Verilog

- Verilog is a **Hardware Description Language (HDL)**. It is used to describe the structure and behavior of the hardware. The final product of a Verilog program is the generated hardware circuit.

- Verilog is **NOT** a sequential programming language.
How to describe functionality?

- Behavioral: what a module does
- Structural: how a module is built
Continuous Assignment Statements (assign)

- Describes only combinational logic

- These statements are re-evaluated anytime any of the inputs on the right hand side changes.

```
assign <out> = <in0> <operator> <in1> ... ;
```
Find the bug in this 2 input AND gate!

```verilog
module assign_example (A, B, Y);
    input A;
    input B;
    output Y
    Y = A & B;
endmodule
```
Fixed version.

```verilog
module assign_example(A,B,Y);
    input A;
    input B;
    output Y;

    assign Y = A & B;
endmodule
```
Some rules (assign)

- Assign statements can only be used to set signals declared as wires.
  - Wire
  - Input
  - Output

- Assign statements cannot be used inside `always` blocks.
  - More on this later.
always Statements

- These statements are executed when a signal in the sensitivity list changes.
  - Otherwise, signals keep their old value.
- Can be used to describe sequential logic.

```verilog
module flipflop (D,CLK,Q);
  input  D;
  input  CLK;
  output reg  Q;
  
  always @(posedge CLK) begin
    Q <= D;
  end
endmodule
```
Find the bug!!

```verilog
module latch_example(D, CLK, Q);
    input D;
    input CLK;
    output reg Q;

    always @(CLK) begin
        if (CLK) begin
            Q <= D;
        end
    end
endmodule
```
Fixed version of latch.

```verilog
module latch_example(D, CLK, Q);
  input D;
  input CLK;
  output reg Q;

  always @(CLK, D) begin
    if (CLK) begin
      Q <= D;
    end
  end
endmodule
```
Some rules (always blocks)

- The sensitivity list for an always block specifies the signals that cause the block to update.
  - * in the sensitivity list updates a block when any signal in the circuit changes.

- Every signal is either a **wire** or a **reg**.

- You can only assign to a reg inside always blocks.

- You can read from a wire or a reg inside always block.
always Statements

- Can also be used to describe combinational logic.

```verilog
module always_ex(A, B, C, Y);
    input A;
    input B;
    input C;
    output reg Y;
    reg intern;

    always @(*) begin
        intern = A & B;
        Y = ~intern | C;
    end
endmodule

module assign_ex(A, B, C, Y);
    input A;
    input B;
    input C;
    output Y;
    wire intern;

    assign intern = A & B;
    assign Y = ~intern | C;
endmodule
```
Wire vs. Reg Recap

● Every signal is either a wire or a reg.
  ○ You can only assign to a wire outside always blocks using `assign` statement
  ○ You can only assign to a reg inside always blocks using `=` or `<=`.
  ○ You can read from wire or reg anywhere.

● Signals declared as inputs or outputs are AUTOMATICALLY considered as wires.

● A reg is not the same as a register, it is a signal which you intend to describe behaviorally.
Inferred Latches

What if you forgot to specify the else branch on an if statement?

```verilog
module inferred_latch(A,Y);
    input [1:0] A;
    output reg    Y;

    always@(*) begin
        if (A == 2'b00) begin
            Y = 1;
        end
        else if (A == 2'b01) begin
            Y = 0;
        end
        else if (A == 2'b10) begin
            Y = 0;
        end
    end
endmodule
```
HDL code

- Synthesizable modules
  - Describes the hardware

- Testbench
  - Intended only for simulation
  - Cannot be synthesized
Testbench

- A Verilog module that is used to test another module, called the unit under test (UUT).
- Contains statements to apply inputs to the UUT.
- The input and desired output patterns are called test vectors.
/*
  Testbench for latch_example.v.
*/

```
timescale 1 ps / 1 ps

module latch_example_test();
  // connections to latch module
  reg D;
  reg CLK;
  wire Q;

  // instance of module to test (Unit Under Test)
  latch_example UUT (  
    .D(D),  
    .CLK(CLK),  
    .Q(Q)  
  );

  // specify inputs to D and CLK below and observe changes in circuit
  always begin
    CLK = 1'b0;
    CLK = 1'b1;
    $timeout(10);
    D = 1'b1;
    $timeout(10);
  end

  initial begin
    D = 1'b0;
    $timeout(10);
    D = 1'b1;
    $timeout(10);
  end
```

```
Debugging using a waveform

```verilog
module latch_example(D, CLK, Q);
  input D;
  input CLK;
  output reg Q;

  always @(CLK) begin
    if (CLK) begin
      Q <= D;
    end
  end
endmodule
```
Latch fixed!

```verilog
module latch_example(D, CLK, Q);
    input D;
    input CLK;
    output reg Q;

    always @(CLK, D) begin
        if (CLK) begin
            Q <= D;
        end
    end
endmodule
```
Debug a circuit using waveform.

1. Download logic_b example for CMS.
2. Work in groups of 1-3 to try to figure out and fix bug.
3. We will discuss results and debug strategy afterwards.
# MSIM>
# MSIM> TEST CASES
# MSIM>
# MSIM> [PASSED]: For A=0, B=0, C=1, SEL=00, the segment logic is correct. OUT = 0
# MSIM> [PASSED]: For A=0, B=0, C=0, SEL=00, the segment logic is correct. OUT = 1
# MSIM> [PASSED]: For A=1, B=1, C=1, SEL=01, the segment logic is correct. OUT = 1
# MSIM> [PASSED]: For A=1, B=0, C=0, SEL=01, the segment logic is correct. OUT = 0
# MSIM> [ININCORRECT]: For A=1, B=1, C=0, SEL=10, the segment logic is incorrect. OUT = 0 expected 1
# MSIM> [ININCORRECT]: For A=1, B=0, C=1, SEL=10, the segment logic is incorrect. OUT = 1 expected 0
# MSIM> [PASSED]: For A=1, B=1, C=0, SEL=11, the segment logic is correct. OUT = 0
# MSIM> [PASSED]: For A=1, B=0, C=1, SEL=11, the segment logic is correct. OUT = 1
Systematic Debugging

1. Based on waveform, only SEL = 10 results in incorrect logic.
2. logic_b.v implements the correct logic for SEL = 10.
3. Check sub-module mux4.
4. Notice that SEL signals to one of the mux2’s are incorrect.
5. SEL[1] -> SEL[0].
/ * 
  * There is a functional problem with the 4 input mux circuit created below. 
  */ 

module logic_b(A,B,C,SEL,OUT); 
// declare inputs and outputs
input A;
input B;
input C;
input [1:0] SEL; //create 1 2-bit input wire (notice placement of brackets before name)
output OUT;

// internal wires
wire mux4_in[3:0]; //create 4 1-bit wires (notice placement of brackets after name)

// logic
assign mux4_in[3] = (A | B) & C; 
assign mux4_in[2] = B; 
assign mux4_in[1] = A & B; 
assign mux4_in[0] = A | B | ~C; 

// setup mux4
mux4 mux4_0 (.IN_3(mux4_in[3]),
              .IN_2(mux4_in[2]),
              .IN_1(mux4_in[1]),
              .IN_0(mux4_in[0]),
              .SEL(SEL),
              .OUT(OUT));
endmodule
module mux4 (IN_3, IN_2, IN_1, IN_0, SEL, OUT);
  // declare inputs and outputs
  input   IN_3;
  input   IN_2;
  input   IN_1;
  input   IN_0;
  input [1:0] SEL;
  output  OUT;

  // outputs of first column of muxes
  wire mux1_out;
  wire mux0_out;

  // first column of mux2 creation
  mux2 mux1 (.IN_1(IN_3),
              .IN_0(IN_2),
              .SEL(SEL[1]),
              .OUT(mux1_out));
  mux2 mux0 (.IN_1(IN_1),
              .IN_0(IN_0),
              .SEL(SEL[0]),
              .OUT(mux0_out));

  // second column mux2 to combine
  mux2 muxf (.IN_1(mux1_out),
             .IN_0(mux0_out),
             .SEL(SEL[1]),
             .OUT(OUT));
endmodule
Quiz

Fix the Verilog implementation of the circuit below.

```verilog
module quiz(A,B,SEL,Y);
    input A;
    input B;
    input SEL;
    output reg Y;

    wire mux_in;

    assign mux_in = (A & ~B) | ~A;

    always @(*) begin
        if (SEL) begin
            Y = B;
        end
        else begin
            Y = mux_in;
        end
    end
endmodule
```
Common Verilog Compiler Messages and Pitfalls

- **Assigning to wire in always block**: “Error (10137): Verilog HDL Procedural Assignment error at universalShift.v(22): object "myErrorWire" on left-hand side of assignment must have a variable data type”

- **Assigning to reg outside always block**: One of several errors: “Error (10170): Verilog HDL syntax error at universalShift.v(17) near text "="; expecting ".", or "(" or “Error (10219): Verilog HDL Continuous Assignment error at universalShift.v(17): object "myErrorReg" on left-hand side of assignment must have a net type”

- **Improper usage of always @(*) vs always @(posedge CLK)**: Not a compiler error or warning, but can result in incorrect behavior if the wrong type of always block is used”
Common Verilog Compiler Messages and Pitfalls

- **Bit truncation warning** “Warning (10230): Verilog HDL assignment warning at universalShift.v(17): truncated value with size 10 to match size of target (8)”
- **Inferred latches** “Warning (10240): Verilog HDL Always Construct warning at universalShift.v(17): inferring latch(es) for variable "accidentalLatch", which holds its previous value in one or more paths through the always construct
- **Trying to assign to a wire in 2 places**: “Error (10028): Can't resolve multiple constant drivers for net "DoubleDrivenWire" at universalShift.v(17)"
Common Verilog Compiler Messages and Pitfalls

- **Driving Input Ports “Error (10231):** Verilog HDL error at universalShift.v(15): value cannot be assigned to input "drivenInput"" followed by “Error (10031): Net "drivenInput" at universalShift.v(15) is already driven by input port "drivenInput", and cannot be driven by another signal”
- **Declaring Inputs as Reg:** “Error (10278): Verilog HDL Port Declaration error at universalShift.v(15): input port "IN" cannot be declared with type """
- **Dangling ports:** Compiler may not throw a warning or error, but occurs when you do not connect a wire to every port on a module.
Common Verilog Compiler Messages and Pitfalls

- Improper Port Declaration (declared in Port list but not listed as input or output): “Error (10161): Verilog HDL error at universalShift.v(5): object "OUT" is not declared” - declared as input or output but not in port list “Error (10206): Verilog HDL Module Declaration error at universalShift.v(14): top module port "OUT" is not found in the port list”