Digital Design with the Verilog HDL

Chapter 5 Behavioral Model - part 2

Binh Tran-Thanh

Department of Computer Engineering
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

February 28, 2022

Interacting Behaviors [1]

- Assignments can trigger other assignments
- Non-blocking assignments CAN trigger blocking assignments

```
always @(posedge clk) begin
    ...
    A <= B;
    ...
end
```

```
always @(A, C) begin
    ...
    D = A & C;
    ...
end
```

- In hardware, reflects that the output of the A flip flop or register can be the input to combinational logic
- When the FF changes value, that change must propagate through the combinational logic it feeds into

Interacting Behaviors [2]

```
always @(posedge clk or posedge rst) begin // behavior1
   if(rst) y1 = 0; //rst
   else y1 = y2;
end

always @(posedge clk or posedge rst) begin // behavior2
   if(rst) y2 = 1; // prst
   else y2 = y1;
end
```

- If behavior1 always first after rst, y1 = y2 = 1
- If behavior2 always first after rst, y2 = y1 = 0.
- Results are order dependent, ambiguous –race condition!
- This is why we don't use blocking assigns for flip-flops...

Interacting Behaviors [3]

```
always @(posedge clk or posedge rst) begin
   if(rst) y1 <= 0; //rst
   else y1 <= y2;
end

always @(posedge clk or posedge rst) begin
   if(rst) y2 <= 1; // prst
   else y2 <= y1;
end
```

- Assignments for y1 and y2 occur in **parallel**
- y1 = 1 and y2 = 0 after rst
- Values swap each clk cycle after the rst
- No race condition!

Synchronous/Asynchronous rst

- Synchronous: triggered by **clk** signal

```
module dff_sync(output reg Q,
                input D, clk, rst);
   always @(posedge clk) begin
      if(rst) Q <= 1'b0;
      else Q <= D;
   end
endmodule
```

- Asynchronous: also triggered by **rst** signal

```
module dff_async(output reg Q,
                 input D, clk, rst, en);
   always @(posedge clk, posedge rst) begin
      if(rst) Q <= 1'b0;
      else if (en) Q <= D;
   end
endmodule
```

What Happens Here?

```
module dff_BAD(reg output Q,
               input D, clk, rst);
   always @(posedge clk, rst) begin
      if(rst) Q <= 1'b0;
      else Q <= D;
   end
endmodule
```

Does this give incorrect behavior?

Datatype Categories

Net

- Represents a physical wire
- Describes structural connectivity
- Assigned to in continuous assignment statements
- Outputs of primitives and instantiated sub-modules

Variables

- Used in Behavioral procedural blocks
- Can represent:
  - Synchronous registers
  - Combinational logic

Variable Datatypes

- `reg` –scalar or vector binary values
- `integer` –32 or more bits
- `time` –time values represented in 64 bits (unsigned)
- `real` –double real values in 64 or more bits
- `realtime` -stores time as double real (64-bit +)
- Assigned value only within a behavioral block
- CANNOT USE AS:
  - Output of primitive gate or instantiated submodule
  - LHS of continuous assignment
  - Input or inout port within a module
- `real` and `realtime` initialize to 0.0, others to x...
  - Just initialize yourself!

Examples Of Variables

```
reg signed [7:0] A_reg; // 8-bit signed vector register
reg Reg_Array [7:0]; // array of eight 1-bit registers
integer Int_Array [1:100]; // array of 100 integers
real B, Real_Array [0:5]; // scalar & array of 6 reals
time Time_Array [1:100];  // array of 100 times
real timeD, Real_Time [1:5]; // scalar & array of 5
    realtimes
initial begin
   A_reg = 8'ha6;// Assigns all eight bits
   Reg_Array[7] = 1;// Assigns one bit
   Int_Array[3] = -1;// Assign integer -1
   B = 1.23e-4;// Assign real
   Time_Array[20] = $time;// Assigned by system call
   D = 1.25;// Assign real time
end
```

wire vs. reg (1/2)

- Same "value" used both as 'wire' and as 'reg'

```
module dff (q, d, clk);
  output reg q;// reg declaration,
  input wire d, clk;// wire declarations, since module
    inputs

  always @(posedge clk) begin
     q <= d;
     // remember q is declared reg and
     // d is declared wire
  end
endmodule
```

wire vs. reg (2/2)

- Same "value" used both as 'wire' and as 'reg'

```
module t_dff;
  wire q, clk;// now declared as wire
  reg d;// now declared as reg

  dff FF(q, d, clk);// why is d reg and q wire?
  clkgen myclk(clk);
  initial begin
    d = 0;
    #5 d = 1;
  end
endmodule
```

Signed vs. Unsigned

- Net types and **reg** variables unsigned by default
  - Have to declare them as signed if desired!

```
reg signed[7:0] signedreg;
wire signed[3:0] signedwire;
```

- All bit-selects and part-selects are **unsigned**
  - `A[6]`, `B[5:2]`, etc.
- `integer`, `real`, `realtime` are signed
- System calls can force values to be signed or unsigned

```
reg [5:0] A = $unsigned(-4);
reg signed [5:0] B = $signed(4'b1101);
```

Operators with Real Operands

Arithmetic

`Unary +/-`

`+ - * / **`

Relational

`> >= < <=` No others allowed!

Logical

`! && ||` No bit-selects!

Equality

`== !=` No part-selects!

Conditional

`? :`

Strings

- Strings are stored using properly sized registers

```
reg[12*8: 1] stringvar;// 12 character string
  // string assignment
  stringvar = "Hello World";
```

- Uses ASCII values
- Unused characters are filled with zeros
- Strings can be copied, compared, concatenated
- Most common use of strings is in testbenches

Memories (1/2)

- A memory is an array of n-bit registers

```
reg[15:0] mem_name [0:127];  //128 16-bit words
reg array_2D [15:0] [0:127]; // 2D array of 1-bit regs
```

- Can only access full word of memory

```
mem_name[122] = 35; // assigns word
mem_name[13][5] = 1;// illegal - works in simulation
array_2D[122] = 35; // illegal - causes compiler error
array_2D[13][5] = 1;// assigns bit
```

- Can use continuous assign to read bits

```
assign mem_val = mem[13]; // get word in slot 13
assign out = mem_val[5];  // get bit in slot 5 of word
assign dataout = mem[addr];
assign databit = dataout[bitpos];
```

Example: Memory

```
module memory(output reg[7:0] out, input[7:0] in, addr,
              input wr, clk, rst);
   reg [7:0] mem [0:255]; // memory cells
   reg [8:0] raddr;       // for reset memory

   always @(posedge clk) begin // WRITE operation
      if(rst) begin // synchronous reset!
         for(raddr = 0; raddr < 256; raddr = raddr + 1)
            begin mem[raddr] <= 8'd0; end
      end
      else if (wr)  begin
         mem[addr] <= in; // synchronous write
      end
   end

   always @(posedge clk) begin // READ operation
      out <= mem[addr]; // synchronous read
   end
endmodule
```

Control Statements

- Behavioral Verilog looks a lot like software
  - (risk! - danger and opportunity)
- Provides similar control structures
  - Not all of these actually synthesize, but some non-synthesizable statements are useful in **testbenches**
- What kinds synthesize?
  - `if`
  - `case`
  - `for` loops with constant bounds

if, else if, else

- Operator ? : for simple conditional assignments
- Sometimes need more complex behavior
- Can use **if-statement**!
  - Does not conditionally "execute" block of "code"
  - Does not conditionally create hardware!
  - It makes a **multiplexer** or similar logic
- Generally:
  - Hardware for all paths is created
  - All paths produce their results in parallel
  - One path's result is selected depending on the condition

Potential Issues With **if**

Can sometimes create long multiplexer chains

```
always @(select, a, b, c, d) begin
  out = d;
  if(select == 2'b00) out = a;
  if(select == 2'b01) out = b;
  if(select == 2'b10) out = c;
end

always @(a, b, c, d) begin
  if (a) begin
    if (b) begin
      if (c) out = d;
      else out = ~d;
    else out = 1;
  else out = 0;
end
```

if Statement: Flip-Flop Set/rst

```
moduled f_sr_behav(q, q_n, data, set, rst,clk);
  input data, set, clk, rst;
  output q, q_n;
  reg q;
  assign q_n= ~q;// continuous assignment

  /* Flip-flop with synchronous set/rst */
  always @(posedge clk) begin
    // Active-high set and rst
    if(rst) q <= 1'b0;
    else if(set) q <= 1'b1;
    else q <= data;
  end
endmodule
```

Does set or rst have priority?

`case` Statements

- Verilog has three types of case statements:
  - `case, casex, and casez`
- Performs bitwise match of expression and case item
  - Both must have same bitwidth to match!
- `case`
  - Can detect x and z! (only in simulation)
- `casez`
  - Can detect x(In simulation)
  - Uses z and ? as wildcard bits in case items and expression
- `casex`
  - Uses x, z, and ? as wildcard bits in case items and expression

Using **case** To detect **x** And **z**

- Only use this functionality in a testbench; it won't synthesize!
- Example taken from Verilog-2001 standard;
- Cannot be synthesized

```
case(sig)
  1'bz: $display("Signal is floating.");
  1'bx: $display("Signal is unknown.");
  default: $display("Signal is %b.", sig);
endcase
```

casex Statement

- Uses x, z, and ? as single-bit wildcards in case item and expression
- Uses only the first match encountered –Inherent **priority**!
- Treats x, z, and ? as don't care

```
always @(code) begin
  casex(code)// case expression
    2'b0?: control = 8'b00100110; //case item 1
    2'b10: control = 8'b11000010; //case item 2
    2'b11: control = 8'b00111101; //case item 3
  endcase
end
```

What is the output for code = 2'b01?
What is the output for code = 2'b1x?
casex construct is often used to describe logic with priority

casez Statement

- Uses z, and ? as single-bit wildcards in case item and expression
- Use z as don't care, x is a value
- Uses first match encountered

```
always @(code) begin
  casez(code)
    2'b0?: control = 8'b00100110;    // item 1
    2'bz1: control = 8'b11000010;    // item 2
    default: control = 8b'xxxxxxxx; // item 3
  endcase
end
```

Adding a default case statement is a great way to avoid inferring latches and make debugging easier!
What is the output for code = 2b'01?
What is the output for code = 2b'zz?

"Don't Care" Assignment With case

- Sometimes we know not all cases will occur
- Can "don't care" what is assigned

```
always @(state, b) begin
  case(state)                        // state is "expression"
    2'd0: next_state = 2'd1;          // 2'd0 is case item
    2'd1: next_state = 2'd2;          // ordering implied
    2'd2: if(b) next_state = 2'd0;
          else next_state = 2'd1;
    default: next_state = 2'dx;// for all other states,
  endcase                            // don't care what is
                                     // assigned
end
```

"Don't Care" Assignment With if

- Some combinations don't happen
- Can "don't care" what is assigned
  - For the real hardware, the actual value assigned is either 1 or 0 (or a vector of 1s and/or 0s), but the **synthesizer** gets to choose

```
always @(a, b, c) begin
  if(a && b)
     d = c;
  else if(a)
     d = ~c;
  else
     d = 1'bx;
end
```

For what inputs does the "don't care" value of d happen?

Inferring Latches

- Earlier we saw how to explicitly create latches
- We can also implicitly "infer" latches
  - Often this is unintentional and the result of poor coding, accidentally creating a latch instead of comb. logic
  - Unintentional latches are a leading cause of simulation and synthesis results not matching!
- Two common ways of inferring latches
  - Failing to fully specify all possible cases when using if/else, case, or conditional assignment when describing **combinational** logic
  - Failing to assign values to each variable in **all** cases

Unintended Latches [1] (BAD)

Always use else or provide "default value" when using if to describe combinational logic!

```
always @(en, a, b) begin
  if(en) c = a + b; // latch! - What about en= 1'b0?
end

always @(en, a, b) begin
  c = 8'b0;           // Fixed! Assign default value to C
  if(en) c = a + b; // Will be overwritten if en=1'b1
end

always @(en, a, b) begin
  if(en) c = a + b; // Fixed! Add an Else statement
  else c = 8'b0;    // Could have used a different value
end
```

Unintended Latches [2] (BAD)

Make sure that no matter what the value of your case selector is, if a variable is assigned in one case, it must be assigned in EVERY case.

```
always @(sel, a, b)
case(sel)
  2'b00 : begin
    out1 = a + b;
    out2 = a - b;
  end
  2'b01 : out1 = a + b; //Latch! What about out2's value?
  2'b10 : out2 = a - b; //Latch! What about out1's value?
  //Latch! What about out1 and out2's values for sel=2'b11?
endcase
```

How could we modify this code to remove the latches?

Mux With if...else if...else

What happens if we forget select in the trigger list?
What happens if select is 2'bxx?

```
module Mux_4_32_if (output[31:0] oData,
                    input[31:0] iData3, iData2, iData1, iData0,
                    input[1:0] select, input enable);
   reg[31: 0] mux_out;
   // add the enable functionality
   assign oData = enable ? mux_out : 32'bz;
   // choose between the four inputs
   always @(iData3 or iData2 or iData1 or iData0 or select)
      if      (select == 0) mux_out = iData0;
      else if(select == 1) mux_out = iData1;
      else if(select == 2) mux_out = iData2;
      else                 mux_out = iData3;
endmodule
```

Mux With case

Case statement implies priority unless use parallel_case pragma
What happens if select is 2'bxx?

```
module Mux_4_32_if (output[31:0] oData,
                    input[31:0] iData3, iData2, iData1, iData0,
                    input[1:0] select, input enable);
   reg[31: 0] mux_out;
   // add the enable functionality
   assign oData = enable ? mux_out : 32'bz;
   // choose between the four inputs
   always @(iData3 or iData2 or iData1 or iData0 or select)
      case(select)
         2'd0: mux_out = iData0;
         2'd1: mux_out = iData1;
         2'd2: mux_out = iData2;
         2'd3: mux_out = iData3;
      endcase
endmodule
```

Encoder With if...else if...else

```
module encoder (output reg[2:0] Code,
                input [7:0] Data);

   always @(Data) begin // encode the data
      if      (Data == 8'b00000001) Code = 3'd0;
      else if(Data == 8'b00000010) Code = 3'd1;
      else if(Data == 8'b00000100) Code = 3'd2;
      else if(Data == 8'b00001000) Code = 3'd3;
      else if(Data == 8'b00010000) Code = 3'd4;
      else if(Data == 8'b00100000) Code = 3'd5;
      else if(Data == 8'b01000000) Code = 3'd6;
      else if(Data == 8'b10000000) Code = 3'd7;
      else Code = 3'bxxx; //invalid, so don't care
   end

endmodule
```

Encoder With case

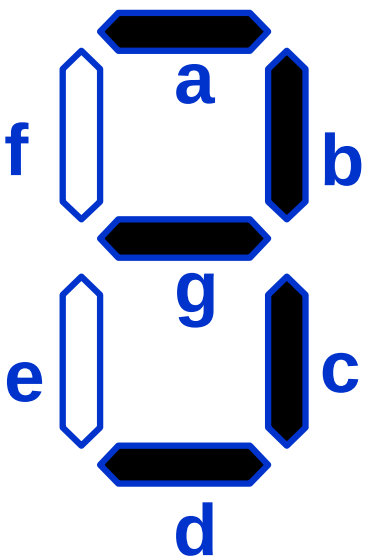
```
module encoder (output reg[2:0] Code, input [7:0] Data);

   always @(Data) begin // encode the data
      case(Data)
        8'b00000001 : Code = 3'd0;
        8'b00000010 : Code = 3'd1;
        8'b00000100 : Code = 3'd2;
        8'b00001000 : Code = 3'd3;
        8'b00010000 : Code = 3'd4;
        8'b00100000 : Code = 3'd5;
        8'b01000000 : Code = 3'd6;
        8'b10000000 : Code = 3'd7;
        default : Code = 3'bxxx; // invalid, so don't care
      endcase
   end
endmodule
```

Priority Encoder With casex

```
module priority_encoder (output reg[2:0] Code,
                         output valid_data,
                         input[7:0] Data);
  assign valid_data = |Data; // "reduction or" operator
  always @(Data) // encode the data
    casex(Data)
      8'b1xxxxxxx : Code = 7;
      8'b01xxxxxx : Code = 6;
      8'b001xxxxx : Code = 5;
      8'b0001xxxx : Code = 4;
      8'b00001xxx : Code = 3;
      8'b000001xx : Code = 2;
      8'b0000001x : Code = 1;
      8'b00000001 : Code = 0;
      default: Code = 3'bxxx; // should be at least one 1,
    don't care
    endcase
endmodule
```

Seven Segment Display




```
module Seven_Seg_Display(Display, BCD, Blanking);
   output reg[6: 0] Display;// abc_defg
   input[3: 0] BCD;
   input Blanking;
   parameter BLANK = 7'b111_1111; // active low
   parameter ZERO  = 7'b000_0001; // h01
   parameter ONE   = 7'b100_1111; // h4f
   parameter TWO   = 7'b001_0010; // h12
   parameter THREE = 7'b000_0110; // h06
   parameter FOUR  = 7'b100_1100; // h4c
   parameter FIVE  = 7'b010_0100; // h24
   parameter SIX   = 7'b010_0000; // h20
   parameter SEVEN = 7'b000_1111; // h0f
   parameter EIGHT = 7'b000_0000; // h00
   parameter NINE  = 7'b000_0100; // h04
   always@ (BCD or Blanking)
      if(Blanking) Display = BLANK;
      else
         case(BCD)
            4'd0:Display = ZERO;
            4'd1:Display = ONE;
            4'd2:Display = TWO;
            4'd3:Display = THREE;
            4'd4:Display = FOUR;
            4'd5:Display = FIVE;
            4'd6:Display = SIX;
            4'd7:Display = SEVEN;
            4'd8:Display = EIGHT;
            4'd9:Display = NINE;
            default:Display = BLANK;
         endcase
endmodule
```

Ring Counter

```
module ring_counter (output reg[7: 0] count,
                     input enable, clk, rst);

   always @(posedge rst or posedge clk) begin
      if (rst == 1'b1) begin count <= 8'b0000_0001; end
      else if (enable == 1'b1)begin
         case (count)
            8'b0000_0001: count <= 8'b0000_0010;
            8'b0000_0010: count <= 8'b0000_0100;
            ...
            8'b1000_0000: count <= 8'b0000_0001;
            default: count <= 8'bxxxx_xxxx;
         endcase
      end
   end
endmodule
```

Ring Counter

```
module ring_counter (output reg[7: 0] count,
                     input enable, clk, rst);

   always @(posedge rst or posedge clk) begin
      if (rst == 1'b1) begin // reset
         count <= 8'b0000_0001;
      end
      else if(enable == 1'b1) begin // ring counter
         count <= {count[6: 0], count[7]};
      end
   end
endmodule
```

Register With Parallel Load

```
module Par_load_reg4 (output reg [3: 0] Data_out,
                      input [3: 0] Data_in,
                      input load, clk, rst);

   always @(posedge rst or posedge clk) begin
      if(rst == 1'b1) begin //reset
         Data_out <= 4'b0;
      end
      else if(load == 1'b1) begin // load
         Data_out <= Data_in;
      end
      else begin // keep the same values
         Data_out <= Data_out;
      end
   end
endmodule
```

Rotator

```
module rotator (output reg[7: 0] Data_out,
                input[7: 0] Data_in,
                input load, clk, rst);

   always @(posedge rst or posedge clk) begin
      if(rst == 1'b1) begin // reset
        Data_out <= 8'b0;
      end
      else if(load == 1'b1) begin // load
         Data_out <= Data_in;
      end
      else begin // rotate operation
         Data_out <= {Data_out[6: 0], Data_out[7]};
      end
   end
endmodule
```

Shift Register

```
module Shift_Reg(output reg[3: 0] oData, // out data
                 output oMSB, oLSB, // out MSB, LSB
                 input[3: 0] iData, input [1:0] sel,
                 input iMSB, iLSB, clk, rst);
  assign oMSB = oData[3];
  assign oLSB = oData[0];
  always @(posedge clk) begin
    if(rst) oData <= 0; // reset
    else case(sel) // select operation
      2'd0: oData <= oData;// Hold
      2'd1: oData <= {iMSB, oData[3:1]}; // Shift right
      2'd2: oData <= {oData[2: 0], iLSB};// Shift left
      2'd3: oData <= iData; // Parallel Load
    endcase
  end
endmodule
```

Register File

Are the reads and writes synchronous or asynchronous?

```
module Register_File(output[15:0] oData1, oData2, // out
    input[15:0] iData, // data in
    input[2:0] rAddr1, rAddr2, wAddr, // read/write address
    input enWrite, clk); // write enable, clock

    // 16-bit by 8-word memory declaration
    reg[7:0] Reg_File [0:15];

    always @(posedge clk) begin
        if(enWrite) begin // WRITE opetation
            Reg_File [wAddr] <= iData;
        end
        oData1 <= Reg_File[rAddr1]; // READ opetation
        oData2 <= Reg_File[rAddr2]; // READ opetation
    end
endmodule
```