cadence bought Gateway
- 1995: Verilog adopted by IEEE as standard 1364
  - Verilog HDL, Verilog 1995
- 2001: First major revision (cleanup and enhancements)
  - Standard 1364-2001 (or Verilog 2001)
- System Verilog under development
  - Better system simulation and verification support
Books

- “HDL Chip Design” by Smith, 1996, Doone Publications, 0-9651934-8
Create Verilog Module

[Image of the New Source dialog box]

[Image of the Define Verilog Source dialog box]
Module Created

- No separate entity and arch – just module
- Ports can be input, output, or inout
- Note: Verilog 2001 has alternative port style:
  - (input a, b, sel, output y);
Add assign statement

- Similar to VHDL conditional signal assignment – continuous assignment
- Same hardware produced as with VHDL
Verilog - general comments

• VHDL is like ADA and Pascal in style
  • Strongly typed – more robust than Verilog
  • In Verilog it is easier to make mistakes
    • Watch for signals of different widths
    • No default required for case statement, etc
• Verilog is more like the ‘c’ language
• Verilog IS case sensitive
• White space is OK
• Statements terminated with semicolon (;)
• Verilog statements between
  • module and endmodule
• Comments // single line and /* and */
Verilog

- Four-value logic system
  - 0 – logic zero, or false condition
  - 1 – logic 1, or true condition
  - x, X – unknown logic value
  - z, Z - high-impedance state
- Number formats
  - b, B binary
  - d, D decimal (default)
  - h, H hexadecimal
  - o, O octal
  - 16’H789A – 16-bit number in hex format
  - 1’b0 – 1-bit
Verilog types

- **Constants**
  - `parameter` DIME = 10;
  - `parameter` width = 32, nickel = 5;
  - `parameter` quarter = 8’b0010_0101;

- **Nets**
  - `wire` clock, reset_n;
  - `wire[7:0]` a_bus;

- **Registers**
  - `reg` clock, reset_n;
  - `reg[7:0]` a_bus;

- **Integer**
  - only for use as general purpose variables in loops
  - `integer` n;
operators

• bitwise

  - ~ negation
  - & and
  - | inclusive or
  - ^ exclusive or
  - y = ~(a & b);
  - y = ~ a;

• reduction (no direct equivalent in vhdl)

  - accept single bus and return single bit result

  - & and
  - ~& nand
  - | or
  - ^ exclusive or

y = a & b;
y = a AND b;
y = a OR b;
y = a XOR b;
y = A NAND b;
y = NOT a;
Operators (cont’d)

- Relational (return 1 for true, 0 for false)
  - `<` less than, `<=`
  - `>` greater than `>=`

- Equality
  - `==` logical equality
  - `!=` logical inequality

- Logical Comparison Operators
  - `!` logical negation
  - `&&` logical and
  - `||` logical or

- Arithmetic Operators
  - `+`
  - `-`
  - `*`
Operators (cont’d)

- **Shift**
  - `<<` logical shift left, (`<<<` arithmetic)
  - `>>` logical shift right (`>>>` arithmetic)

- **Conditional**
  - Only in Verilog - selects one of pair expressions
  - `? :`
  - Logical expression before `?` is evaluated
  - If true, the expression before `:` is assigned to output
  - If false, expression after `:` is assigned to output
    - `Y = (A > B) ? 1 : 0`
    - `Y = (A == B) ? A + B : A - B`
Tri-state example

• Using conditional operator in continuous assignment

```
module tri_ex(input oe, input [7:0] a, output [7:0] y);
  assign y = oe ? a : 8'bZZZZZZZZ;
endmodule
```
Concurrent statements

• VHDL
  – Process
  – Signal assignments

• Verilog
  – always statement
  – Continuous assignment - assign
Sequential Statements

- VHDL
  - reside in process statement
- Verilog
  - reside in an always statement
  - if statements
  - case statements
  - Note: use begin and end to block sequential statements
Verilog wire and register data objects

- **Wire** – net, connects two signals together
  - `wire clk, en;`
  - `wire [15:0] a_bus;`

- **Reg** – register, holds its value from one procedural assignment statement to the next
  - Does not imply a physical register – depends on use
  - `reg [7:0] b_bus;`
Index and Slice

- VHDL
  - Use `to` and `downto` to specify slice
  - Concatenation &
    - `c_bus(3 downto 0) <= b_bus(7 downto 4);`
    - `c_bus(5 downto 0) <= b_bus(7) & a_bus(6 downto 3) & '0';`

- Verilog
  - Use colon :
  - Concatenation {}
    - `assign c_bus[3:0] = b_bus[7:4];`
    - `assign c_bus[5:0] = {b_bus[7], a_bus[6:3], 1'b0};`
Internal wires

- Declare internal wires:

```verilog
module buses (a_bus, b_bus, c_bus, d_bus);
  input [7:0] a_bus;
  input [7:0] b_bus;
  output [7:0] c_bus;
  output [7:0] d_bus;

  wire [7:0] internal_bus;
  assign c_bus[5:0] = (b_bus[7], a_bus[6:3], 1'b0);
  assign internal_bus = (8'h5A & a_bus) | b_bus;
  assign c_bus = ~internal_bus;
endmodule
```
Decoder

- 2 to 4 decoder with enable
- Combinational logic using always statement with sensitivity list
  - similar to VHDL process – for cyclic behavior
  - (@) event control operator
  - begin .. end block statement
  - note reg for y

```verilog
module decoder (sel, en, y);
input [1:0] sel;
input en;
output [3:0] y;

reg y;

always @(sel, en)
begin // required if multiple sequential statements
  if (en == 0)
    y = 4'b1111;
  else
    if (sel == 2'b00)
      y = 4'b1110;
    else if (sel == 2'b01)
      y = 4'b1011;
    else if (sel == 2'b10)
      y = 4'b1011;
    else
      y = 4'b0111;
end
endmodule
```
Decoder (cont’d)

• Combinational logic using always statement with sensitivity list
  – similar to VHDL process – for cyclic behavior
  – (@) event control operator
  – begin .. end block statement
    • Statements execute sequentially
      – if statement
      – case statement
        • Note: case expression can concatenate signals (,{,})
  – Sensitivity list
    • (a or b or c)
    • Verilog 2001 allows comma-separated list (a, b, c)
Flip-flops in Verilog

- Always inferred using edge-triggered `always` statement

```verilog
dtype.v - ISE Text Editor

module dtype(clk, d, q);
  input clk;
  input d;
  output q;

  reg q;

  always @(posedge clk) begin
    q <= d;
  end
endmodule
```
Flip-flops in Verilog (with async)

- Add async signals to sensitivity list
Counters in Verilog

- Just extension of D type
- This example has async clear

```verilog
module counter(clk, clr, q);
    input clk;
    input clr;
    output [3:0] q;

    reg [3:0] q;

    always @(posedge clk or posedge clr)
        if (clr)
            q <= 0;
        else
            q <= q + 1;
endmodule
```
Counters in Verilog (cont’d)

- With terminal count

```verilog
counter.v
module counter2(clk, clr, q);
    input clk;
    input clr;
    output [3:0] q;
    reg [3:0] q;
    always @(posedge clk or posedge clr)
        if (clr)
            q <= 0;
        else if (q == 13) // can read output
            q <= 0;
        else
            q <= q + 1;
endmodule
```
State Machine

- SM1 – 4 states
- Two **always** statements

```
module sm1(clk, en, reset, c);
  input clk;
  input en;
  input reset;
  output c;
  reg c;
  parameter [1:0] s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  reg [1:0] current_state, next_state;

  // sequential logic - use non-blocking
  always @(posedge clk or posedge reset)
  if (reset)
    current_state <= 0;
  else
    current_state <= next_state;

  // combinational logic - use blocking
  always @(current_state, en)
  case (current_state)
    s0: begin
      c = 0;
      if (en)
        next_state = s1;
      else
        next_state = s0;
    end
    s1: begin
      c = 0;
      if (en)
        next_state = s2;
      else
        next_state = s1;
    end
    s2: begin
      c = 1;
      if (en)
        next_state = s3;
      else
        next_state = s0;
    end
    s3: begin
      c = 0;
      if (en)
        next_state = s3;
      else
        next_state = s0;
    end
  endcase
endmodule
```
State Machine hardware

- four flip-flops (can be changed through synthesis options)
- Combinational logic for next state and outputs
Blocking and non-blocking assignment

- To ensure correct synthesis and simulation results:
  - Combinational logic
    - Use blocking assignment = statements in always block
  - Sequential logic
    - Use non blocking assignment <= statements in always block
    - Can only be used on reg types
      - Can only be used in an initial or always procedural blocks
    - Can not be used in continuous assignments
Top-Down Design Hierarchy

- Instantiate module (counter example with decoder)

```verilog
module decoder(count,seven_seg);
    input [3:0] count;
    output [6:0] seven_seg;

    // instantiate decoder module in counter
    // using position of ports
    decoder d1 (count_val, seven_seg_val);

    // or using formal and actual names
    decoder d1 (.count(count_val), .seven_seg(seven_seg_val));
```
Decoder Test Fixture Example

module decoder(
    input [2:0] sel,
    output [7:0] y
    // output reg [7:0] y // required for always statement
);

    // continuous assignment statement used in this example
    assign y = (sel == 0) ? 8'b00000001 :
        (sel == 1) ? 8'b00000010 :
        (sel == 2) ? 8'b00000100 :
        (sel == 3) ? 8'b00001000 :
        (sel == 4) ? 8'b00010000 :
        (sel == 5) ? 8'b00100000 :
        (sel == 6) ? 8'b01000000 :
        (sel == 7) ? 8'b10000000 :
            8'b10000000;

    // alternative style using always with case statement
    // note requirement for reg declaration

    /*
    always @ (sel)
    case (sel)
        0: y = 8'b00000001;
        1: y = 8'b00000010;
        2: y = 8'b00000100;
        3: y = 8'b00001000;
        4: y = 8'b00010000;
        5: y = 8'b00100000;
        6: y = 8'b01000000;
        7: y = 8'b10000000;
    endcase
    */

endmodule
Create Verilog Test Fixture
Test Fixture Created

```verilog
module decoder_tf;

// Inputs
reg [2:0] sel;

// Outputs
wire [7:0] y;

// Instantiate the Unit Under Test (UUT)
decoder uut (;
    .sel(sel),
    .y(y)
);

initial begin
    // Initialise Inputs
    sel = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

end

demodule
```

Jim Duckworth, WPI
Add Stimulus

```verilog
34   decoder uut (
35      .sel(sel),
36      .y(v)
37   );
38
39   initial begin
40     // Initialize Inputs
41     sel = 0;
42
43     // Wait 100 ns for global reset to finish
44     #100;
45
46     // Add stimulus here
47     sel = 3'b001;
48     #50
49     sel = 3'b010;
50     #50
51     sel = 3'b011;
52     #50
53     sel = 3'b100;
54     #50
55     sel = 3'b101;
56     #50
57     sel = 3'b110;
58     #50
59     sel = 3'b111;
60
61   end
62
63   endmodule
```
Simulation Results
Simulation Notes

- **Initial** block declares a single-pass behavior
  - Executes once when simulator is activated
- Delay control operator (#) and delay value - #10
- Timescale compiler directive
  - timescale <reference_time_unit>/<time_precision>
  - `timescale 10ns/1ns
- Inputs are declared as reg values – retains value until updated
- Outputs are just monitored as wires
Generating clocks

- Generating repetitive signals

  ```verilog
  initial
  begin
    clk = 0;
  end
  
  always begin
    #5 clk = ~clk;
  end
  
  Or

  always begin
    #5 clk = 0;
    #5 clk = 1;
  end
  ```