Introduction to Verilog

INEl 4205 - Logic Circuits
M. Toledo - Spring 2008
Hardware description languages (HDL)

- High-level program-like description of logic circuit
- Developed for circuit simulation, now also used to synthesize circuits for ASIC’s and FPGA’s
- Two main hdl’s used today
  - VHDL - ADA-like syntax;
  - Verilog - C-like syntax;
Verilog

- Basic units: `module`
  - normally describe a single piece of hardware
  - has `declarations` that describe names & types of module’s inputs & outputs

```verilog
module fourbitadder(X, Y, S, C); // c++ style remarks
input [3:0] X, Y; // four-bit input vectors
output reg [3:0] S; // 4-bit sum vector
output reg C; // & one-bit carry
```
Gate-level modeling

- Textual description of schematic diagram
- Uses keywords like `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`, `buf`
- Four-level logic: 0, 1, high-impedance (z), unknown (x)
- Use `timescale <reference_time_unit>/ <time_precision>` to change the reference time scale and precision of a simulation. Example: `timescale 1ns/1p`
HDL Example 3-2

// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
  input A,B,C;
  output x,y;
  wire e;
  and #(30) g1(e,A,B);
  or #(20) g3(x,e,y);
  not #(10) g2(y,C);
endmodule

Fig. 3-37 Circuit to Demonstrate HDL
Modules and testing

- Gate primitives: or, nor, and, nand, xor, xnor
  - 1 output (1st terminal) and any no. of inputs

- Transmission gate primitives: buf, not (inverter); with active-high control input: bufif1, notif1; with active-low control input: bufifo, notifo

- Also switch primitives

- User-defined primitives are declared with the keyword `primitive` - not used frequently
Simulations are specified in a *testbench* which is a module

```verilog
module stimcircuit;
  reg A,B,C;
  wire x,y;
  circuit_with_delay cwd(A,B,C,x,y);
  initial
    begin
      A = 1'b0; B = 1'b0; C = 1'b0;
      #100
      A = 1'b1; B = 1'b1; C = 1'b1;
      #100
      $finish;
    end
  endmodule
```
Data Types

- Nets - represent structural connections
  - “wire” is the most common net type
- reg - represent variables used to store data
  - store the last value assigned to them until another assignment statement changes their value
Port Connection Rules

- **Inputs**: internally must always be type net, externally the inputs can be connected to variable reg or net (wire) type.

- **Outputs**: internally can be type net or reg, externally the outputs must be connected to a variable net type.

- **Inouts**: internally or externally must always be type net, can only be connected to a variable net type.

- **implicit type** (not explicitly declared) is one-bit wire
Fig. 3-38  Simulation Output of HDL Example 3-3
### Truth Table for Predefined Primitive Gates

<table>
<thead>
<tr>
<th>and</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xor</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
// Gate-level description of a 2-to-4-line decoder
// Figure 4-19
module decoder_g1 (A, B, E, D);
    input A, B, E;
    output [0:3] D;
    wire Anot, Bnot, Enot;
    not
        n1 (Anot, A),
        n2 (Bnot, B),
        n3 (Enot, E);
    nand
        n4 (D[0], Anot, Bnot, Enot),
        n5 (D[1], Anot, B, Enot),
        n6 (D[2], A, Bnot, Enot),
        n7 (D[3], A, B, Enot);
endmodule
bufif1 (OUT, A, control);
notif0 (Y, B, enable);

In the first example, input A is transferred to OUT when control = 1. OUT goes to z when control = 0. In the second example, output $Y = z$ when enable = 1 and output $Y = B'$ when enable = 0.
module muxtri (A, B, select, OUT);
  input A, B, select;
  output OUT;
  tri OUT;
  bufif1 (OUT, A, select);
  bufif0 (OUT, B, select);
endmodule

Fig. 4-32 2-to-1-Line Multiplexer with Three-State Buffers
Dataflow modelling

- Uses operators instead of gates
- Keyword `assign`
- Represent structural connections
- Drives wire data type; left-hand side must be wire
- Continuous assignments
- Used outside procedural blocks (`always` and `initial`)
- Syntax: `assign #(delay) net = expression`
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>binary addition</td>
</tr>
<tr>
<td>-</td>
<td>binary subtraction</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
</tr>
<tr>
<td>~</td>
<td>bit-wise NOT</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>{</td>
<td>concatenation</td>
</tr>
<tr>
<td>}</td>
<td>conditional</td>
</tr>
<tr>
<td>? :</td>
<td></td>
</tr>
</tbody>
</table>
HDL Example 3-4

//Circuit specified with Boolean expressions
module circuit_bln (x,y,A,B,C,D);
   input A,B,C,D;
   output x,y;
assign x = A | (B & C) | (~B & D);
assign y = (~B & C) | (B & ~C & ~D);
endmodule
// Dataflow description of a 2-to-4-line decoder
// See Fig. 4-19

module decoder_df (A,B,E,D);
    input A,B,E;
    output [0:3] D;
    assign D[0] = ~(~A & ~B & ~E),
                 D[1] = ~(~A & B & ~E),
                 D[2] = ~(A & ~B & ~E),
                 D[3] = ~(A & B & ~E);
endmodule
// Dataflow description of 4-bit adder

module binary_adder (A, B, Cin, SUM, Cout);
    input [3:0] A, B;
    input Cin;
    output [3:0] SUM;
    output Cout;
    assign {Cout, SUM} = A + B + Cin;
endmodule
/Dataflow description of a 4-bit comparator.

module magcomp (A,B,ALSB,AGTB,AEQB);
  input [3:0] A,B;
  output ALTB,AGTB,AEQB;
  assign ALTB=(A < B),
         AGTB = (A > B),
         AEQB = (A == B);
endmodule
HDL Example 4-6

// Dataflow description of 2-to-1-line multiplexer
module mux2x1_df (A, B, select, OUT);
  input A, B, select;
  output OUT;
  assign OUT = select ? A : B;
endmodule
32-bit adder

module adder(a, b, y);
    input [31:0] a, b;
    output [31:0] y;

    assign y = a + b;
endmodule
module testcircuit

reg TA, TB;
wire TC;
circuit cr (TA, TB, TC);

module circuit (A, B, C);
input A, B;
output C;

Fig. 4-33 Stimulus and Design Modules Interaction
Behavioral Model

-Declared using keywords
  - **initial**: Executes following block once
  - **always**: Executes repeatedly. Block is reevaluated only when signals in the header change
-Executes following statement, or block of statements within `begin` and `end`
Example: use of `always` with event control expressions

```plaintext
always @(A or B or Reset)
always @(posedge clock or negedge reset)
```

Following block or statement executes only if the expression is True.
• procedural assignment: occur within a `begin` or `always` block

• target output must be of `reg` type

• Blocking: use symbol `=`
  • are executed sequentially in the order they appear

• Non-blocking: use symbol `<=`
  • first `evaluate` all right-hand side, then `assign` to the left-hand side

---

Blocking

B = A ; // A copied into B
C = B+1 ; // A+1 copied into B

Non-blocking

B<= A ; // A <- B
C<=B+1; // C is Orig B + 1
//HDLC Example 5-1
//Description of D latch (See Fig.5-6)
module D_latch (Q,D,control);
   output Q;
   input D,control;
   reg Q;
   always @ (control or D)
   if (control) Q = D;
   //Same as: if (control = 1)
endmodule
// HDL Example 5-2
// D flip-flop
module D_FF (Q,D,CLK);
  output Q;
  input D,CLK;
  reg Q;
  always @ (posedge CLK)
    Q = D;
endmodule

// D flip-flop with asynchronous reset.
module DFF (Q,D,CLK,RST);
  output Q;
  input D,CLK,RST;
  reg Q;
  always @(posedge CLK or negedge RST)
    if (~RST) Q = 1'b0; // Same as: if (RST = 0)
    else Q = D;
endmodule
/HDL Example 5-3: T flip-flop from D flip-flop and gates

module TFF (Q,T,CLK,RST);
    output Q;
    input T,CLK,RST;
    wire DT;
    assign DT = Q ^ T ;
    DFF TF1 (Q,DT,CLK,RST); //Instantiate the D flip-flop
endmodule
// HDL Example 5-3: Continuation
// JK flip-flop from D flip-flop and gates

module JKFF (Q,J,K,CLK,RST);

   output Q;

   input J,K,CLK,RST;

   wire JK;

   assign JK = (J & ~Q) | (~K & Q);

   DFF JK1 (Q,JK,CLK,RST); // Instantiate D FF (prev slide)

endmodule
//HDL Example 5-4

// Functional description of JK flip-flop

module JK_FF
(J,K,CLK,Q,Qnot);
output Q,Qnot;
input J,K,CLK;
reg Q;
assign Qnot = ~ Q;

always @ (posedge CLK)
case ({J,K})
  2'b00: Q = Q;
  2'b01: Q = 1'b0;
  2'b10: Q = 1'b1;
  2'b11: Q = ~ Q;
endcase
endmodule
Fig. 5-16 State Diagram of the Circuit of Fig. 5-15
module Mealy_mdl (x, y, CLK, RST);

  input x, CLK, RST;
  output y;

  reg y;

  reg [1:0] Prstate, Nxtstate;

  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge CLK or negedge RST) //Initialize to state S0
    if (~RST) Prstate = S0; //Initialize to state S0
    else Prstate = Nxtstate; //Clock operations

  always @ (Prstate or x) //Find next state
    case (Prstate)
      S0: if (x) Nxtstate = S1;
      S1: if (x) Nxtstate = S3;
        else Nxtstate = S0;
      S2: if (~x) Nxtstate = S0;
      S3: if (x) Nxtstate = S2;
        else Nxtstate = S0;
    endcase

  always @ (Prstate or x) //Evaluate output
    case (Prstate)
      S0: y = 0;
      S1: if (x) y = 1'b0; else y = 1'b1;
      S2: if (x) y = 1'b0; else y = 1'b1;
      S3: if (x) y = 1'b0; else y = 1'b1;
    endcase

endmodule
Verilog Problem

• Write Verilog gate-level, dataflow and behavioral descriptions for the following circuits. Test the circuit by writing a testbench module and running a simulation.

  • a priority encoder such as the one shown in figure 4.23,

  • a four-bit adder-subtractor such as the one shown in figure 4.13, and

  • a sequence detector such as the one shown in figure 5.29