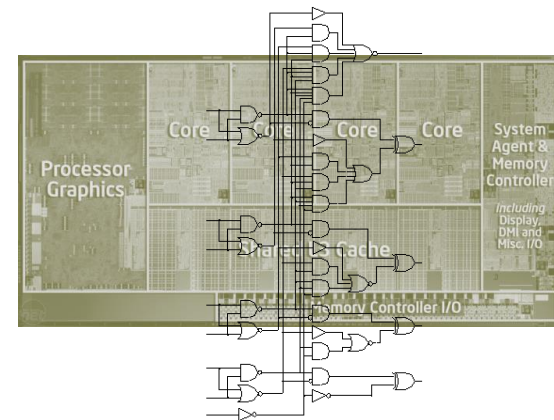




COSE221, COMP211 Logic Design

Lecture 5. Verilog HDL #3

Prof. Taeweon Suh

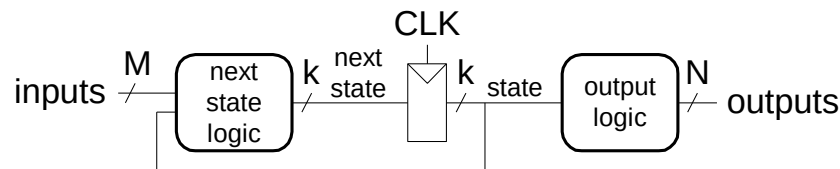
Computer Science & Engineering

Korea University

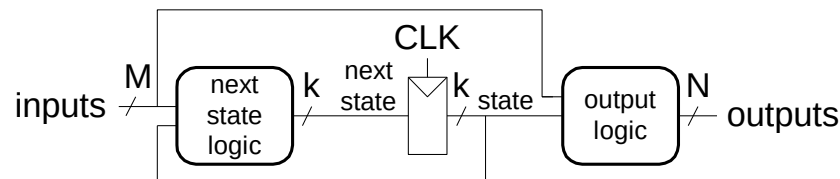
FSM Revisit

- Synchronous sequential circuit can be drawn like below
 - These are called **FSMs**
 - Super-important in digital circuit design
- FSM is composed of
 - **State register**
 - **Combinational logic** that
 - Computes the **next state** based on current state and input
 - Computes the **outputs** based on current state (and input)

Moore FSM



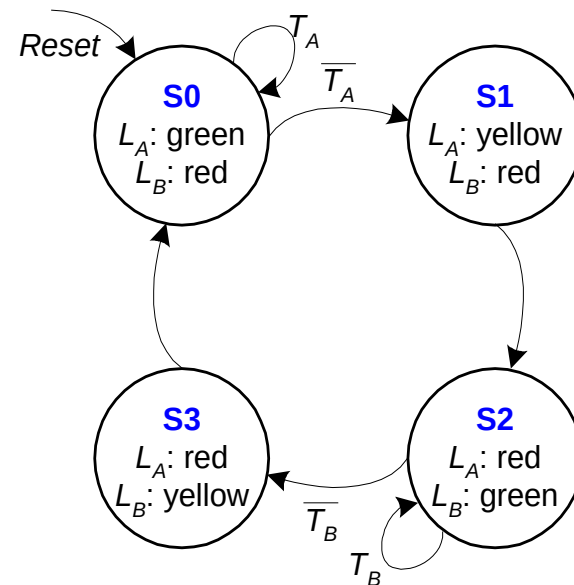
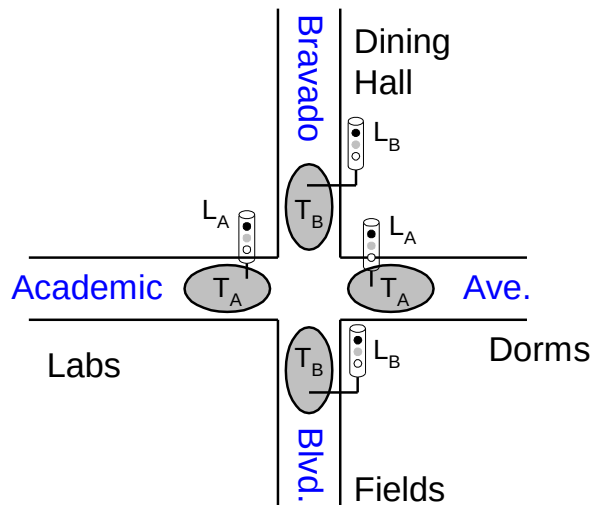
Mealy FSM



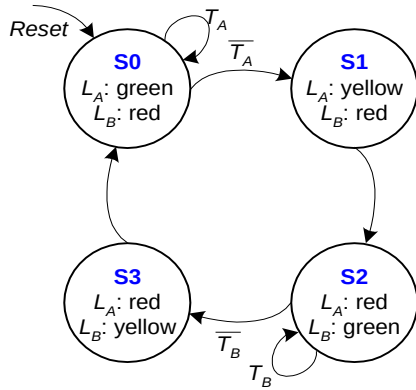
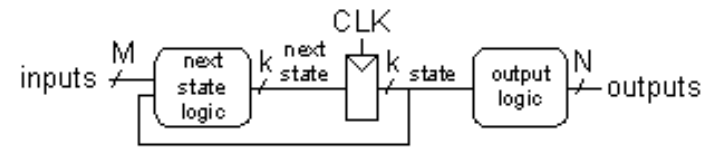
Traffic Light Controller Revisit

- A simplified traffic light controller

- Traffic sensors (sensing human traffic): T_A , T_B
 - Each sensor becomes TRUE if students are present
 - Each sensor becomes FALSE if students are NOT present (i.e., the street is empty)
- Lights: L_A , L_B
 - Each light receives digital inputs specifying whether it should be green, yellow, or red



Moore FSM in Verilog



```
`timescale 1ns/1ps

module moore_traffic_light
    (input      clk,  reset,  TA,  TB,
     output reg [1:0] LA, LB);

    reg [1:0] currstate, nextstate;

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

    parameter green  = 2'b00;
    parameter yellow = 2'b01;
    parameter red    = 2'b10;

    // state register
    always @ (posedge clk, posedge reset)
    begin
        if (reset) currstate <= S0;
        else       currstate <= nextstate;
    end
```

```
// next state logic
always @ (*)
begin
    case (currstate)
        S0: if (~TA) nextstate = S1;
            else      nextstate = S0;
        S1: nextstate = S2;
        S2: if (~TB) nextstate = S3;
            else      nextstate = S2;
        S3:      nextstate = S0;
        default: nextstate = S0;
    endcase
end
```

```
// output logic
always @ (*)
begin
    if (currstate == S0) begin
        LA = green;
        LB = red;
    end
    else if (currstate == S1) begin
        LA = yellow;
        LB = red;
    end
    else if (currstate == S2) begin
        LA = red;
        LB = green;
    end
    else begin
        LA = red;
        LB = yellow;
    end
end
```

Testbench for Traffic Light FSM

```
`timescale 1ns/1ps

module moore_traffic_light_tb();
    reg clk, reset;
    reg TA, TB;
    wire [1:0] LA, LB;

    parameter clk_period = 10;

    moore_traffic_light dut(.clk      (clk),
                           .reset    (reset),
                           .TA       (TA),
                           .TB       (TB),
                           .LA       (LA),
                           .LB       (LB) );

    initial
    begin
        reset = 1;
        #13
        reset = 0;
    end
```

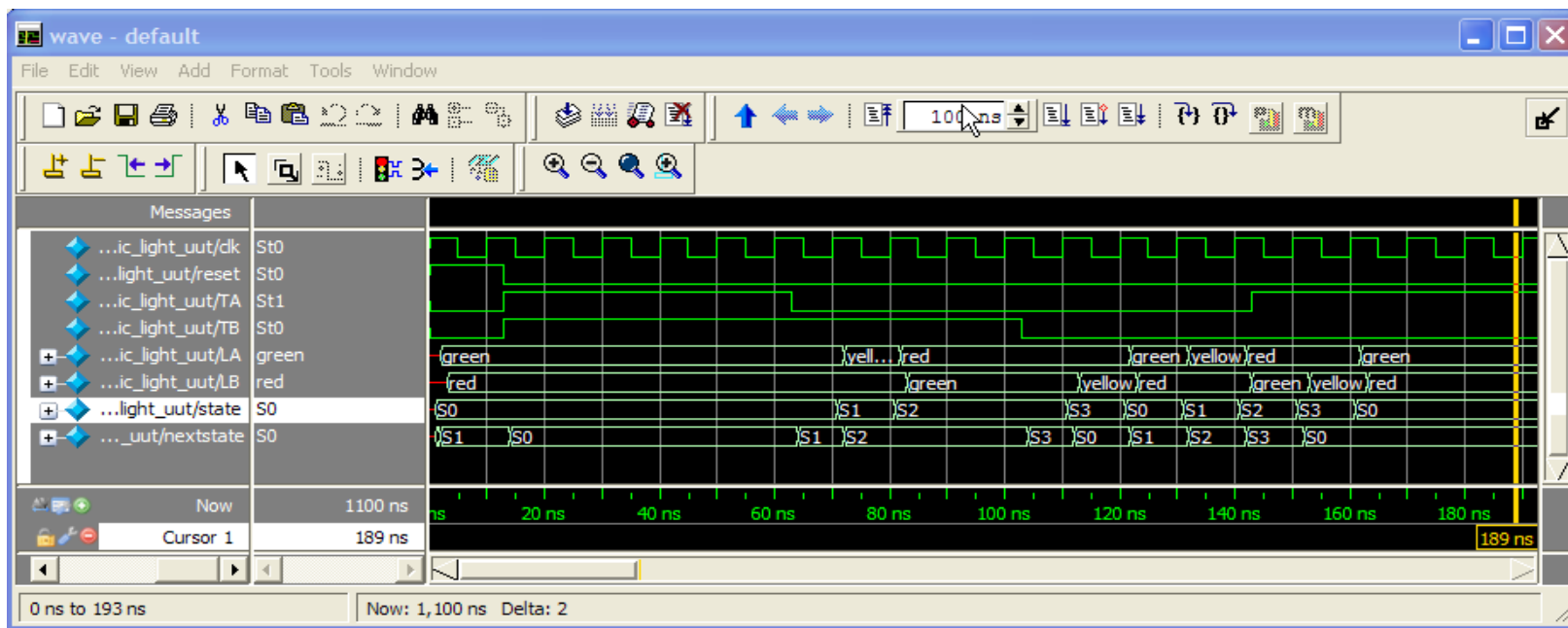
```
always
begin
    clk = 1;
    forever #(clk_period/2) clk = ~clk;
end

initial
begin
    TA = 0; TB = 0; #3
    TA = 0; TB = 0; #(clk_period)
    TA = 1; TB = 1; #(clk_period*5)
    TA = 0; TB = 1; #(clk_period*4)
    TA = 0; TB = 0; #(clk_period*4)
    TA = 1; TB = 0;
end

endmodule
```

Simulation with ModelSim

- Useful tips in using ModelSim
 - To display state information as described in Verilog code
 - Format: radix define name { }
 - Example: radix define mystate {2'b00 "S0", 2'b01 "S1", 2'b10 "S2", 2'b11 "S3", -default binary}
 - radix define mylight {2'b00 "green", 2'b01 "yellow", 2'b10 "red", -default binary}
 - Save the display information for the use in the future
 - File->Save Format, Then click on "OK"
 - By default, it will save the waveform format to "wave.do"

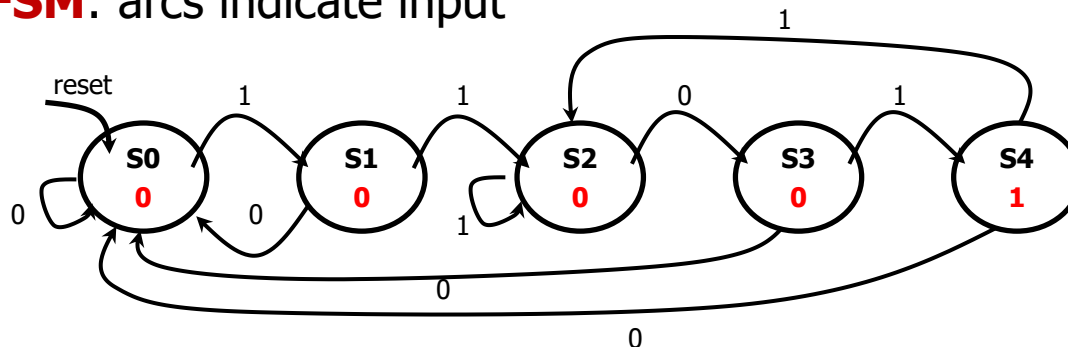


Snail FSM Revisit

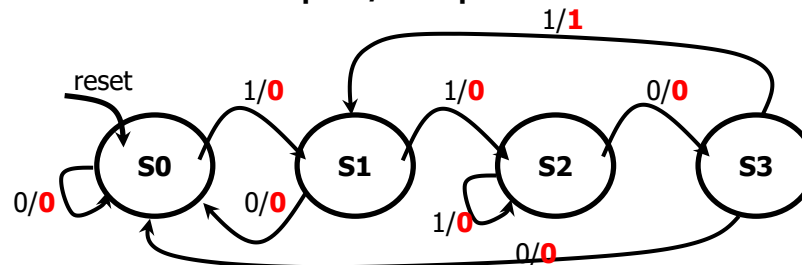


- There is a snail
 - The snail crawls down a paper tape with 1's and 0's on it
 - The snail smiles whenever the last four digits it has crawled over are **1101**

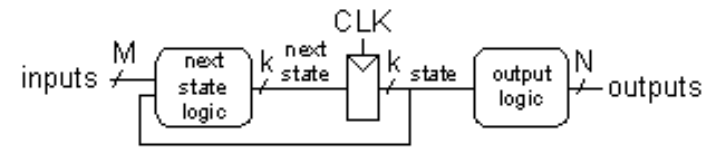
Moore FSM: arcs indicate input



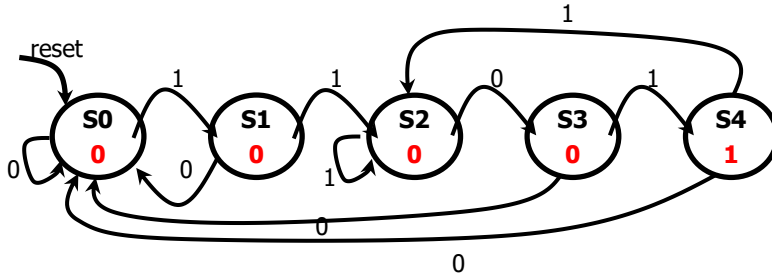
Mealy FSM: arcs indicate input/output



Moore FSM in Verilog



Moore FSM: arcs indicate input



```
`timescale 1ns/1ps

module moore_snail(input  clk, reset, bnum,
                  output reg smile);

    reg [2:0] currstate, nextstate;

    parameter S0 = 3'b000;
    parameter S1 = 3'b001;
    parameter S2 = 3'b010;
    parameter S3 = 3'b011;
    parameter S4 = 3'b100;

    parameter delay = 1;

    // state register
    always @(posedge reset, posedge clk)
    begin
        if (reset) #delay currstate <= S0;
        else      #delay currstate <= nextstate;
    end
```

```
// next state logic
always @(*)
begin
    case (currstate)
        S0: if (bnum) #delay nextstate = S1;
            else      #delay nextstate = S0;

        S1: if (bnum) #delay nextstate = S2;
            else      #delay nextstate = S0;

        S2: if (bnum) #delay nextstate = S2;
            else      #delay nextstate = S3;

        S3: if (bnum) #delay nextstate = S4;
            else      #delay nextstate = S0;

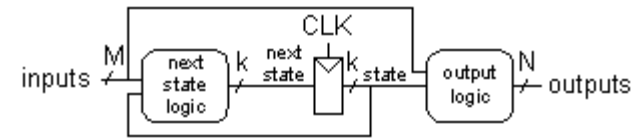
        S4: if (bnum) #delay nextstate = S2;
            else      #delay nextstate = S0;

        default:      #delay nextstate = S0;
    endcase
end

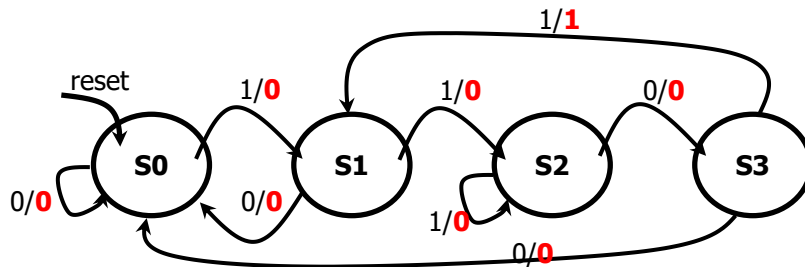
// output logic
always @(*)
begin
    if (currstate == S4) smile = 1'b1;
    else                  smile = 1'b0;
end

endmodule
```

Mealy FSM in Verilog



Mealy FSM: arcs indicate input/output



```
`timescale 1ns/1ps

module mealy_snail(input clk, reset, bnum,
                  output reg smile);

    reg [1:0] currstate, nextstate;

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

    parameter delay = 1;

    // state register
    always @(posedge reset, posedge clk)
    begin
        if (reset) #delay currstate <= S0;
        else      #delay currstate <= nextstate;
    end
```

```
// next state logic
always @(*)
begin
    case (currstate)
        S0: begin
            if (bnum) #delay nextstate = S1;
            else      #delay nextstate = S0;
        end
        S1: begin
            if (bnum) #delay nextstate = S2;
            else      #delay nextstate = S0;
        end
        S2: begin
            if (bnum) #delay nextstate = S2;
            else      #delay nextstate = S3;
        end
        S3: begin
            if (bnum) #delay nextstate = S1;
            else      #delay nextstate = S0;
        end
        default:      #delay nextstate = S0;
    endcase
end

// output logic
always @(*)
begin
    case (currstate)
        S3: begin
            if (bnum) #delay smile = 1'b1;
            else      #delay smile = 1'b0;
        end
        default:      #delay smile = 1'b0;
    endcase
end

endmodule
```

Testbench for Snail FSM

```
`timescale 1ns/1ps

module fsm_snail_tb( );

    reg clk, reset, bnum;
    wire smile_moore;
    wire smile_mealy;

    parameter clk_period = 10;

    moore_snail    moore_snail_uut
        (.clk      (clk),
         .reset     (reset),
         .bnum      (bnum),
         .smile     (smile_moore));

    mealy_snail    mealy_snail_uut
        (.clk      (clk),
         .reset     (reset),
         .bnum      (bnum),
         .smile     (smile_mealy));
```

```
initial
begin
    reset = 1; #13
    reset = 0;

end

always
begin
    clk = 1;
    forever #(clk_period/2) clk = ~clk;
end

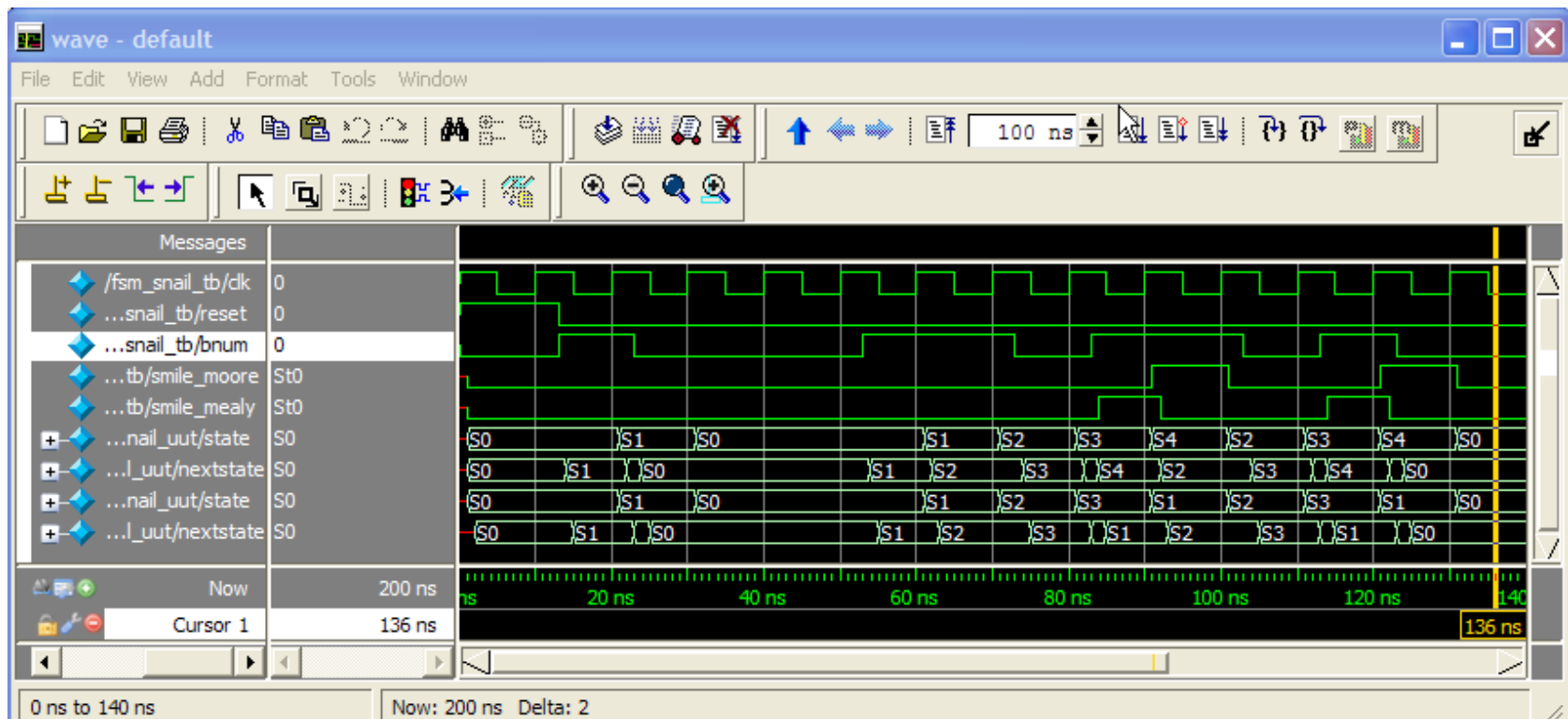
initial
begin
    bnum = 0;    #3;
    bnum = 0;    #clk_period;
    bnum = 1;    #clk_period;
    bnum = 0;    #clk_period;
    bnum = 0;    #clk_period;
    bnum = 0;    #clk_period;
    bnum = 1;    #clk_period;
    bnum = 1;    #clk_period;
    bnum = 0;    #clk_period;
    bnum = 1;    #clk_period; // Smile
    bnum = 1;    #clk_period;
    bnum = 0;    #clk_period;
    bnum = 1;    #clk_period; // Smile
    bnum = 0;

end

endmodule
```

Simulation with ModelSim

- Use radices below for display purpose
 - radix define moore_state {3'b000 "S0" , 3'b001 "S1" , 3'b010 "S2" , 3'b011 "S3" , 3'b100 "S4" , -default binary}
 - radix define mealy_state {2'b00 "S0" , 2'b01 "S1" , 2'b10 "S2" , 2'b11 "S3" , -default binary}



HDL Summary

- HDLs are extremely important languages for modern digital designers
- You are able to design digital systems with HDL much faster than drawing schematics
- Debug cycle is also often much faster because modifications require code changes instead of tedious schematic redrawing
 - However, the debug cycle can be much longer with HDLs if you don't have a good idea of the hardware your code implies

HDL Summary

- The most important thing to remember when writing HDL code is that **you are describing real hardware!** (not writing a software program)
- The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce
 - If you don't know what hardware your code is implying, you mostly likely don't get what you want
- When designing with HDL, sketch a block diagram of your system
 - Identify which portions are combinational logic, sequential logic, FSMs and so on, so forth
 - Write HDL code for each portion and then merge together

Backup Slides

Testbench and TestVector

- Testbench
 - HDL code written to test another HDL module, the **device under test (dut)** (also called the **unit under test (uut)**)
 - Testbench contains statements to apply input to the DUT and ideally to check the correct outputs are produced
- Testvectors
 - Inputs to DUT and desired output patterns from DUT
- Types of testbenches
 - Simple testbench
 - Self-checking testbench
 - **Self-checking testbench with testvectors**

Simple Testbench Revisit

```
module sillyfunction
    (input  a, b, c,
     output y);

    assign y = ~a & ~b & ~c |
               a & ~b & ~c |
               a & ~b &  c;

endmodule
```

testbench

```
`timescale 1ns/1ps

module testbench1();
    reg  a, b, c;
    wire y;

    // instantiate device under test
    sillyfunction dut(.a (a), .b (b),
                     .c (c), .y (y));

    // apply inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
    end

endmodule
```

testvectors

Self-checking Testbench Revisit

```
module testbench2();  
    reg a, b, c;  
    wire y;  
  
    // instantiate device under test  
    sillyfunction dut(.a (a), .b (b),  
                     .c (c), .y (y));
```

```
// apply inputs one at a time  
// checking results  
initial begin  
    a = 0; b = 0; c = 0; #10;  
    if (y !== 1) $display("000 failed.");  
    c = 1; #10;  
    if (y !== 0) $display("001 failed.");  
    b = 1; c = 0; #10;  
    if (y !== 0) $display("010 failed.");  
    c = 1; #10;  
    if (y !== 0) $display("011 failed.");
```

```
    a = 1; b = 0; c = 0; #10;  
    if (y !== 1) $display("100 failed.");  
    c = 1; #10;  
    if (y !== 1) $display("101 failed.");  
    b = 1; c = 0; #10;  
    if (y !== 0) $display("110 failed.");  
    c = 1; #10;  
    if (y !== 0) $display("111 failed.");  
end
```

```
endmodule
```

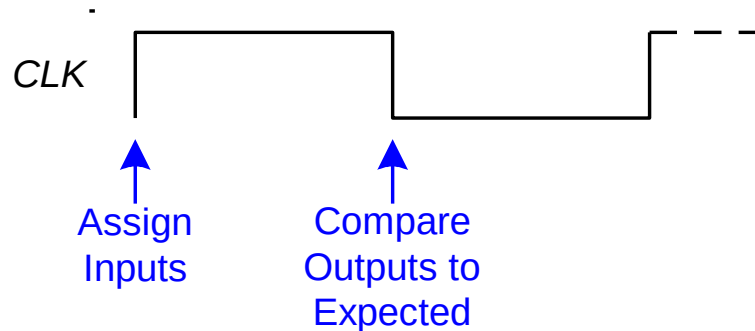
testvectors

Self-Checking Testbench with Testvectors

- Writing code for each testvector is tedious, especially for modules that require a large number of vectors
- A better approach is to place the testvectors in a separate file
- Then, testbench reads the testvector file, applies input to DUT and compares the DUT's outputs with expected outputs
 1. Generate clock for assigning inputs and reading outputs
 2. Read the testvector file into array
 3. Assign inputs and expected outputs to signals
 4. Compare outputs to expected outputs and report errors if there is discrepancy

Testbench with Testvectors

- Testbench clock is used to assign inputs (on the rising edge) and compare outputs with expected outputs (on the falling edge)



- The testbench clock may also be used as the clock source for synchronous sequential circuits

Testvector File Example

example.tv contains vectors of `abc_yexpected`

000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0

```
module sillyfunction(input  a, b, c,  
                      output y);  
  
    assign y =      ~a & ~b & ~c |  
                    a & ~b & ~c |  
                    a & ~b &  c;  
  
endmodule
```

Self-Checking Testbench with Testvectors

1. Generate clock for assigning inputs, reading outputs
2. Read a testvector file into array
3. Assign inputs and expected outputs to signals

```
module testbench3();
    reg          clk, reset;
    reg          a, b, c, yexpected;
    wire         y;
    reg [31:0] vectornum, errors;
    reg [3:0] testvectors[10000-1:0];
    // array of testvectors

    // instantiate device under test
    sillyfunction dut(.a (a), .b (b),
                     .c (c), .y (y));

    // clock
    always
    begin
        clk = 1; #5; clk = 0; #5;
    end
```

```
    initial
    begin
        $readmemb("example.tv", testvectors);
    end

    initial
    begin
        reset = 1;
        #27;
        reset = 0;
    end

    // Note: $readmemh reads testvector files
    // written in hexadecimal
    // apply test vectors on rising edge of clk
    always @(posedge clk)
    begin
        {a, b, c, yexpected} = testvectors[vectornum];
    end
```

Self-Checking Testbench with Testvectors

4. Compare outputs to expected outputs and report errors if there is discrepancy

```
// check results on falling edge of clk
always @(negedge clk)
begin
    if (reset) begin
        vectornum = 0;
        errors = 0;
    end
    else
    begin // skip during reset
        if (y !== yexpected) begin
            $display("Error: inputs = %b", {a, b, c});
            $display("  outputs = %b (%b expected)", y, yexpected);
            errors = errors + 1;
        end

        // increment array index and read next testvector
        vectornum = vectornum + 1;
        if (testvectors[vectornum] === 4'bx) begin
            $display("%d tests completed with %d errors", vectornum, errors);
            $stop;          // stop the simulation here
            //$finish;      // exit the simulation
        end
    end
end
end

endmodule
```

// Note: "===" and "!=" can compare values that are x or z.