What is Verilog???

Verilog is a programming language that has been developed for describing digital circuits and systems. Verilog combines:

- a design entry language
- a simulation modeling language
- a test language
There are 4 methods of describing designs:

- behavioral
- dataflow
- structural
- any combination of the above

**NOTE:** behavioral and dataflow descriptions are also known as RTL descriptions
For now we are only going to consider the dataflow style. We will use the following decoder circuit as an example:
The basic unit of description in Verilog is the \textit{module}. The module for our circuit is

\begin{verbatim}
module decoder2X4 (A, B, ENB, Y);

    input A, B, ENB; // binary signals
    output [3:0] Y;  // 4-bit bus signal
    wire Abar, Bbar; // internal signals

    // make NAND prop delay = 3

    assign #3 Abar = ~A;
    assign #3 Bbar = ~B;
    assign #3 Y[0] = ~(Abar & Bbar & ENB);
    assign #3 Y[1] = ~(Abar & B & ENB);
    assign #3 Y[2] = ~(A & Bbar & ENB);
    assign #3 Y[3] = ~(A & B & ENB);

endmodule
\end{verbatim}
NOTES:

1. Verilog is case-sensitive

2. Verilog is free format—i.e., statements can be written on one line or on multiple lines

3. **assign** continually attaches a value to a net (wire). The format is

   ```
   assign [delay] LHS-net = RHS-expression
   ```

   Whenever something on the RHS changes, it is evaluated and the LHS net is updated after the delay.

4. **assign** statements may be in any order

5. `//` is used to create a comment line
Verilog supports two types of operators:

1. Arithmetic operators
   - `+` (add)
   - `-` (subtract)
   - `*` (multiply)
   - `/` (divide)
   - `%` (modulus)

2. Bit-wise operators
   - `~` (unary negation)
   - `&` (binary and)
   - `|` (binary or)
   - `^` (binary exclusive-or)
We need a testbench to simulate the design:

```verilog
module DecoderTest;

  reg DA, DB, DENB; // registered signals
  wire [3:0] DY;

// instantiate the device

decoder2X4 D1 (DA, DB, DENB, DY);

initial // generate waveforms
  begin
    DENB = 0;
    DA = 0;
    DB = 0;
    #10 DENB = 1;
    #10 DA = 1;
    #10 DB = 1;
    #10 DA = 0;
    #10 DB = 0;
    #10 $stop;
  end

endmodule
```
Verilog has 4 basic data types:

- 0 (logic 0)
- 1 (logic 1)
- x (don’t care)
- z (high impedance)

The format of an integer is

\[
[size \ in \ bits] \ 'base \ value
\]

some examples are

4’d2 4-bit decimal with value “2”
3’b001 3-bit binary with value “001”
4’b1x01 4-bit binary with value “1x01”

**NOTE:** with Boolean variables, A = 0 and A = ’b0 are equivalent statements
This integer notation provides a convenient method for constructing testbenches.

Suppose you have a digital system with two inputs ($I_0$ and $I_1$) plus an enable input $E$. The output $Y = I_0 \oplus I_1$ if $E = 0$, but $Y = 1$ regardless of the $I_0$, $I_1$ inputs if $E = 1$.

The testbench to verify the enable function is easily written:

```verilog
reg [1:0] I;
reg E;

initial // generate waveforms
begin
  I = 'b01; // check when $E = 0$
  E = 0; // could also use E = 'b0;

  #10 E = 1; // check when $E = 1$
  #10 I = 'bxx;

  #10 $stop;
end
```
We are now going to consider the *structural* or *gate-level* modeling style.

Verilog provides the following primitive logic gates:

**and, nand, or, nor, xor, xnor**

The format for a gate instantiation is

```
gate_type [name] (Out, In1, In2, ...);
```

The syntax for multiple instances of the same gate type is:

```
gate_type
    [name1] (Out, In1, In2, ...),
    [name2] (Out, In1, In2, ...),
    [name3] (Out, In1, In2, ...),
    ... ,
    [nameM] (Out, In1, In2, ...);
```
In the structural modeling style you explicitly state what is connected to every gate input and every gate output. For the decoder circuit previously described, the module syntax would be as follows:

```
module decoder2X4 (A, B, ENB, Y);

    input A, B, ENB;  // binary signals
    output [3:0] Y;   // 4-bit bus signal
    wire Abar, Bbar; // internal signals

    // Define NOT gates
    not #(3,3) U1 (Abar, A), U2 (Bbar, B);

    // Define NAND gates
    nand #(4,3) U3 (Y[0], Abar, Bbar, ENB), U4 (Y[1], Abar, B, ENB), U5 (Y[2], A, Bbar, ENB), U6 (Y[3], A, B, ENB);

endmodule
```
NOTES:

1. Observe that actual inverter gates are described in the gate-level Verilog program. (The decoder schematic shows NAND gates used as inverters.)

2. Each gate is labelled with a reference designation (U1, U2, etc.)

3. #(4,3) means $t_{PLH} = 4$ and $t_{PHL} = 3$
A convenient declaration for FSMs is the **parameter** statement.

Two program structures are used extensively for describing FSMs. The two structures are

- the **if-then-else** structure

- the **case** structure

These notes discuss both of these topics.
A *parameter* is a constant. It can only be assigned a value once in the program.

The syntax is

```plaintext
parameter param1 = value1;
parameter param2 = value2;
```

or, you can do multiple assignments in one line

```plaintext
parameter param1 = value1, param2 = value2, ...
                   paramN = valueN;
```
Once a parameter is declared, the compiler replaces each instance in the program with the corresponding value.

For example,

    parameter word_size=16;

    :

    reg [word_size-1:0] alpha;

The above defines a 16-bit register “alpha”
The syntax of an if statement is:

```plaintext
if (condition_1)
    procedural_statement_1
{else if (condition_2)
    procedural_statement_2}
[ else
    procedural_statement_3 ]
```

If `condition_1` is non-zero, `procedural_statement_1` is executed. If `condition_1` is 0, `X`, or `Z`, the else branch (if it exists) is executed.
For example,

```plaintext
if (Sum < 60)
    begin
        Grade = C;
        Total_C = Total_C + 1;
    end
else if (Sum < 75)
    begin
        Grade = B;
        Total_B = Total_B + 1;
    end
else
    begin
        Grade = A;
        Total_A = Total_A + 1;
    end
```
The **begin-end** statements are only needed if there are more than two procedural statements.

For example,

```plaintext
if (Nickel_In)      // no begin-end needed
    deposit = 5;
else
    deposit = 0;
```

```plaintext
if (Dime_In)        // must have a begin-end
begin
    deposit = 10;
    Total = Total + 10;
end
```
A case statement executes a conditional branch in two or more ways. The syntax is

```
case (case_expr)
case_item_expr: procedural_statement
...
...
endcase
```
For example,

**parameter**
MON = 0, TUE = 1, WED = 2, THU = 3, FRI = 4, SAT = 5, SUN = 6;

**reg** [2:0] Day;

**integer** money;

**case** (Day)
  TUE: money = 2; // branch 1
  MON, WED, THU: money = 0; // branch 2
  FRI: money = 6; // branch 3
  **default:** money = 3; // branch 4
**endcase**
<table>
<thead>
<tr>
<th>present state</th>
<th>next state</th>
<th>output</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

module MooreFSM (r, ClkM, Z, reset);
input r, ClkM, reset;
output Z;
reg Z;
parameter A = 0, B = 1, C = 2, D = 3;
reg [1:0] MooreState;
always @(reset)
  if (!reset)
    MooreState <= A;
always @(posedge ClkM)
  case (MooreState)
    A: if (!r) MooreState <= A; else MooreState <= C;
    B: if (!r) MooreState <= B; else MooreState <= D;
    C: if (!r) MooreState <= B; else MooreState <= D;
    D: if (!r) MooreState <= D; else MooreState <= A;
  endcase
always @(MooreState)
  case (MooreState)
    A: Z = 1;
    B: Z = 0;
    C: Z = 0;
    D: Z = 1;
  endcase
endmodule
module FSMtest; // testbench

    reg R;
    reg RESET;
    reg CLKM;
    wire ZOUT;

    // instantiate model

    MooreFSM gwg (R, CLKM, ZOUT, RESET);

    initial
    begin

        R= 1;       // initial input

        RESET = 1;  // initialize FSM
        #2 RESET= 0;
        #4 RESET = 1;

        #15 R = 0;   // test inputs
        #25 R = 1;
        #35 R = 0;

        #10 $stop;
    end

    initial
    begin

        CLKM = 0;
        #1 forever
            #10 CLKM = ~CLKM;
    end

endmodule