Finite State Machine Simulation

The Mealy and Moore machines studied in class are examples of Finite State Machines (FSMs). In general, a FSM has an input $\boldsymbol{X}$ which consists of a finite sequence of binary symbols. For example, the sequence $\boldsymbol{X} = X_1, X_2, X_3, X_4 = 0010$ consists of four binary symbols. As the name finite state machine implies, the machine has a finite number of states which can be represented by $S_A, S_B, \ldots, S_K$, for a total of n states. Finally, there is an output $\boldsymbol{Z}$ that consists of a finite sequence of binary symbols just as the input sequence $\boldsymbol{X}$ did.

The dynamics of a FSM are captured by how inputs are processed into outputs. Initially the machine starts in state S_0. Given an input X_1, the machine outputs Z_1 and then makes a transition to state S_1. Thus an input sequence $\boldsymbol{X} = X_1, X_2, X_3, X_4$ is transformed into the output sequence $\boldsymbol{Z} = Z_1, Z_2, Z_3, Z_4$ while passing through the state sequence $\boldsymbol{S} = S_0, S_1, S_2, S_3, S_4$. where S_0 corresponds to the FSM's initial state.

It is convenient to represent a FSM by a state table such as the ones shown in Tables 1 and 2 below. Table 1 shows the state table of the Mealy machine studied in class which detected the input sequence 01 while Table 2 shows the state table of the Moore machine studied in class which also detected the sequence 01. In general the state table of Table 1 corresponds to a clocked Mealy sequential networks while the state table in Table 2 corresponds to a clocked Moore sequential networks as discussed in chapter 13, section 4 of the textbook.

Table 1. Mealy FSM State Table.

Present State	Present Output		Next-State	
	X=0	X=1	X=0	X=1
R_A	0	0	R_B	R_A
R_B	0	1	R_B	R_A

Table 2. Moore FSM State Table.

Present State	Present Output	Next-State	
		X=0	*X=1*
S_A	0	S_B	S_A
S_B	0	S_B	S_C
S_C	1	S_B	S_A

In order to simulate a FSM using Verilog, several new concepts and features need to be introduced. First, recall that we used *reg* when we needed to store a single Boolean value. We can also declare a vector or range of n bits by using the notation [n-1:0] in front of the various *reg* variables. For example,

```
reg X;
reg [3:0]S;
```

defines a scalar or 1-bit register named *X* and a 4-bit vector register named *S* that is made up of four 1-bit registers *S*[3], *S*[2], *S*[1] and *S*[0] from the most to least significant bit.

Values can be assigned to *S* using either binary, octal, hexadecimal or decimal valued constants. For example, the Verilog code

```
reg [3:0] A,B,C,D;
initial
begin
A=4'b1011;
B=4'o13;
C=4'hb;
D=4'D11;
end
```

defines four 4-bit vector registers *A*, *B*, *C* and *D* and then assigns them each the same value in binary (b), octal (o), hexadecimal (h) and decimal (D) formats, respectively. Note that the 4 in front of the prime indicates that the least significant four bits of each expression are to be used. Thus the constants 4'o13 and 4'o33 yield the same values as their least significant four bits are the same.

Sometimes it is convenient to work with constants as letters rather than as numbers. This can be accomplished by using the *parameter* feature. Thus the following code makes the same assignments as the code above.

```
reg [3:0] A,B,C,D;
parameter N0=4'b1011,
          N1=4'o13,
          N2=4'hb,
          N3=4'D11;

initial
begin
A=N0;
B=N1;
C=N2;
D=N3;
end
```

Conditional branching statements in Verilog are similar to the corresponding statements in other programming languages. The *if-then* construct is given by:

```
if (condition)
   [begin]
   statement 1;
   [statement 2;]

   [statement n;]
   [end]
```

while the *if-then-else* construct is:

```
if (condition)
   [begin]
   statement 1;
   [statement 2;]

   [statement n;]
   [end]
else
   [begin]
   statement 1;
   [statement 2;]

   [statement m;]
   [end]
```

The brackets around *begin*, *end* and *statement* means that they are optional or only needed if there is more than one *statement*. The *condition* is a Boolean expression which dictates whether the *if* part of the *if-then* or *if-then-else* is executed. Boolean expressions used in Verilog are similar to those used in other programming languages. Most Boolean expressions are evaluated as either equality or relational operators and include:

```
if (A==B)      //   if A equal B then
if (A!=B)      //   if A not equal B then
if (A>B)       //   if A greater than B then
if (A>=B)      //   if A greater than or equal B then
if (A<B)       //   if A less than B then
if (A<=B)      //   if A less than or equal B then
```

Note that the // are used to insert a one line comments into Verilog code. Finally, Boolean expressions can also involve Boolean operations such as

```
if (A&&B)      //   if A and B are both true then
if (A||B)      //   if A or B is true then
if (!A)        //   if not A is true then
```

Event control statements play an important role in behavioral modeling. One which we will be using in the *"@( )"* control statement which is given by:

```
@(event)
   [begin]
   statement 1;
   [statement 2;]

   [statement n;]
   [end]
```

The event may correspond to a change in the level of a signal or an edge transition of the signal. For example,

```
always @(negedge clock)
   [begin]
   statement 1;
   [statement 2;]

   [statement n;]
   [end]
```

means that statements will be executed every time there is a negative edge transition of the signal *clock* while

```
always @(state)
   [begin]
   statement 1;
   [statement 2;]

   [statement n;]
   [end]
```

means that statements will be executed every time there is a level change of the signal *state*. So if *state* changes from a 0 to a 1 or a 1 to a 0, then the statements will be executed.

The last feature we will be using of Verilog is the *case* statement which can be viewed as a generalization of the *if-then-else* statement. The formal syntax of the *case* statement is:

```
reg [1:0]x;

case(x)
2'b00:   statement 0;
2'b01:   statement 1;
2'b10:   statement 2;
2'b11:   statement 3;
default: statement default;
endcase
```

Here *x* is a 2-bit vector register that evaluates to one of the cases. Note that one does not need to define all the possible cases as any case not explicitly listed will evaluate to the default case. Also, if more than 4 cases are needed, then *x* would need to be a longer vector register.

We now consider how the *case* statement can be used to implement FSMs in Verilog. A behavioral module for the Mealy FSM described in Table 1 is shown in Table 3. While only two states are needed, there is no problem if a longer vector register, which in this case could accommodate up to four states, is used. The FSM is initialized to state S_0 with an output of zero. Note that both the next-state changes on negative edge transitions of the clock while the output changes of both negative edge transitions of the clock or level changes of the input. Recall that the output of a Mealy machine are only valid when the clock is high.

Table 3. Verilog Module Implementing a Mealy FSM.

```
module state(x,ck,state,z);    //  Mealy FSM
  input x,ck;
  output z;
  output [1:0]state;
  reg    z;
  reg    [1:0]state;

  parameter S0=2'b00,
            S1=2'b01;

  initial begin state=S0; z=0; end

  always @(negedge ck)         //  Next-State
  case(state)

  S0:  if      (x==0) state=S1;
       else if (x==1) state=S0;

  S1:  if      (x==0) state=S1;
       else if (x==1) state=S0;
  endcase

  always@(state or x)             //  Output
  case(state)

  S0:  if      (x==0) z=0;
       else if (x==1) z=0;

  S1:  if      (x==0) z=0;
       else if (x==1) z=1;
  endcase
endmodule
```

While it may seem redundant to include a second conditional to check the condition (*x*==1) after checking the condition (*x*==0), it is critical for the proper simulation of the FSM. This is because Verilog also allows the input signal *x* to assume the level x (the value we see as the output of a JK flip-flop when a simulation starts) and we do not want a level of x to change the state or output of the FSM. Note that the next-state transitions occurs on the negative edge of the clock while the output transitions occur on any change in either the state or input.

Table 4 shows module main that will apply a stimulus pattern to the Mealy FSM described by the Verilog code in Table 3. With the exception of declaring the 2-bit vector register *state* as a *wire*, the Verilog code is quite similar to that used to simulate this example in class. Figure 1 shows the timing diagram that results from running the modules main and state.

Table 4. Verilog Module to Simulate a Mealy FSM.

```
module main;
  reg  x,ck;
  wire z;
  wire [1:0]state;

  state FSM(x,ck,state,z);

  initial ck=0;

  always #10 ck=~ck;

  initial
  begin
    $monitor($time,,"clock=%b x=%b state=%b | z=%b",
             ck,x,state,z);
    #00 x=1;
    #25 x=0;
    #20 x=0;
    #20 x=1;
    #20 x=0;
    #20 x=1;
    #20 x=1;
    #20 x=0;
    #20 x=0;
    #20 x=1;
    #20 $finish;
  end
endmodule
```

The Verilog code for a behavioral module of the Moore FSM described in Table 2 is shown in Table 5. Note that the second case statement used to compute the output is not a function of the input. Since the name and parameters of the modules used to implement the Mealy and Moore FSM is the same, the module main can be used to simulate both of

them. Figure 2 shows the timing diagram that results from running the modules main and state for the Moore FSM. Finally, we note that the outputs for the Mealy and Moore FSMs both detect the output 01, just as the Mealy and Moore JK flip-flop sequential networks did in class and produce identical results when taking into account the initialization of the JK flip-flop circuits.

Table 5. Verilog Module Implementing a Moore Type FSM.

```
module state(x,ck,state,z);    //  Moore FSM
  input x,ck;
  output z;
  output [1:0]state;
  reg    z;
  reg    [1:0]state;

  parameter S0=2'b00,
            S1=2'b01,
            S2=2'b10;

  initial begin state=S0; z=0; end

  always@(negedge ck)          // Next-State
  case(state)

  S0:  if      (x==0) state=S1;
       else if (x==1) state=S0;

  S1:  if      (x==0) state=S1;
       else if (x==1) state=S2;

  S2:  if      (x==0) state=S1;
       else if (x==1) state=S0;

  endcase

  always@(state)                  // Output
  case(state)

  S0:  z=0;
  S1:  z=0;
  S2:  z=1;

  endcase

endmodule
```

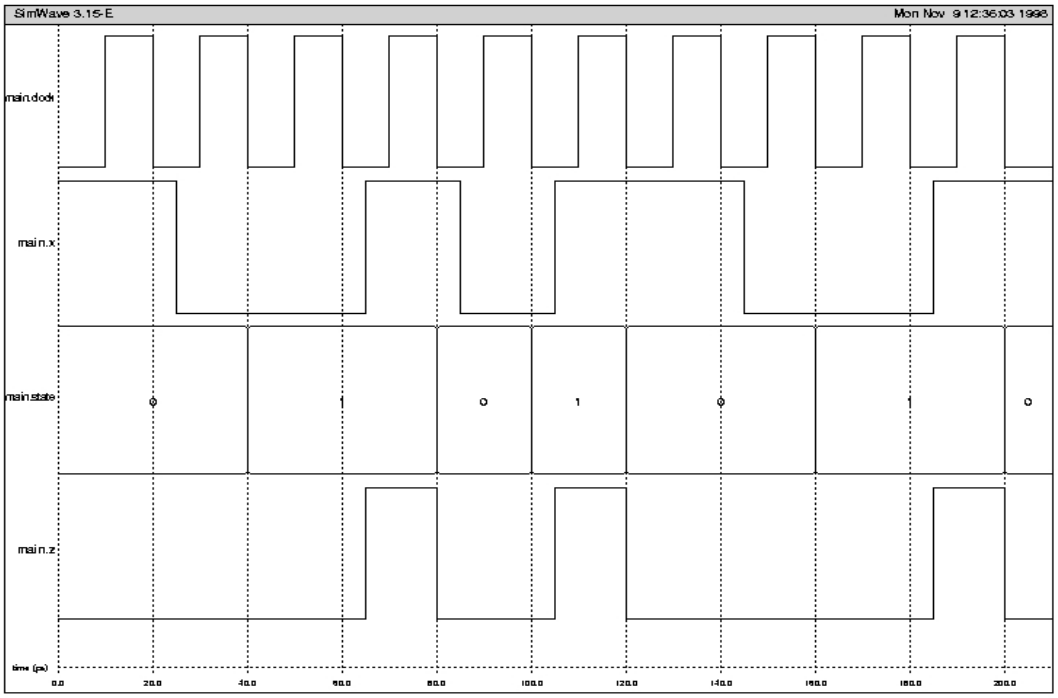

Figure 1. Timing Diagram for the Mealy FSM Specified in Table 1.

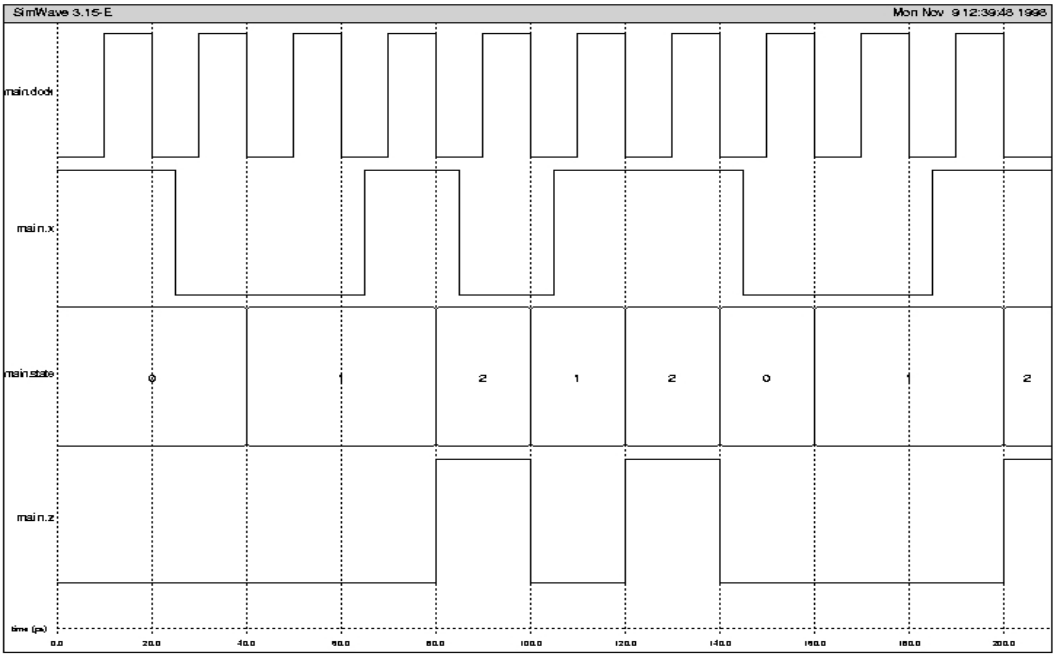

Figure 2. Timing Diagram for the Moore FSM Specified in Table 2.