ECE 274 Digital Logic – Fall 2008

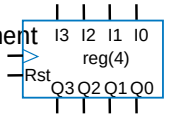
Sequential Logic Design using Verilog

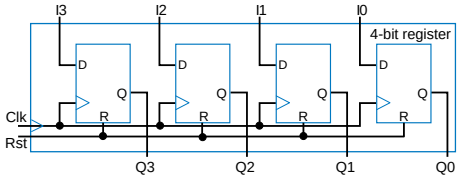
Verilog for Digital Design Ch. 3

THE UNIVERSITY OF ARIZONA. TUCSON ARIZONA

Register Behavior

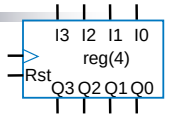
- Sequential circuits have storage
- Register: most common storage component
  - N-bit register stores N bits
  - Structure may consist of connected flip-flops





Register Behavior
Vectors

- Typically just describe register behaviorally
  - Declare output Q as reg variable to achieve storage
- Uses vector types
  - Collection of bits
    - More convenient than declaring separate bits like I3, I2, I1, I0
  - Vector's bits are numbered
    - Options: [0:3], [1:4], etc.
    - [3:0]
      - Most-significant bit is on left
  - Assign with binary constant (more on next slide)



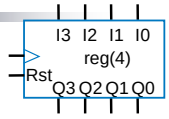

```
module Reg4(I3,I2,I1,I0,Q3,...);
    input I3, I2, I1, I0;
    I3 | I2 | I1 | I0

module Reg4(I, Q, ...);
    input [3:0] I;
I:
  I[3] I[2] I[1] I[0]
```

```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

  input [3:0] I;
  output [3:0] Q;
  reg [3:0] Q;
  input Clk, Rst;

  always @(posedge Clk) begin
     if (Rst == 1 )
       Q <= 4'b0000;
     else
        Q <= I;
  end
endmodule
```

vldd_ch3_Reg4.v

Register Behavior
Constants

- Binary constant
  - 4'b0000
    - 4: size, in number of bits
    - 'b: binary base
    - 0000: binary value
- Other constant bases possible
  - d: decimal base, o: octal base, h: hexadecimal base
  - 12'hFA2
    - 'h: hexadecimal base
    - 12: 3 hex digits require 12 bits
    - FA2: hex value
  - Size is always in bits, and optional
    - *'hFA2* is OK
  - For decimal constant, size and 'd optional
    - 8'd255 or just 255
    - In previous uses like "A <= 1;" 1 and 0 are actually decimal numbers. 'b1 and 'b0 would explicitly represent bits
- Underscores may be inserted into value for readability
  - 12'b1111_1010_0010
  - 8_000_000

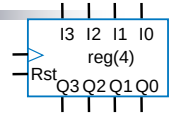



```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

   input [3:0] I;
   output [3:0] Q;
   reg [3:0] Q;
   input Clk, Rst;

   always @(posedge Clk) begin
      if (Rst == 1 )
         Q <= 4'b0000;
      else
         Q <= I;
   end
endmodule
```

vldd_ch3_Reg4.v

Register Behavior

- Procedure's event control involves Clk input
  - *Not* the I input. Thus, synchronous
  - "posedge Clk"
    - Event is not just any change on Clk, but specifically change from 0 to 1 (positive edge)
    - negedge also possible
- Process has synchronous reset
  - Resets output Q only on rising edge of Clk
- Process writes output Q
  - Q declared as reg variable, thus *stores* value too




```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

   input [3:0] I;
   output [3:0] Q;
   reg [3:0] Q;
   input Clk, Rst;

   always @(posedge Clk) begin
      if (Rst == 1 )
         Q <= 4'b0000;
      else
         Q <= I;
   end
endmodule
```

vldd_ch3_Reg4.v

Register Behavior
Testbench

- reg/wire declarations and module instantiation similar to previous testbenches
- Module uses two procedures
  - One generates 20 ns clock
    - 0 for 10 ns, 1 for 10 ns
    - Note: always procedure repeats
  - Other provides values for inputs Rst and I (i.e., vectors)
    - initial procedure executes just once, does not repeat
    - (more on next slide)

```
`timescale 1 ns/1 ns

module Testbench();

   reg [3:0] I_s;
   reg Clk_s, Rst_s;
   wire [3:0] Q_s;

   Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

   // Clock Procedure
   always begin
      Clk_s <= 0;
      #10;
      Clk_s <= 1;
      #10;
   end   // Note: Procedure repeats

   // Vector Procedure
   initial begin
      Rst_s <= 1;
      I_s <= 4'b0000;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b0000;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b1010;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b1111;
   end
endmodule
```

vldd_ch3_Reg4TB.v

Register Behavior
Testbench

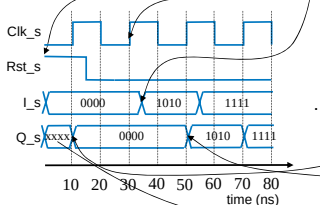
- Variables/nets can be shared between procedures
  - Only one procedure should write to variable
    - Variable can be read by many procedures
    - Clock procedure writes to Clk_s
    - Vector procedures reads Clk_s
- Event control "@(posedge Clk_s)"
  - May be prepended to statement to synchronize execution with event occurrence
    - Statement may be just ";" as in example
    - In previous examples, the "statement" was a sequential block (begin-end)
  - Test vectors thus don't include the clock's period hard coded
- Care taken to change input values away from clock edges

```
`timescale 1 ns/1 ns

module Testbench();

   reg [3:0] I_s;
   reg Clk_s, Rst_s;
   wire [3:0] Q_s;

   Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

   // Clock Procedure
   always begin
      Clk_s <= 0;
      #10;
      Clk_s <= 1;
      #10;
   end   // Note: Procedure repeats

   // Vector Procedure
   initial begin
      Rst_s <= 1;
      I_s <= 4'b0000;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b0000;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b1010;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      I_s <= 4'b1111;
   end
endmodule
```

vldd_ch3_Reg4TB.v

Register Behavior
Testbench

- Simulation results
  - Note that Q_s updated only on rising clock edges
  - Note Q_s thus unknown until first clock edge
    - Q_s is reset to "0000" on first clock edge




```
...
// Vector Procedure
initial begin
   Rst_s <= 1;
   I_s <= 4'b0000;
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   I_s <= 4'b0000;
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   I_s <= 4'b1010;
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   I_s <= 4'b1111;
end
```

vldd_ch3_Reg4TB.v

```
...
always @(posedge Clk) begin
   if (Rst == 1 )
      Q <= 4'b0000;
   else
      Q <= I;
end
...
```

Remember that Q_s is connected to Q, and I_s to I, in the testbench

vldd_ch3_Reg4.v

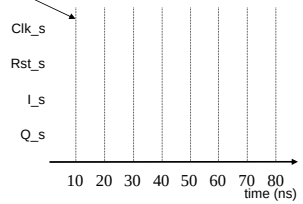
Initial value of a bit is the unknown value x

Common Pitfalls

- Using "always" instead of "initial" procedure
  - Causes repeated procedure execution
- Not including any delay control or event control in an always procedure
  - May cause infinite loop in the simulator
    - Simulator executes those statements over and over, never executing statements of another procedure
    - Simulation time can never advance
  - Symptom – Simulator appears to just hang, generating no waveforms

```
// Vector Procedure
always begin
   Rst_s <= 1;
   I_s <= 4'b0000;
   @(posedge Clk_s);
   ...
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   I_s <= 4'b1111;
end
```

```
// Vector Procedure
always begin
   Rst_s <= 1;
   I_s <= 4'b0000;
end
```



Common Pitfalls

- Not initializing all module inputs
  - May cause undefined outputs
  - Or simulator may initialize to default value. Switching simulators may cause design to fail.
  - *Tip:* Immediately initialize all module inputs when first writing procedure

```
// Vector Procedure
always begin
   Rst_s <= 1;
   I_s <= 4'b0000;
   @(posedge Clk_s);
   ...
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   I_s <= 4'b1111;
end
```

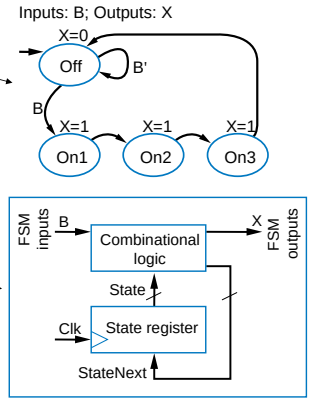

Common Pitfalls

- Forgetting to explicitly declare as a wire an identifier used in a port connection
  - e.g., Q_s
  - Verilog implicitly declares identifier as a net of the default net type, typically a *one-bit* wire
    - Intended as shortcut to save typing for large circuits
    - May not give warning message during compilation
    - Works fine if a one-bit wire was desired
    - But may be mismatch – in this example, the wire should have been four bits, not one bit
    - Unexpected simulation results
  - Always explicitly declare wires
    - Best to avoid use of Verilog's implicit declaration shortcut

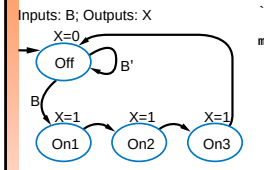
```
`timescale 1 ns/1 ns

module Testbench();

   reg [3:0] I_s;
   reg Clk_s, Rst_s;
   wire [3:0] Q_s;

   Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

...
```

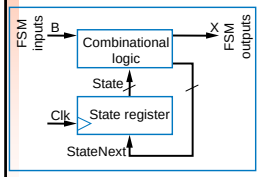

Finite-State Machines (FSMs)—Sequential Behavior

- Finite-state machine (FSM) is a common model of sequential behavior
  - Example: If B=1, hold X=1 for 3 clock cycles
    - Note: Transitions implicitly ANDed with rising clock edge
  - Implementation model has two parts:
    - State register
    - Combinational logic
  - HDL model will reflect those two parts



Finite-State Machines (FSMs)—Sequential Behavior
Modules with Multiple Procedures and Shared Variables



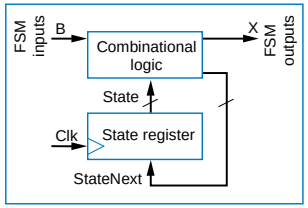



```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On1;
         end
         ...
         S_On1: begin
            X <= 1;
            StateNext <= S_On2;
         end
         S_On2: begin
            X <= 1;
            StateNext <= S_On3;
         end
         S_On3: begin
            X <= 1;
            StateNext <= S_Off;
         end
      endcase
   end

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
endmodule
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior

- Modules has two procedures
  - One procedure for combinational logic
  - One procedure for state register
  - But it's still a behavioral description




```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior
Parameters

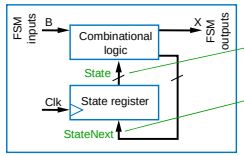
- parameter declaration
  - Not a variable or net, but rather a *constant*
  - A constant is a value that must be initialized, and that cannot be changed within the module's definition
  - Four parameters defined
    - *S_Off, S_On1, S_On2, S_On3*
    - Correspond to FSM's states
      - Should be initialized to unique values

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior

- Module declares two reg variables
  - *State, StateNext*
  - Each is 2-bit vector (need two bits to represent four unique state values 0 to 3)
  - Variables are shared between CombLogic and StateReg procedures
- CombLogic procedure
  - Event control sensitive to *State* and input *B*
  - Will output *StateNext* and *X*
- StateReg procedure
  - Sensitive to *Clk* input
  - Will output *State*, which it stores




```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior
Procedures with Case Statements

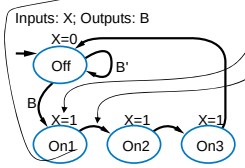
- Procedure may use case statement
  - Preferred over if-else-if when just one expression determines which statement to execute
  - case (expression*)*
    - Execute statement whose case item expression value matches case expression
      - *case item expression : statement*
      - *statement* is commonly a begin-end block, as in example
  - First case item expression that matches executes; remaining case items ignored
  - If no item matches, nothing executes
  - Last item may be "*default : statement*"
    - Statement executes if none of the previous items matched

```
// CombLogic
always @(State, B) begin
  case (State)
     S_Off: begin
        X <= 0;
        if (B == 0)
           StateNext <= S_Off;
        else
           StateNext <= S_On1;
     end
     S_On1: begin
        X <= 1;
        StateNext <= S_On2;
     end
     S_On2: begin
        X <= 1;
        StateNext <= S_On3;
     end
     S_On3: begin
        X <= 1;
        StateNext <= S_Off;
     end
  endcase
end
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior
Procedures with Case Statements

- FSM's CombLogic procedure
  - Case statement describes states
  - case (State)
    - Executes corresponding statement (often a begin-end block) based on State's current value
  - A state's statements consist of
    - Actions of the state
    - Setting of next state (transitions)
- Ex: State is S_On1
  - Executes statements for state On1, jumps to endcase



Suppose State is S_On1

```
reg [1:0] State, StateNext;

// CombLogic
always @(State, B) begin
  case (State)
     S_Off: begin
        X <= 0;
        if (B == 0)
           StateNext <= S_Off;
        else
           StateNext <= S_On1;
     end
     S_On1: begin
        X <= 1;
        StateNext <= S_On2;
     end
     S_On2: begin
        X <= 1;
        StateNext <= S_On3;
     end
     S_On3: begin
        X <= 1;
        StateNext <= S_Off;
     end
  endcase
end
```

vldd_ch3_LaserTimerBeh.v

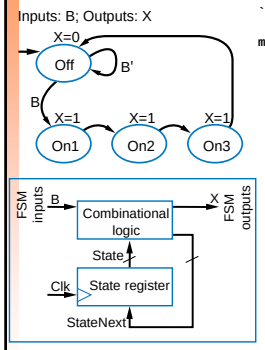
Finite-State Machines (FSMs)—Sequential Behavior

- FSM StateReg Procedure
  - Similar to 4-bit register
    - Register for State is 2-bit vector reg variable
  - Procedure has synchronous reset
    - Resets State to FSM's initial state, S_Off

```
...
   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

...

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
...
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior
Modules with Multiple Procedures and Shared Variables



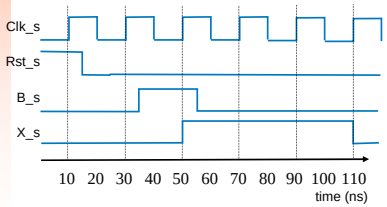

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On1;
         end
         ...
```

```
         ...
         S_On1: begin
            X <= 1;
            StateNext <= S_On2;
         end
         S_On2: begin
            X <= 1;
            StateNext <= S_On3;
         end
         S_On3: begin
            X <= 1;
            StateNext <= S_Off;
         end
      endcase
   end

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
endmodule
```

vldd_ch3_LaserTimerBeh.v

Finite-State Machines (FSMs)—Sequential Behavior
Self-Checking Testbenches

- FSM testbench
  - First part of file (variable/net declarations, module instantiations) similar to before
  - Vector Procedure
    - Resets FSM
    - Sets FSM's input values ("test vectors")
    - Waits for specific clock cycles
  - We observe the resulting waveforms to determine if FSM behaves correctly

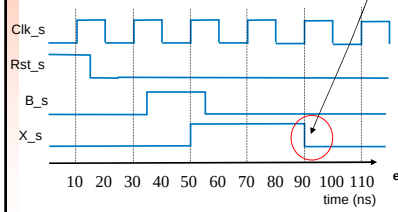



```
...
// Clock Procedure
always begin
   Clk_s <= 0;
   #10;
   Clk_s <= 1;
   #10;
end   // Note: Procedure repeats

// Vector Procedure
initial begin
   Rst_s <= 1;
   B_s <= 0;
   @(posedge Clk_s);
   #5 Rst_s <= 0;
   @(posedge Clk_s);
   #5 B_s <= 1;
   @(posedge Clk_s);
   #5 B_s <= 0;
   @(posedge Clk_s);
   @(posedge Clk_s);
   @(posedge Clk_s);
end
endmodule
```

vldd_ch3_LaserTimerTB.v

Finite-State Machines (FSMs)—Sequential Behavior
Self-Checking Testbenches

- Reading waveforms is error-prone
- Create *self-checking testbench*
  - Use *if* statements to check for expected values
    - If a check fails, print error message
    - Ex: if X_s fell to 0 one cycle too early, simulation might output:
      - *95: Third X=1 failed*




```
// Vector Procedure
initial begin
   Rst_s <= 1;
   B_s <= 0;
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Reset failed", $time);
   Rst_s <= 0;
   @(posedge Clk_s);
   #5 B_s <= 1;
   @(posedge Clk_s);
   #5 B_s <= 0;
   if (X_s != 1)
      $display("%t: First X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 1)
      $display("%t: Second X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 1)
      $display("%t: Third X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 0)
      $display("%t: Final X=0 failed", $time);
end
```

vldd_ch3_LaserTimerTBDisplay.v

Finite-State Machines (FSMs)—Sequential Behavior
$display System Procedure

- $display – built-in Verilog system procedure for printing information to display during simulation
  - A system procedure interacts with the simulator and/or host computer system
    - To write to a display, read a file, get the current simulation time, etc.
    - Starts with $ to distinguish from regular procedures
- String argument is printed literally...
  - $display("Hello") will print "Hello"
  - Automatically adds newline character
- ...except when special sequences appear
  - *%t*: Display a time expression
  - Time expression must be next argument
    - $time – Built-in system procedure that returns the current simulation time
      - *95: Third X=1 failed*

```
// Vector Procedure
initial begin
   Rst_s <= 1;
   B_s <= 0;
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Reset failed", $time);
   Rst_s <= 0;
   @(posedge Clk_s);
   #5 B_s <= 1;
   @(posedge Clk_s);
   #5 B_s <= 0;
   if (X_s != 1)
      $display("%t: First X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 1)
      $display("%t: Second X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 1)
      $display("%t: Third X=1 failed", $time);
@(posedge Clk_s);
#5 if (X_s != 0)
      $display("%t: Final X=0 failed", $time);
end
```

vldd_ch3_LaserTimerTBDisplay.v

Common Pitfall: Not Assigning Every Output in Every State

- FSM outputs should be combinational function of current state (for Moore FSM)
- Not assigning output in given state means previous value is remembered
  - Output has memory
  - Behavior is not an FSM
- Solution 1
  - Be sure to assign every output in every state
- Solution 2
  - Assign default values before case statement
  - Later assignment in state overwrites default

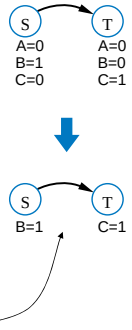
Could delete this without changing behavior (but probably clearer to keep it)

```
// CombLogic
always @(State, B) begin
   X <= 0;
   case (State)
      S_Off: begin
         X <= 0;
         if (B == 0)
            StateNext <= S_Off;
         else
            StateNext <= S_On1;
      end
      S_On1: begin
         X <= 1;
         StateNext <= S_On2;
      end
      S_On2: begin
         X <= 1;
         StateNext <= S_On3;
      end
      S_On3: begin
         X <= 1;
         StateNext <= S_Off;
      end
   endcase
end
```

Common Pitfall: Not Assigning Every Output in Every State

- Solution 2
  - Assign default values before case statement
  - Later assignment in state overwrites default
  - Helps clarify which actions are important in which state
  - Corresponds directly to the common simplifying FSM diagram notation of implicitly setting unassigned outputs to 0

```
case State
   S: begin
      A <= 0;
      B <= 1;
      C <= 0;
   end
   T: begin
      A <= 0;
      B <= 0;
      C <= 1;
   end
endcase

A <= 0;
B <= 0;
C <= 0;
case State
   S: begin
      B <= 1;
   end
   T: begin
      C <= 1;
   end
endcase
```



The Simulation Cycle

- Instructive to consider how an HDL simulator works
  - HDL simulation is complex; we'll introduce simplified form
- Consider example SimEx1
  - Three reg variables – *Q*, *Clk*, *S*
  - Three procedures – P1, P2, P3
- Simulator's job: Determine values for nets and variables over time
  - Repeatedly *executes* and *suspends* procedures
    - Note: Actually considers more objects, known collectively as *processes*, but we'll keep matters simple here to get just the basic idea of simulation
  - Maintains a simulation time *Time*

```
`timescale 1 ns/1 ns

module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
      #10;
   end

   // P2
   always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
      @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Start of simulation
  - Simulation time *Time* is 0
  - Bit variables/nets initialized to the unknown value *x*
  - Execute each procedure
    - In any order, until stops at a delay or event control

We'll use arrow to show where a procedure stops

Procedures

P1	Clk <= 0, then stop. Activate when Time is 0+10=10 ns.
P2	No actions, then stop. Activate when S changes.
P3	No actions, then stop. Activate when Clk changes to 1

Time (ns): Start | 0

Variables		
Q	x	x
Clk	x	*0*
S	x	x

```
`timescale 1 ns/1 ns

module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
      #10;
   end

   // P2
   always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
      @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Simulation cycle
  - Set time to next time at which a procedure activates (note: could be same as current time)
    - In this case, time = 10 ns (P1 activates)
  - Execute active procedures (in any order) until stops

Procedures

P1	Activate when Time is 10 ns. Clk <= 1, stop, activate when Time=10+10=20 ns.
P2	Activate when S changes.
P3	Activate when Clk changes to 1.

Time (ns): Start | 0 | 10

Variables			
Q	x	x	x
Clk	x	0	*1*
S	x	x	x

```
`timescale 1 ns/1 ns

module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
      #10;
   end

   // P2
   always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
      @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Simulation cycle
  - Set time to next time at which a procedure activates
    - Still 10 ns; Clk just changed to 1 (P3 activates)
  - Execute active procedures (in any order) until stops

Procedures

- P1 — Activate when Time is 20 ns.
- P2 — Activate when S changes.
- P3 — Activate when Clk changes to 1
  S <= 1, stop, activate when Clk changes to 1 again

Time (ns):	Start	0	10	10
Q	x	x	x	x
Clk	x	0	1	1
S	x	x	x	*1*

Variables

```
`timescale 1 ns/1 ns
module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
   => #10;
   end

   // P2
=> always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
   => @ (posedge Clk);
      S <= 1;
   => @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Simulation cycle
  - Set time to next time at which a procedure activates
    - *Still* 10 ns; S just changed (P2 activates)
  - Execute active procedures until stops

Procedures

- P1 — Activate when Time is 20 ns.
- P2 — Activate when S changes.
  Q <= 0 (~S), stop, activate when S changes.
- P3 — Activate when change on Clk to 1.

Time (ns):	Start	0	10	10	10
Q	x	x	x	x	*0*
Clk	x	0	1	1	1
S	x	x	x	1	1

Variables

```
`timescale 1 ns/1 ns
module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
   => #10;
   end

   // P2
=> always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
   => @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Simulation cycle
  - Set time to next time at which a procedure activates
    - In this case, set Time = 20 ns (P1 activates)
  - Execute active procedures until stops

Procedures

- P1 — Activate when Time is 20 ns.
  Clk <= 0, stop, activate when T=20+10=30ns.
- P2 — Activate when S changes.
- P3 — Activate when change on Clk to 1.

Time (ns):	Init	0	10	10	10	20
Q	x	x	x	x	0	0
Clk	x	0	1	1	1	*0*
S	x	x	x	1	1	1

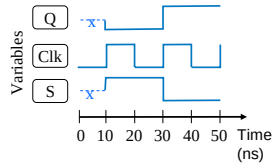
Variables

```
`timescale 1 ns/1 ns
module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
   => #10;
      Clk <= 1;
   => #10;
   end

   // P2
=> always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
   => @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

The Simulation Cycle

- Simulation ends when user-specified time is reached
- Variable/net values translate to waveforms



Variables

Time (ns):	Init	0	10	10	10	20	30	30	30	40	50
Q	x	x	x	x	*0*	0	0	0	*1*	1	1
Clk	x	*0*	*1*	1	1	*0*	*1*	1	1	*0*	*1*
S	x	x	x	*1*	1	1	1	*0*	0	0	0

Variables

```
`timescale 1 ns/1 ns
module SimEx1(Q);

   output reg Q;
   reg Clk, S;

   // P1
   always begin
      Clk <= 0;
      #10;
      Clk <= 1;
      #10;
   end

   // P2
   always @(S) begin
      Q <= ~S;
   end

   // P3
   initial begin
      @ (posedge Clk);
      S <= 1;
      @ (posedge Clk);
      S <= 0;
   end

endmodule
```

vldd_ch3_SimEx1.v

Variable Updates

- Assignment using "<=" ("non blocking assignment") doesn't change variable's value immediately
  - Instead, *schedules* a change of value by placing an *event* on an event queue
  - Scheduled changes occur at end of simulation cycle
- Important implications
  - Procedure execution order in a simulation cycle doesn't matter
    - Assume procedures 1 and 2 are both active
      - Proc1 schedules B to be 1, *but does not change the present value of B.* B is still 0.
      - Proc2 schedules A to be 0 (the present value of B).
      - At end of simulation cycle, B is updated to 1 and A to 0
  - Order of assignments to different variables in a procedure doesn't matter
    - Assume C was 0. Scheduled values will be C=1 and D=0, for either Proc3a or Proc3b.
  - Later assignment in procedure effectively overwrites earlier assignment
    - E will be updated with 0, but then by 1; so E is 1 at the end of the simulation cycle.

Recall FSM output assignment example, in which default assignments were added before the case statement.

Simulation cycle (revised)

- Set time to next time at which a procedure resumes
- Execute active procedures
- Update variables with schedule values

Assume B is 0.
Proc1:

```
B <= ~B;
```

Proc2:

```
A <= B;
```

A will be 0, not 1.

Proc3a: ← Same → Proc3b:

```
C <= ~C;        D <= C;
D <= C;         C <= ~C;
```

Proc4:

```
E <= 0;
...
E <= 1;
```