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, 12, 11, 10)
  - Vector’s bits are numbered
    - Options: [0:3] [1:4], etc.
  - Size of vector is not implicit
  - Assign with binary constant (more on next slide)

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

- 4-bit register
- Structure may consist of connected flip-flops
  - Other constant bases possible
    - Decimal base
      - 4'b0000
    - Hexahedronal base
      - 'hFA2
    - Binary constant
      - 4'b0000

Constants

- Binary constant
  - 4'd0000
  - 4 bits, in octal, decimal, binary, hexadecimal base
- Other constant bases possible
  - Decimal base: 4'd0000
  - Octal base: 4'o1
  - Hexadecimal base: 4'hA
  - 0x0000 is just 0
  - 8_000_000
  - Size always in bits, and optional
  - For decimal constant, size and ‘d optional
    - 8'd255 or just 255
  - 8'b1111_1010_0010
  - For decimal constant, size and ‘d optional
  - 'd255
  - 8'hFA2

Testbench

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

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

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 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

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 initializes all module inputs when first writing procedure
Finite-State Machines (FSMs)—Sequential Behavior

Modules with Multiple Procedures and Shared Variables

• Code will be explained on following slides

Finite-State Machines (FSMs)—Sequential Behavior

• Modules has two procedures
  • One procedure for combinational logic
  • One procedure for state register

• One procedure for state register

FSM's CombLogc procedure

– Case statement describes states
– Case (State)
  • Execute corresponding statement when a begin-end block based on State's current value
  • Actions of the state

– State is $S_{On1}$
  • Executes statements for state On1, jumps to endcase
Finite-State Machines (FSMs)—Sequential Behavior

- **FSM StateReg Procedure**
  - Similar to 4-bit register
  - Register for State is 2-bit vector reg
  - Procedure has synchronous reset
  - Results State to FSM’s initial state.

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

- **Capture structure**: Controller
- **Capture behavior**: FSM
- **Capture behavior (circuit)**
- **Capture behavior (architecture)**

- **Encode states**
  - Use FSM input values ("test vectors")
  - Wait for specific clock cycles

- **Register for State is 2-bit vector reg
  - Register for State is 2-bit vector reg

- **Combinational logic**
  - Use FSM input values ("test vectors")
  - Wait for specific clock cycles

- **Self-checking testbench**
  - Use if statements to check for expected values
  - Ex. If X_s == 0, print error message

- **Finite-State Machines (FSMs)—Sequential Behavior**
  - Reading waveforms is error-prone
  - Create self-checking testbench
  - Use if statements to check for expected values
  - Ex. If X_s == 0, print error message

- **Initial begin**
  - // CombLogic
  - // StateReg

- **Code should now be clear**
Top-Down Design – FSMs to Controller Structure

- Recall from Chapter 2
  - Top-down design
    - Capture behavior, and simulate
    - Capture structure (circuits), simulate again
    - Gets behavior right first, unfettered by complexity of creating structure
  - Capture behavior: FSM
  - Capture structure: Controller
    - Create architecture (state register and combinational logic)
    - Encode states
    - Create stable table (describes combinational logic)
    - Implement combinational logic

Common Pitfall: Not Assigning Every Output in Every State

- Solution 2
  - Assign default values before case statement
  - Later assignment in state overwrites default
  - 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 initialized to the unknown value x
  - Execute each procedure
    - In any order, until stops at a delay or event control

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

Verilog for Digital Design
Copyright © 2007
Frank Vahid and Roman Lysecky
### 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 Clk changes.
  - P3: Activate when Clk changes to 1 again

- **Variable Updates**
  - If E will be updated with 0, but then by 1; so E is 1 at the end of the simulation cycle.
  - In this case, set Time = 20 ns (P1 activates)

- **Simulation cycle (revised)**
  - Set time to next time at which a procedure activates
    - Still 10 ns. Clk just changed to 1 (P3 activates)
  - Execute active procedures until stops

- **Procedures**
  - P1: Activate when Time is 20 ns.
  - P2: Activate when Clk changes.
  - P3: Activate when Clk changes to 1 again

### 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
    - Proc schedules B to be 1, but does not change the present value of B
    - Proc schedules B to be 0 (the present value of B)
    - At end of simulation cycle, B is updated to 1 and not 0
  - Order of assignments to different variables in a procedure doesn't matter
    - Assume C case: Scheduled values will be C=1 and D=2
    - Forward Proc 3 or Proc 4
    - 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

### Resets

- **Reset – Behavior of a register when a reset input is asserted**
  - Good practice dictates having defined reset behavior for every register
  - Reset behavior should always have priority over normal register behavior
  - Usually clears register to 0s
  - May initialize to other value
    - e.g., state register of a controller may be initialized to encoding of initial state of FSM
  - Reset usually asserted externally at start of sequential circuit operation, but also to restart due to failure, user request, or other reason

### Simulation cycle (revised)

- **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 until stops

- **Procedures**
  - P1: Activate when Time is 20 ns.
  - P2: Activate when Clk changes.
  - P3: Activate when Clk changes to 1 again

- **Variable/net values translate to waveforms**
  - Waveforms of variables/net values relation to simulation time

### 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
    - Proc schedules B to be 1, but does not change the present value of B
    - Proc schedules B to be 0 (the present value of B)
    - At end of simulation cycle, B is updated to 1 and not 0
  - Order of assignments to different variables in a procedure doesn't matter
    - Assume C case: Scheduled values will be C=1 and D=2
    - Forward Proc 3 or Proc 4
    - 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

### Resets

- **Reset – Behavior of a register when a reset input is asserted**
  - Good practice dictates having defined reset behavior for every register
  - Reset behavior should always have priority over normal register behavior
  - Usually clears register to 0s
  - May initialize to other value
    - e.g., state register of a controller may be initialized to encoding of initial state of FSM
  - Reset usually asserted externally at start of sequential circuit operation, but also to restart due to failure, user request, or other reason
Synchronous Reset

- Previous examples used synchronous resets
  - Rst input only considered during rising clock

Asynchronous Reset

- Can also use asynchronous reset
  - Rst input considered independently from clock
    - Add `posedge Rst` to sensitivity list

Synchronous versus Asynchronous Resets

- Which is better – synchronous or asynchronous reset?
  - Hotly debated in design community
    - Each has pros and cons
      - e.g., asynchronous can still reset even if clock is not functioning,
        synchronous avoids timing analysis problems sometimes accompanying asynchronous designs
    - We won’t try to settle the debate here
  - What’s important is to be consistent throughout a design
    - All registers should have defined reset behavior that takes priority over normal register behavior
    - That behavior should all be synchronous reset or all be asynchronous reset
  - We will use synchronous resets in all of our remaining examples