Hardware Description Languages: Verilog

- Syntax, Loops, Behavioral modeling

Verilog Continuous Assignment

- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the RHS operands changes
- Target is *NEVER* a reg

```
assign A = X | (Y & ~Z);
```

use of Boolean operators
(~ for bit-wise, ! for logical negation)

```
assign B[3:0] = 4'b01XX;
```

bits can take on four values
(0, 1, X, Z)

```
assign #3  S[3:0] = A[3:0] + B[3:0] + Cin;
```

delay of performing computation

use of arithmetic operator

Initial block

- Starts at time 0, executes exactly once during a simulation
- Multiple initial blocks--> each block starts to execute concurrently at time 0

```
module stimulus;
initial
begin
#5 a = 1'b1;
#25 b = 1'b0;
end
endmodule
```

Simple Behavioral Model: the `always` block

- `always` block
  - starts at time 0 and executes continuously in a loop
  - Always waiting for a change to a trigger signal
  - Then executes the body
  - Signal assignment is always made to a reg

```
module and_gate (out, in1, in2);
  input   in1, in2;
  output  out;
  reg     out;

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

A Verilog register is needed because of assignment in always block

Specifies when block is executed
I.e., triggered by which signals

`always` Block

- Procedure that describes the function of a circuit
  - Can contain many statements including `if`, `for`, `while`, `case`
  - Statements in the `always` block are executed *sequentially*
    - (Continuous assignments <= are executed in *parallel*)
  - `begin`/`end` used to group statements

Event control @

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
```

```
always @( reset or clock or d)
                                //Wait for reset or clock or d to
change
```

```
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

Verilog `if`

- Same as C if statement

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;

endmodule
```

Verilog `if`

- Another way

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0) Y = A;
      else             Y = B;
    else
      if (sel[1] == 0) Y = C;
      else             Y = D;
endmodule
```

Verilog case

- Sequential execution of cases
  - Only first case that matches is executed (implicit break)
  - Default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;        // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                  // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

Conditions tested in top to bottom order

Verilog case

- With default

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:     Y = 3'bXXX;    // Don't care when input is not 1-hot
    endcase
endmodule
```

Verilog `for`

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

integer i;                // Temporary variables for program only
reg [7:0] test;

  always @(A) begin
    test = 8b'00000001;
    Y = 3'bX;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = i;
       test = test << 1;
    end
  end
endmodule
```

while

```
initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                 $display("Count = %d", count);
                count = count + 1;
        end
end
```

Other topics

Non blocking, blocking statements

Functions, files

Practice - 1

- Simulate the dff.v (it contains the testbench too).
  - Important: Once you run ./a.out, it will keep running infinitely, because it is in an always block. You need to hit Ctrl +Z to stop it, else, the vcd will become a large file and will never end.
  - Note that in the testbench, the clock and d input are being toggled using the following statements. See the comments in the code

```
always
#3 clk=~clk;
always
#5 d=~d;
```

- Add a synchronous reset input to this D-Flip Flop. You will need to change the testbench to simulate it
- Create another Verilog module to make this an asynchronous reset. Note the difference between the two types of reset

Practice - 2

- Simulate the counter.v (it contains the testbench too)
  - Important: Once you run ./a.out, it will keep running infinitely, because it is in an always block. You need to hit Ctrl +Z to stop it, else, the vcd will become a large file and will never end.
- This is a 4 bit up-counter.
- Note that, the counter wraps around. Once it reaches “1111”, it goes back to 0000--> you should be able to see why.
- Convert this to an up-down counter. You will need to introduce a signal called- “up_down”. If up_down=1, count up. If up_down=0, count down. Test it with a testbench