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 I[3], I[2], I[1], I[0], etc.
    - Most-significant bit is on left
    - Assign with binary constant (more on next slide)

```verilog
module Reg4(I3, I2, I1, I0, Q);
    input I3, I2, I1, I0;
    output [3:0] Q;
    wire [3:0] Q;
    reg [3:0] Q;
    input [3:0] I;
    assign Q = I;
    Q <= 4'b0000;
endmodule
```

**Constants**
- Binary constant
  - 4'b0000
- Other constant bases possible
  - Decimal base
  - Octal base
  - Hexadecimal base
    - 12'hFA2
    - Size is always in bits, and optional
    - For decimal constant, size and 'd optional
      - 12b1111_1010_0010
      - For decimal constant, size and 'd optional

```verilog`
module Reg5(Clk);
    input Clk;
    output [3:0] Q;
    reg [3:0] Q;
    input [3:0] I;
    always @(posedge Clk) begin
        Q <= 4'b0000;
    end
endmodule

module Reg6(Clk);
    input Clk;
    output [3:0] Q;
    reg [3:0] Q;
    input [3:0] I;
    always @(posedge Clk) begin
        Q <= 4'b0000;
    end
endmodule
```

**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
  - Other provides values for inputs Rst and I (i.e., vectors)
    - Initial procedure executes just once, does not repeat
    - Note: always procedure repeats

```verilog`
module Testbench();
    `timescale 1 ns/1 ns
    module Reg4(Clk);
    input Clk;
    output [3:0] Q;
    reg [3:0] Q;
    always begin
        Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);
        wire [3:0] Q_s;
        reg Clk_s, Rst_s;
        reg [3:0] I_s;
        I_s <= 4'b1111;
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        I_s <= 4'b1010;
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        I_s <= 4'b0000;
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        I_s <= 4'b0000;
        Rst_s <= 1;
        #10;
        Clk_s <= 1;
        #10;
        Clk_s <= 0;
    end`
endmodule
```

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
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
- Identification of a procedure
  - Always explicitly declare procedure
  - 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
- Simulation results
  - May not initialize all module inputs
  - Or simulator may initialize to default value. Switching simulators may cause design to fail.
  - Tip: Immediately initialize all module inputs when first writing procedure

Common Pitfalls

- Forgetting to explicitly declare a wire
  - E.g., Q_s
  - Verilog implicitly declares identifier as 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
  - Better to avoid use of Verilog's implicit declaration shortcut

Common Pitfalls

- Event control @ (posedge Clk_s)
  - May be prepared to statement to synchronize execution with event occurrence
  - Statement may be just "@" as in
  - In previous examples, the statement was a sequential block (begin-end)
  - Text vectors thus don't include the clock's period hard coded
  - May be prepended to statement
  - Only one procedure should write an identifier used in a port connection
  - E.g., Q_s
  - Verilog implicitly declares identifier as 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
  - Better to avoid use of Verilog's implicit declaration shortcut
Finite-State Machines (FSMs)—Sequential Behavior

• Code will be explained on following slides.
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
  - Result: State to FSM’s initial state, \( S_{\text{Off}} \)

- **String argument is printed literally...**
  - $\text{display}$ – 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, need a file, get the current simulation time, etc.
    - Starts with $ to distinguish from regular Verilog
    - To write to a display, read a file, get the current simulation time, etc.
  - Solution 1
    - FSM outputs should be combinational function of current state (for Moore FSM)
  - Solution 2
    - Not assigning output in given state means previous value is remembered
    - Output has memory
    - Behavior is not an FSM
  - Code should now be clear

- **Self-Checking Testbenches**
  - Common Pitfall: Not Assigning Every Output in Every State
    - Use if statements to check for expected values
  - Use self-checking testbench
  - Example:
    - If a check fails, print error message
  - Probably clearer to understand and remember state means previous value is remembered

- **Finite-State Machines (FSMs) Modules with Multiple Procedures and Shared Variables**
  - Reading waveforms is error-prone
  - Use self-checking testbench
  - Example:
    - Use if statements to check for expected values
    - If a check fails, print error message
  - Enumerate all states
    - Use self-checking testbench
  - Example:
    - If a check fails, print error message

- **Self-Checking Testbenches**
  - Code should now be clear
  - Use if statements to check for expected values
  - If a check fails, print error message
  - Enumerate all states
    - Use self-checking testbench
  - Example:
    - If a check fails, print error message

- **Forcing Outputs**
  - Use self-checking testbench
  - Example:
    - If a check fails, print error message
  - Enumerate all states
    - Use self-checking testbench
  - Example:
    - If a check fails, print error message
The Simulation Cycle

- Instructive to consider how an HDL simulator works
- HDL simulation is complex; will 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

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

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
- Time in main:
  - Still 10 ns; Clk just changed to 1 (P3 activates)
- Procedures
  - P1: S <= 1, stop, activate when Clk changes to 1 again
  - P2: Activate when S changes.
  - P3: Activate when Time is 20 ns.
- Variables
  - Q: 0 (~S), stop, activate when S changes.
  - Clk: 0 to 1, to 0
  - S: 0 to 1

Start of simulation

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

Variables

- Q: 0 (~S), stop, activate when S changes.
- Clk: 0 to 1, to 0
- S: 0 to 1

Module SimEx1

```verilog
module SimEx1(Q);

reg Clk, S;
output reg Q;

// P1
always begin
  S <= 0;
  @ (posedge Clk);
  S <= 1;
  @ (posedge Clk);
  Q <= ~S;
  #10;
  Clk <= 0;
end

// P2
always @(S) begin
  S <= 1;
  @ (posedge Clk);
  S <= 1;
  @ (posedge Clk);
  Q <= ~S;
  #10;
  Clk <= 1;
  #10;
  Clk <= 0;
end

// P3
always begin
  Q <= ~S;
end

endmodule
```

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 procedures (in any order) until stops

Variables

- Q: 0 (~S), stop, activate when S changes.
- Clk: 0 to 1, to 0
- S: 0 to 1

Module SimEx1

```verilog
module SimEx1(Q);

reg Clk, S;
output reg Q;

// P1
always begin
  S <= 0;
  @ (posedge Clk);
  S <= 1;
  @ (posedge Clk);
  Q <= ~S;
  #10;
  Clk <= 0;
end

// P2
always @(S) begin
  S <= 1;
  @ (posedge Clk);
  S <= 1;
  @ (posedge Clk);
  Q <= ~S;
  #10;
  Clk <= 1;
  #10;
  Clk <= 0;
end

// P3
always begin
  Q <= ~S;
end

endmodule
```
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
  - Activate when Time is 20 ns.
  - Activate when S changes.
  - Activate when change on Clk to 1.

Procedures

```verilog
`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
```

Variable Updates

- Assignment using "=" ("non blocking assignment") doesn't change variable's value immediately
  - Instead, schedule 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
  - Proc2 schedules B to be 0
  - At end of simulation cycle, B is updated 1 and becomes 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
  - 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.

Simulation cycle (revised)

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

Variable Updates

- Assignment using "=" ("non blocking assignment") doesn't change variable's value immediately
  - Instead, schedule 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
  - Proc2 schedules B to be 0
  - At end of simulation cycle, B is updated 1 and becomes 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
  - 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.

Simulation cycle (revised)

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