Introduction To Verilog Design

Chun-Hung Chou
Outline

• Typical Design Flow
• Design Method
• Lexical Convention
• Data Type
• Data Assignment
• Event Control
• Conditional Description
• Register Description
• Synthesizable Verilog Code
• Simulation Environment
Typical Design Flow

Design Specification

Algorithmic Model

RTL Model

Gate-Level Model

Switch-Level Model

Physical Layout

Behavior Language

Structural Language

Verilog HDL
Design Method (1/3)

- Top-Down Design
  - design form top module to bottom element

```
Top Module
   /       /
Sub Module Sub Module
   |       |
Element Element Element Element
```
Design Method(2/3)

• **Bottom-Up Design**
  – design from element module to top block
Design Method (3/3)

Module module_name(port name);

- port declaration
- data type declaration
- task & function declaration
- module functionality

Endmodule
Lexical Convention (1/10)

• Number Specification
  – `<size>`’<base>`<value>`
    • `size` is the size in bits
    • `base` can be b(binary), o(octal), d(decimal), or h(hexadecimal)
    • `value` is any legal number in the selected base, including x and z
    • default value is 32 bits decimal number
  – **Negative:** `-<size>`’<base>`<value>`
  – **Example**
    • 8’b1100_0001_1111  →  8’b110000011111
    • 16’habc  →  0000 1010 1011 1100
    • 12’b0  →  0000 0000 0000
    • 6’hx  →  xx xxxx
    • 2’b1101  →  2’b01
Lexical Convention (2/10)

• Operators
  – Arithmetic Operator
    • +    add
    • -    subtract
    • *    multiply
    • /    divide
    • %    modulus

Note: verilog automatically perform a 2’s complement when a negative value is assigned to an unsigned variable such as a reg.

module arithops();
integer ans, int;
parameter five = 5;
reg [3:0] rega, regb;
reg [3:0] num;
initial begin
  rega = 3;
  regb = 4’b1010;
  int = -3;
end
initial begin
  #10 ans = five * int;  // ans = -15
  #10 ans = (int + 5) / 2; // ans = 1
  #10 ans = five / int;  // ans = -1
  #10 num = rega + regb; // num = 1101
  #10 num = rega + 1;   // num = 0100
  #10 num = int;        // num = 1101
  #10 num = regb % rega; // num = 1
end
endmodule
Lexical Convention (3/10)

• Operators

  – Bit-Wise Operator

    • ~ not
    • & and
    • | or
    • ^ xor
    • ~^ xnor
    • ^~ xnor

```
module bitwise();
reg [3:0] rega, regb, regc;
reg [3:0] num;
initial begin
  rega = 4'b1001;
  regb = 4'b1010;
  regc = 4'b11x0
end
initial begin
  #10 num = rega & 0; // num = 0000
  #10 num = rega & regb; // num = 1000
  #10 num = rega | regb; // num = 1011
  #10 num = regb & regc; // num = 10x0
  #10 num = regb | regc; // num = 1110
end
endmodule
```
Lexical Convention (4/10)

• Operators
  – Logical Operator
    • ! not
    • && and
    • || or

module logical();
parameter five = 5;
reg [3:0] rega, regb, regc;
reg [3:0] ans;
initial begin
  rega = 4'b0011;
  regb = 4'b10xz;
  regc = 4'b0z0x;
end
initial begin
  #10 ans = rega && 0; // ans = 0
  #10 ans = rega || 0; // ans = 1
  #10 ans = rega && five; // ans = 1
  #10 ans = regb && rega; // ans = 1
  #10 ans = regc || 0; // ans = x
end
endmodule

Note 1: if it contains any ones, its logical value is true. If an operand contains all zeros, its logical value is false.

Note 2: if it is unknown (contains only zeros and/or unknown bits), its logical value is ambiguous.
Lexical Convention (5/10)

- **Operators**
  - *Shift Operator*
    - `>>` shift right
    - `<<` shift left

```verilog
module shift();
  reg [7:0] rega;
  reg [9:0] num;
  initial
    rega = 8'b0000_1100;
  initial begin
    #10 num = rega << 5; // num = 0110000000
    #10 num = rega >> 3; // num = 0000000001
  end
endmodule
```
Lexical Convention(6/10)

• Operators
  – Relational Operator
    • >   greater than
    • <   less than
    • >= greater than or equal
    • <= less than or equal

```
module relationals();
reg [3:0] rega, regb, regc;
reg val;
initial begin
  rega = 4'b0011;
  regb = 4'b1010;
  regc = 4'b0x10;
end
initial begin
  #10 val = regc > rega;    //val=x
  #10 val = regb < rega;    //val=0
  #10 val = regb >= rega;   //val=1
  #10 val = regb > regc;    //val=x
end
endmodule
```
Lexical Convention (7/10)

- **Operators**
  - *Equality Operator*
    - `==` logical equality
    - `!=` logical inequality

```verilog
module equalities();
reg [3:0] rega, regb, regc;
reg val;
initial begin
  rega = 4'\b0011;
  regb = 4'\b1010;
  regc = 4'\b0x10;
end
initial begin
  #10 val = regc > rega; // val = x
  #10 val = regb < rega; // val = 0
  #10 val = regb >= rega; // val = 1
  #10 val = regb > regc; // val = x
end
endmodule
```
Lexical Convention(8/10)

• Operators
  – *Conditional Operator*
    • $<\text{LHS}> = <\text{condition}> \; ? \; <\text{true_expression}> : <\text{false_expression}>$

```verilog
module mux(a, b, sel, out);
  input a, b, sel;
  output out;
  assign out = (sel == 1)? a : b;
endmodule
```
Lexical Convention (9/10)

• Comment
  – single-line comments with //
  – multiple-line comments with /* … */
Lexical Convention (10/10)

- **Concatenation**
  - allows you to select bits from different vectors and join them into a new vector
  - `{ }`

Note 1:  
- `rega[3:0]`
- `regb[3:0]`
- `new[7:0]`

Note 2: `a[4:0] = {a[3:0], 1'b0}; a = a<<1;`

```
module concatenation();
  reg regc, regd;
  reg [3:0] rega, regb;
  reg [7:0] new;
  reg [3:0] out;
  reg [5:0] val;
  initial begin
    rega = 8'b0000_0011;
    regb = 8'b0000_0100;
    regc = 1'b1;
    regd = 1'b0; end
  initial begin
    #10 new = {rega, regb};  // new = 8'b00110100
    #10 out = {4{regc}};  // out=4'b1111
    #10 out = {4{regd}};  // out=4'b0000
    #10 val = {regc, {2{regc,regd}}, regd};  // val = 6'b110100
  end
endmodule
```
Data Type (1/6)

- 4-Value Logic System in Verilog

- **Zero**
- **One**
- **Unknown**
- **High Impedance**

NOTE: In real hardware, this node will most be at either 1 or 0.
Data Type (2/6)

- Three Major Data type classes in Verilog
  - Nets
    - represent physical connection between devices
  - Registers
    - represent abstract storage elements
  - Parameters
    - are run-time constants
- Declaration Syntax
  - `<data_type> [<<MSB>>:<LSB>>] <list_of_identifier>`
Data Type (3/6)

- **Nets**
  
  - *wire* is the most common net type

NOTE: Verilog automatically propagates a new value onto a wire when the drivers on the wire change value. This means that whatever value is on the *or* gate will be automatically driven onto the wire *out*
Data Type(4/6)

• Registers
  – *reg* is the most common register type

**NOTE1:** A register holds its value until a new value is assigned to it

**NOTE2:** Apply values to registers from procedural blocks
Data Type(5/6)

• Vectors
  – the wire and register can be represented as a vector
    • wire [7:0]temp1;  →  8-bit bus
    • reg [7:0]temp2;

• Arrays
  – <array_name>[<subscript>]
    • integer temp1[7:0];  →  (8x32)bit
    • reg [7:0]temp2[0:1023]  →  Memory(1k, 1 byte)

NOTE: array is not well for the backend verification
module and1(out, in1, in2);
input in1, in2;
output out;
wire out;
assign out = in1 & in2;
endmodule

module and2(out, in1, in2);
input in1, in2;
output out;
reg out;
always@ (in1 or in2)
  out = in1 & in2;
endmodule
Data Assignment (1/3)

- Continuous Assignment
  - you can model **combinational logic** with continuous assignments, instead of using gates and interconnect wires
  - imply that whatever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS
  - assign [#delay] <wire_name> = <expression>

- wire [3:0]a;
  assign a = b + c; //continuous assignment
Data Assignment (2/3)

• Procedural Assignment

– assignment to register data types may occur within always, initial, task and function. These expressions are controlled by triggers which cause the assignment to evaluate

• reg CLK;
  always #10 CLK = ~CLK; //procedural assignment

• reg a, b;
  always@ (b) //procedural assignment with trigger
  a = ~b;

– procedural assignment consists blocking & non-blocking assignment
Data Assignment (3/3)

- Blocking Assignment ➔ Sequential !!

```verbatim
always@ (posedge clk)
begin
  b = a;
  c = b;
end
```

![Sequential Assignment Diagram]

- Non-Blocking Assignment ➔ Parallel !!

```verbatim
always@ (posedge clk)
begin
  b <= a;
  c <= b;
end
```

![Non-Blocking Assignment Diagram]
Event Control

• Combinational Circuit
  – @(a): act if signal ‘a’ changes
    • always@ (a) $c = a + 1$;
  – @(a or b): act if signal ‘a’ or ‘b’ changes
    • always@ (a or b) $c = a + b$;

• Sequential Circuit
  – @(posedge clk): act when rising edge of clk
    • always@ (posedge clk) $c = a + 1$;
  – @(negedge clk): act when falling edge of clk
    • always@ (negedge clk) $c = a + 1$;
Conditional Description (1/2)

• *if* and *if-else* Statements

```plaintext
if (index > 0) // beginning of outer if
    if (rega > regb) // beginning of the 1st inner if
        result = rega;
    else
        result = 0;
else
    if (index == 0)
        begin
            $display("Note : Index is zero");
            result = regb;
        end
    else
        $display("Note : Index is negative");
```

[Critical path diagram]

---

27
Conditional Description(2/2)

• **case Statement**

```verilog
module compute(result, rega, regb, opcode);
  input [7:0] rega, regb;
  input [2:0] opcode;
  output [7:0] result;
  reg [7:0] result;

  always@ (rega or regb or opcode)
  begin
    case (opcode)
      3'b000 : result = rega + regb;
      3'b001 : result = rega – regb;
      3'b100 : result = rega / regb;
      default : result = 8’bx;
    endcase
  end
endmodule
```
Register Description(1/2)

- **Informal Description**
  - combinational circuit and memory element is combined in one always block

```verilog
module MUX(out, sel, clk, reset, in1, in2);

input sel, clk, reset;
input [7:0]in1, in2;
output [7:0]out;
reg [7:0]out;

always@ (posedge reset or posedge CLK)
begin
  if(reset==1'b1) out <= 1'b0;
  else if(sel==1'b0) out <= in1;
  else out <= in2;
end
endmodule
```
Register Description (2/2)

• Normative Description
  – separate the combinational and sequential parts

```verilog
module MUX(out, sel, clk, reset, in1, in2);
  input sel, clk, reset;
  input [7:0] in1, in2;
  output [7:0] out;
  reg [7:0] out;
  wire c;

  // combinational part
  assign c = (sel) ? in1 : in2;

  // sequential part
  always@ (posedge reset or posedge CLK)
  begin
    if(reset) out <= 1'b0;
    else out <= c;
  end
endmodule
```

NOTE: it’s the better way
Synthesizable Verilog Code (1/2)

- Four data type can be synthesized
  - input, output, wire, reg
- 1-D data type is convenient for synthesis
  - reg [7:0] a;
  - reg [7:0]a[3:0];
- Synthesizable Verilog functions
  - assign, always block, called sub-module
- Synthesizable register description
  - always@ (posedge clk) always@ (negedge clk)
  - always@ (posedge clk or posedge reset) if(reset)..else..
  - always@ (negedge clk or posedge reset) if(reset)..else..
  - always@ (posedge clk or negedge reset) if(~reset)..else..
  - always@ (negedge clk or negedge reset) if(~reset)..else..
• What kind of Verilog code can’t be synthesized?

```verilog
always@ (posedge clk or negedge clk)
wire [7:0]a;
reg [3:0]b;
assign a[b] = 0;

always@ (posedge clk)
  out = out + 1;
always@ (negedge clk)
  out = a;

always@ (posedge clk or posedge reset)
  if(reset) out = 0;
  else out <= out + in;
```
Simulation Environment (1/5)

timescale
include verilog file
module
module_name;

- data type declaration
- Instantiate modules
- Applying simulation
- Display results

endmodule
Simulation Environment (2/5)

MUX.v
module MUX(OUT,CLK,SEL,RESET,IN1,IN2);
  input CLK,SEL,RESET,IN1,IN2;
  output OUT;
reg OUT;
wire C;
assign C = (SEL) ? IN1 : IN2;
always@ (posedge CLK)
  begin
    if(RESET) OUT <= 1'b0;
    else OUT <= C;
  end
endmodule

TEST_MUX.v
`timescale 1 ns/10 ps
`include "MUX.v"
module stimulus;

reg CLK,SEL,RESET,IN1,IN2;
wire OUT;
//connect port by ordering
MUX t(OUT,CLK,SEL,RESET,IN1,IN2);

//connect port by name
MUX t(OUT,CLK,SEL,RESET,IN1,IN2);
  .OUT(OUT),
  .CLK(CLK),
  .SEL(SEL),
  .RESET(RESET),
  .IN1(IN1),
  .IN2(IN2));
initial begin
  .......
end
endmodule

Your Design!!

Testbench!!
Simulation Environment (3/5)

- **Clock Generation**
  - Initial clk = 0;
  - `always #20 clk = ~clk;`

- **Display Simulation Result (Texture Format)**
  - `$display("%t,clk=%d in=%d out=%d\n",$time clk,in,out);`
  - `$monitor($time,"clk=%d out=%d\n", clk,out);`
  - `$time: current time`
  - `$finish: finish the simulation`

- **Dump a FSDB file for debug (debussy)**
  - `$fsdbDumpfile("file_name.fsdb");`
  - `$fsdbDumpvars;`

**NOTE:** `$` denotes Verilog system tasks and functions
Simulation Environment (4/5)

Format Specifiers:

%h  %o  %d  %b  %c  %s  %v  %m  %t
hex  octal  decimal  binary  ASCII  string  strength  module  time

Escaped Literals:

\t  \n  \\  \"
\<1-3 digit octal number>
tab  new line  backslash  double quote  ASCII representation of above
Simulation Environment (5/5)

- Compiler Directive
  - `define <macro_name> <macro_text>
    - `define delay #1
      and `delay and1(a1, a, sel);
  - `include "<file_name>"
    - `include "counter.v"
  - `timescale <time_unit> / <time_precision>
    - `timescale 1ns / 10ps
module stimulus;
reg CLK,SEL,RESET,IN1,IN2;
wire OUT;

parameter cycle = 5;

MUX m(.OUT(OUT),.CLK(CLK),.SEL(SEL),.RESET(RESET),.IN1(IN1),.IN2(IN2));

always #(cycle) CLK = ~CLK;

initial 
begin
$fsdbDumpfile("MUX.fsdb");
$fsdbDumpvars;
IN1 = 0; IN2 = 1; SEL = 0; RESET = 0; CLK = 0; //time = 0
#(cycle) IN2 = 0; //time = 5
#(cycle) IN2 = 1; SEL = 1; //time =10
#(cycle*2) IN1 = 1; //time =20
$display("%d", OUT);
$finish;
End
endmodule