EE Summer Camp - 2006
Verilog Lab

Objective : Simulation of basic building blocks of digital circuits in Verilog using ModelSim simulator

Points to be kept in mind:

- For getting points in any question, you will have to simulate the testbenches and show us the waveform files for each question on Sunday, 14th May, at 10:30 AM, in the VLSI Lab.
- Consultation is allowed for questions 1 and 3 amongst students.
- Consultation for questions 2, 4 and 5 is only allowed with us.
- Please do not attempt to copy from each other or from internet. We would very much like to personally clear any doubts that you have, just mail us. It would be highly beneficial to consult your digital electronics textbooks like Taub & Scilling.

1. Learn use of ModelSim simulator by writing the Verilog code to simulate a half adder; where a, b are 1-bit inputs and sum,carry are 1-bit outputs. A sample code and its associated test bench is given below. (4 points)

```
module
halfadder(a,b,sum,carry);
input a,b;
output sum, carry;
wire sum, carry;

assign sum = a^b;    // sum bit
assign carry = (a&b) ;
//carry bit

endmodule
```

```
module main;
reg  a, b;
wire sum, carry;

halfadder add(a,b,sum,carry);
always @(sum or carry)
begin
   $display("time=%d:%b + %b = %b,
carry = %b\n",$time,a,b,sum,carry);
end

initial
begin
   a = 0; b = 0;
   #5
   a = 0; b = 1;
   #5
   a = 1; b = 0;
   #5
   a = 1; b = 1;
end
endmodule
```

2. Write the verilog code for a Full Adder, that takes in three 1-bit inputs, a, b and carryin, and gives sum and carryout 1-bit outputs. Write the code for a testbench for the adder, and give appropriate inputs to test all possible combinations. (6 points)

3. Simulate the code for the D flipflop discussed in class, and given below.(4 points)

```
`define TICK #2     //Flip-flop
delay

module dflipflop (d, clk, reset,
q);
input d, clk, reset;
output q;
reg q;

always @ (posedge clk or posedge
reset) begin
  if (reset) begin
    q <= 0;
  end
  else begin
    q <= `TICK d;
  end
end

endmodule
```

```
module main;
reg d, clk, rst;
wire q;
dflipflop dff (d, clk, rst, q);

//Always at rising edge of clock
display the signals
always @(posedge clk)begin
   $display("d=%b, clk=%b, rst=%b,
q=%b\n", d, clk, rst, q);
end

//Module to generate clock with
period 10 time units
initial begin
   forever begin
      clk=0;
      #5
      clk=1;
      #5
      clk=0;
   end
end

initial begin
   d=0; rst=1;
   #4
   d=1; rst=0;
   #50
   d=1; rst=1;
   #20
   d=0; rst=0;
end
endmodule
```

4. Write the verilog code for a JK Flipflop, and its testbench. Use all possible combinations of inputs to test its working (6 points)

5. Write the hardware description of a 4-bit PRBS (pseudo-random Binary sequence) generator using a linear feedback shift register and test it. The way it is implemented is as given in http://en.wikipedia.org/wiki/Linear_feedback_shift_register Please bear in mind that you have to make just a 4-bit PRBS generator. You are free to choose your own polynomial for the generator. The suggested skeleton file is written below: (10 points)

   Note: Please bear in mind that the shift register should not have all-zeros to start of with for the PRBS generator to produce the desired output. Make suitable adjustments.

```
module prbs (rand, clk, reset)

input clk, reset;
output  rand;
……

……

endmodule
```

Some Additional Information:

- Use different folders for each problem to avoid confusion.
- Constant Vectors are specified as: 4'b1011 This says that the data is of 4 bits, and its representation in binary is 1011.
- To concatenate two vectors use this format:
  A = 3'b101;
  B = 4'b1001;
  C = {A,B}; // This means that C will now be 7'b1011001.
  D = {A[0], B[1], B[2], 2'b11}; //This means D will now be 5'b 10011
- Use registers whenever you need to store values or when the signal is not being driven continuously by a combinatorial circuit.
- Try to think of what the behavior of the module should be and then write the verilog code.
- We will award partial credit for incomplete files. So be sure to document what you are doing to show us at the time of evaluation.
- Please approach us in case of any doubt whatsoever.

5.2 Write the hardware description of a 8-bit register with shift left and shift right modes of operation and test its operation. The suggested skeleton file has been written below: (10 points)

```
module slsr(sl, sr, din, clk, reset,Q);
input sl, sr, din, clk, reset;
output [7:0] Q;

endmodule
```

5.3 Write the hardware description of a 8-bit register with parallel load and shift left modes of operation and test its operation. The suggested skeleton file has been written below: (10 points)

```
module regPLSL (din, PLdata, PL, SL, q, clk, reset);
input din, SL, PL, clk, reset;
input       [7:0] PLdata;
output  [7:0]     q;

endmodule
```

5.4 Write the hardware description of a 4-bit down counter and test it. The suggested skeleton file has been written below: (10 points)

```
module counter(count, clk, reset);
input clk, reset;
output  [3:0]     count;

endmodule
```

5.5 Write the hardware description of a 4-bit mod-13 counter and test it. The suggested skeleton file has been written below: (10 points)

```
module counter(count, clk, reset);
input clk, reset;
output  [3:0]     count;

endmodule
```

5.6 Write the hardware description of a 4-bit adder/subtractor and test it. An adder/subtractor is a piece of hardware that can give the result of addition or subtraction of the two numbers based on a control signal. Assume that the numbers are in 2's complement notation. Please keep in mind that this is a combinatorial circuit. The suggested skeleton file to start with is given below: (10 points)

```
module addsub (a, b, sel, res);
input [3:0] a, b;
input sel;
output  [3:0] res;

endmodule
```

EE Summer Camp 2006
Verilog Lab Solution File

Pointers

- We were primarily teaching you how to use ModelSim to make simple digital circuits through this lab.
- We have given a behavioral solution for all the questions. However, working structural solutions also deserve full credit.
- Equal credits have been allotted for the file and the testbench made.

2. Full Adder

fulladder.v

```
module fulladder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
wire sum,carry;

assign sum=a^b^c;    // sum bit
assign carry=((a&b) | (b&c) | (a&c));   //carry bit

endmodule
```

testfulladder.v

```
module main;
reg  a, b, c;
wire sum, carry;

fulladder add(a,b,c,sum,carry);
always @(sum or carry)
begin
   $display("time=%d:%b + %b + %b = %b, carry =
%b\n",$time,a,b,c,sum,carry);
end

initial
begin
   a = 0; b = 0; c = 0;
   #5
   a = 0; b = 1; c = 0;
   #5
   a = 1; b = 0; c = 1;
   #5
   a = 1; b = 1; c = 1;
end

endmodule
```

4. JK Flipflop

jkflop.v

```
`define TICK #2                //Flip-flop time delay 2 units

module jkflop(j,k,clk,rst,q);
input j,k,clk,rst;
output q;
reg q;
always @(posedge clk)begin
   if(j==1 & k==1 & rst==0)begin
      q <=`TICK ~q;         //Toggles
   end
   else if(j==1 & k==0 & rst==0)begin
      q <= `TICK 1;        //Set
   end
   else if(j==0 & k==1)begin
      q <= `TICK 0;       //Cleared
   end
end
always @(posedge rst)begin
   q <= 0; //The reset normally has negligible delay and hence ignored.
end
endmodule
```

testjkflop.v

```
module main;
reg j,k,clk,rst;
wire q;
jkflop jk(j,k,clk,rst,q);
//Module to generate clock with period 10 time units
initial begin
   forever begin
      clk=0;
      #5
      clk=1;
      #5
      clk=0;
   end
end
initial begin
   j=0; k=0; rst=1;
   #4
   j=1; k=1; rst=0;
   #40
   rst=1;
   #10
   j=0; k=1;
   #10
   rst=0;
   #10
   j=1; k=0;
end
endmodule
```

5.1 PRBS Generator

prbs.v

```
module prbs (rand, clk, reset);
input clk, reset;
output rand;
wire rand;

reg [3:0] temp;

always @ (posedge reset) begin
    temp <= 4'hf;
end

always @ (posedge clk) begin
    if (~reset) begin
        temp <= {temp[0]^temp[1],temp[3],temp[2],temp[1]};
    end
end

assign rand = temp[0];
endmodule
```

testprbs.v

```
module main;
reg clk, reset;
wire rand;

prbs pr (rand, clk, reset);

initial begin
    forever begin
        clk <= 0;
        #5
        clk <= 1;
        #5
        clk <= 0;
    end
end

initial begin
    reset = 1;
    #12
    reset = 0;
    #90
    reset = 1;
    #12
    reset = 0;
end

endmodule
```

5.2 Shift Left-Shift Right Register

slsr.v

```
module slsr(sl, sr, din, clk, reset,Q);
input sl, sr, din, clk, reset;
output [7:0] Q;
reg [7:0] Q;

always @ (posedge clk) begin
    if (~reset) begin
        if (sl) begin
            Q <= #2 {Q[6:0],din};
        end
        else if (sr) begin
            Q <= #2 {din, Q[7:1]};
        end
    end
end

always @ (posedge reset) begin
    Q<= 8'b00000000;
end

endmodule
```

testslsr.v

```
module main;
reg clk, reset, din, sl, sr;
wire [7:0] q;
slsr  slsr1(sl, sr, din, clk,
reset, q);

initial begin
    forever begin
        clk <= 0;
        #5
        clk <= 1;
        #5
        clk <= 0;
    end
end

initial begin
    reset = 1;
    #12
    reset = 0;
    #90
    reset = 1;
    #12
    reset = 0;
end

initial begin
    sl = 1;
    sr = 0;
    #50
    sl = 0;
    #12
    sr = 1;
end

initial begin
    forever begin
       din = 0;
       #7
       din = 1;
       #8
       din = 0;
   end
end

endmodule
```

5.3 Parallel Load -Shift Left Register

plsl.v

```
module plsl(pl, sl, slin, Din, clk, reset, Q);
input pl, sl, slin, clk, reset;
input [7:0] Din;
output [7:0] Q;
reg [7:0] Q;

always @ (posedge clk) begin
    if (~reset) begin
        if (sl) begin
            Q <= `TICK {Q[6:0],slin};
        end
        else if (pl) begin
            Q <= `TICK Din;
        end
    end
end

always @ (posedge reset) begin
    Q <= 8'b00000000;
end

endmodule
```

testplsl.v

```
module main;
reg clk, reset, slin, sl, pl;
reg [7:0] Din;
wire [7:0] q;

plsl  plsl1(pl, sl, slin, Din,
clk, reset, Q);

initial begin
    forever begin
        clk <= 0;
        #5
        clk <= 1;
        #5
        clk <= 0;
    end
end

initial begin
    reset = 1;
    #12
    reset = 0;
    #90
    reset = 1;
    #12
    reset = 0;
end

initial begin
    sl = 1;
    pl = 0;
    Din = 8'h42;
    #50
    sl = 0;
    #12
    pl = 1;
    #5
    Din = 8'h21;
    #20
    pl = 0;
    sl = 1;
end

initial begin
    forever begin
       slin = 0;
       #7
       slin = 1;
       #8
       slin = 0;
   end
end

endmodule
```

5.4 4 Bit Down Counter

downCntr.v

```
`define TICK #2
module downCntr(clk, reset, Q);
input clk, reset;
output [3:0] Q;
reg [3:0] Q;

//Behavioral Code for a Down Counter
always @ (posedge clk) begin
    if (~reset) begin
        Q <= `TICK Q-1;
    end
end

always @ (posedge reset) begin
    Q <= 4'b0000;
end

endmodule
```

testDnCntr.v

```
module main;
reg clk, reset;
wire [3:0] Q;

downCntr dnCntr1(clk, reset, Q);

initial begin
    forever begin
        clk <= 0;
        #5
        clk <= 1;
        #5
        clk <= 0;
    end
end

initial begin
    reset = 1;
    #12
    reset = 0;
    #170
    reset = 1;
    #12
    reset = 0;
end

endmodule
```

5.5 4 Bit Mod 13 Counter

mod13Cntr.v

```
`define TICK #2
module mod13Cntr(clk, reset, Q);
input clk, reset;
output [3:0] Q;
reg [3:0] Q;

//Behavioral Code for a Mod-13 counter
always @ (posedge clk) begin
    if (~reset) begin
        if (Q == 4'b1100) begin
           Q <= `TICK  4'b0;
       end
       else begin
           Q <= `TICK Q+1;
       end
    end
end

always @ (posedge reset) begin
    Q <= 4'b0000;
end

endmodule
```

testmod13Cntr.v

```
module main;
reg clk, reset;
wire [3:0] Q;

downCntr dnCntr1(clk, reset, Q);

initial begin
    forever begin
        clk <= 0;
        #5
        clk <= 1;
        #5
        clk <= 0;
    end
end

initial begin
    reset = 1;
    #12
    reset = 0;
    #170
    reset = 1;
    #12
    reset = 0;
end

endmodule
```

5.6 Adder/Subtractor

addSub.v

```
module addSub(A, B, sel, Result);
input sel;
input [3:0] A,B;
output [3:0] Result;
wire [3:0] Result;

assign Result = (sel)? A + B : A - B;

endmodule
```

testAS.v

```
module main;

reg   [3:0] A, B;
reg   sel;
wire  [3:0] Result;

addSub as1(A, B, sel, Result);

initial begin
    A = 4'b0001;
    B = 4'b1010;
end


initial begin
    forever begin
        #10
        A = A + 1'b1;
        B = B + 1'b2;
    end
end

initial begin
    sel = 1;
    #200
    sel = 0;
end

endmodule
```

EE Summer Camp - 2006
Verilog Lab Clarifications

1. **Non-blocking assignment & Blocking assignment**: A simple explanation would be that a non-blocking assignment (`<=`) actually is used when the order of assignment does not matter (or rather not defined) and the statements need to be concurrent in execution. These assignments are also useful in specifying the `` `TICK `` delay for flops with ease without blocking concurrent statements hence named non-blocking. The blocking assignment (`=`) is used when we need the operations to follow one after the other within a block (from begin to end). (We do understand that there was some confusing in the demo about the concurrency.)
2. **Wire and reg**: The basic difference between wire and register is that a wire needs to be 'driven' at all times (driven can be either from a register or through a Boolean assign statement that continuously keeps assigning values) and a reg stores the values when changed. For example, in the Dflipflop demo code we need q to be a reg because it is not being 'driven' continuously and needs to store values between clock edges (positive) i.e. in the period between the edges the value needs to be stored. On the other hand in the code of halfadder, the sum being the output of a combinatorial circuit is continuously driven. Hence it is a wire.
3. You may remove the **`display`** statement. There is no use of this statement when we have the waveform viewer. Using it in the demo was just an illustration.
4. Useful statement:
   ```
   assign out = sel ? 0:1;
   ```
   assigns the value of **`out`** as **`0`** if **`sel`** is **`1`** and the value of **`out`** as **`1`** when **`sel`** is **`0`**.
5. The behavior of your circuit to 'abnormal' inputs is left to your discretion. You can assume anything as long as you can explain it in the demo.
6. Codes copied from internet or from each other will be considered cheating.

EE Summer Camp - 2006
Verilog Lab Clarifications 2

1. Use of multiple if else blocks in checking conditions.
   a. Nested

```
if (condition) begin
    if(subcondition) begin
         ...
    end
    else begin
         ...
    end
end
else begin
    ...
end
```

   b. else if blocks

```
if(condition) begin
    ...
end
else if(condition2) begin
    ...
end
else begin
    ...
end
```

   The conditions can contain Boolean operations like '&'.

2. To specify case statement for more than one signal, concatenation can be used. (Refer to the assignment additional information for details on how concatenation is done).

```
case ({a,b})
      {1'b0,1'b1}: ... ; // when a is 0 and b is 1
      ...
      default: ... ;
endcase
```

   Note: The length in bits of the constants needs to be specified for concatenating.

EE Summer Camp - 2006
Verilog Lab Clarifications 3

Always Block

Syntax:

```
always @ (signal1 or signal2 or signal3) begin
      Block
end
```

- This block implies, the processor should schedule `Block` whenever there is a change/transition in any of the three signals – signal1, signal2 or signal3.
- Here the sensitivity list of this always block consists of these three signals.
- We usually do not use logical operations inside the sensitivity list. Instead, condition checking is done inside the `Block`.
- Here, the verilog scheduler monitors all the three signals individually. Whenever there is a change it enters the `Block`.
- An example of the condition checking is seen here:

```
always @ (signal1 or signal2 or signal3) begin
      if (signal1 == 1 and signal2 == 0)begin
            Block1
      end

      else if (signal3 == 0) begin
            Block2
      end

      else begin
            Block3
      end
end
```