Verilog - 1

More Verilog

- Registers
  - Counters
  - Shift registers
- FSMs
- Delayed assignments
- Test Fixtures

Verilog - 2

8-bit Register with Synchronous Reset

module reg8 (reset, CLK, D, Q);
input reset;
input CLK;
input [7:0] D;
output [7:0] Q;
reg [7:0] Q;

always @(posedge CLK)
if (reset)
  Q = 0;
else
  Q = D;
endmodule // reg8

Verilog - 3

N-bit Register with Asynchronous Reset

module regN (reset, CLK, D, Q);
input reset;
input CLK;
parameter N = 8; // Allow N to be changed
input [N-1:0] D;
output [N-1:0] Q;
reg [N-1:0] Q;

always @(posedge CLK or posedge reset)
if (reset)
  Q = 0;
else if (CLK == 1)
  Q = D;
endmodule // regN

Verilog - 4

Shift Register Example

// 8-bit register can be cleared, loaded, shifted left
// Retains value if no control signal is asserted
module shiftReg (CLK, clr, shift, ld, Din, SI, Dout);
input CLK;
input clr;
// clear register
input shift;
// shift
input ld;
// Load register from Din
input [7:0] Din;
// Data input for load
input SI;
// Input bit to shift in
output [7:0] Dout;
reg [7:0] Dout;

always @(posedge CLK)
if (clr)
  Dout <= 0;
else if (ld)
  Dout <= Din;
else if (shift) Dout <= [Dout[6:0], SI];
endmodule // shiftReg

Verilog - 5

Blocking and Non-Blocking Assignments

- Blocking assignments (\( q = a \))
  - variable is assigned immediately before continuing to next statement
  - new variable value is used by subsequent statements
- Non-blocking assignments (\( q <= a \))
  - variable is assigned only after all statements already scheduled are executed
  - value to be assigned is computed here but saved for later
  - usual use: register assignment
    - registers simultaneously take their new values after the clock tick
- Example: swap

always @(posedge CLK) begin
  temp <= a;
  a <= b;
end

always @(posedge CLK) begin
  temp <= b;
  b <= a;
end

Verilog - 6

Swap (continued)

- The real problem is parallel blocks
  - one of the blocks is executed first
  - previous value of variable is lost
- Use delayed assignment to fix this
  - both blocks are scheduled by posedge CLK

always @(posedge CLK) begin
  temp <= a;
  a <= b;
end

always @(posedge CLK) begin
  temp <= b;
  b <= a;
end
Non-Blocking Assignment

- Non-blocking assignment is also known as an RTL assignment
- if used in an always block triggered by a clock edge
- mimic register-transfer-level semantics - all flip-flops change together
- My rule: ALWAYS use <= in sequential (posedge clk) blocks

Counter Example

- Simple components with a register and extra computation
- Customized interface and behavior, e.g.
  - counters
  - shift registers
  - 8-bit counter with clear and count enable controls

Verilog FSM - Reduce 1s example

- Change the first 1 to 0 in each string of 1s
- Example Moore machine implementation

Verilog FSM for Reduce-1s example
Restricted FSM Implementation Style

- Mealy machine requires two always blocks
  - register needs posedge CLK block
  - input to output needs combinational block
- Moore machine can be done with one always block
  - e.g. simple counter
- Not a good idea for general FSMs
  - Can be very confusing (see example)
- Moore outputs
  - Share with state register, use suitable state encoding

Single-always Moore Machine
(Not Recommended!)

```verilog
module reduce (clk, reset, in, out);
input clk, reset, in;
output out;
reg reg [1:0] state; // state register
parameter zero = 0, one = 1, two = 2;
parameter zero = 0, one = 1, two = 2;
parameter zero = 0, one = 1, two = 2;
reg [1:0] state;
always @ (posedge clk)
  case (state)
    zero: begin
      reg [1:0] out;
      if (in)
        state = one;
      else
        state = zero;
    endcase
endmodule
```

Delays

- Delays are used for simulation only
- Delays are useful for modeling time behavior of circuit
- Synthesis ignores delay numbers
- If your simulation relies on delays, your synthesized circuit will probably not work
- #10 inserts a delay of 10 time units in the simulation

```verilog
module and_gate (out, in1, in2);
input in1, in2;
output out;
assign #10 out = in1 & in2;
endmodule
```

Verilog Propagation Delay

- May write things differently for finer control of delays
- always @ (posedge clk)
  case (state)
    zero: begin
      if (sum == 0)
        #6 zero = 1;
      else
        #3 zero = 0;
    endcase
endmodule
```

Initial Blocks

- Like always blocks
- execute once at the very beginning of simulation
- not synthesizable
- use reset instead

```verilog
module reduce (clk, reset, in, out);
input clk, reset, in;
output out;
reg reg [1:0] state; // state register
```

```verilog
module and_gate (out, in1, in2);
input in1, in2;
output out;
assign #10 out = in1 & in2;
endmodule
```
Tri-State Buffers

- 'Z' value is the tri-state value.
- This example implements tri-state drivers driving BusOut.

```verilog
module tri_state (EnA, EnB, BusA, BusB, BusOut);
  input EnA, EnB;
  input [7:0] BusA, BusB;
  output [7:0] BusOut;
  assign BusOut = EnA ? BusA : 8'bz;
  assign BusOut = EnB ? BusB : 8'bz;
endmodule
```

Test Fixtures

- Provides clock
- Provides test vectors/checks results
- Test vectors and results are precomputed
- Usually read vectors from file
- Models system environment
- Complex program that simulates external environment
- Test fixture can have all the language features
  - Initial, delay, read/write, file, etc.

```
Simulation

| Test Fixture (Specification) | Circuit Description (Synthesizable) |
```

Example Test Fixture

```verilog
module stimulus (a, b, c);
  parameter delay = 10;
  output a, b;
  reg [1:0] cnt;
  initial begin
    cnt = 0;
    repeat (4) begin
      #delay cnt = cnt + 1;
      end
      #delay $finish;
      end
  assign (a, b) = cnt;
endmodule
```

Simulation Driver

```verilog
module stimulus (a, b);
  parameter delay = 10;
  reg [1:0] cnt;
  initial begin
    cnt = 0;
    repeat (4) begin
      #delay cnt = cnt + 1;
      end
      #delay $finish;
      end
  assign (a, b) = cnt;
endmodule
```

```
Simulation Driver

<table>
<thead>
<tr>
<th>2-bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial block executed only once at start of simulation</td>
</tr>
<tr>
<td>Directive to stop simulation</td>
</tr>
<tr>
<td>Bundles two signals into a vector</td>
</tr>
</tbody>
</table>
```
Test Vectors

```
module testdata(clk, reset, data);
  input clk;
  output reset, data;
  reg [1:0] testVector [100:0];
  reg reset, data;
  integer count;
  initial begin
    loadmem("data.in", testVector);
    reset = 1;
    testVector[0] = data;
  end
  always @(posedge clk) begin
    count = count + 1;
    #1 (reset, data) = testVector[count];
  endmodule
```

Interpreted vs. Compiled Simulation

- Interpreted
  - data structures constructed from input file
  - simulator walks data structures and decided when something occurs
  - basic algorithm:
    - take an event from queue, evaluate all modules sensitive to that event, place new events on queue, repeat

- Compiled
  - input file is translated into code that is compiled/linked with kernel
  - basic algorithm:
    - same as above
    - except that new functions associated with elements are simply executed and directly place events on queue
    - overhead of compilation may be amortized over total simulation time and it's harder to make changes - need dynamic linking

Simulation Time and Event Queues

- Event queue
  - changes in signal values are "events" placed on the queue
  - queue is a list of changes to propagate
  - priority queue of pending events based on time of occurrence
  - multiple events on same signal can be on queue

- Time
  - advanced whenever an event is taken off the queue
  - advance to time of event
  - parallel activities are implicitly interleaved
  - what do we do about events with zero delay?

Verilog Simulation

- Interpreted vs. compiled simulation
- performance of the simulation
- Level of simulation
- accuracy of the model
- Relationship to synthesis
  - can all that can be simulated be synthesized?

Simulation Level

- Electrical
  - solve differential equations for all devices simultaneously to determine precise analog shape of waveforms
- Transistor
  - model individual transistors as switches - this can be close to electrical simulation if restricted to digital circuits
- Gate
  - use abstraction of Boolean algebra to view gates as black-boxes if only interested in digital values and delay
- Cycle or register-transfer
  - determine correct values only at clock edges, ignore gate delays if interested only in proper logical behavior of detailed implementation
- Functional (or behavioral) level
  - no interest in internal details of circuit implementation (just a program)

Verilog Time

- All computations happen in zero time unless there are explicit delays or waits in the code
  - #delay - blocks execution until that much time has passed
    - event placed on queue to wake up block at that time
  - @ or wait - waits for an event, e.g., @posedge clk and wait(x=0)
    - nothing happens until that event is taken off the queue
  - When an event is removed from the queue, all the blocks sensitive to it are evaluated in parallel and advance to their next blocking point (delay/wait)
  - Time advances as long as there are events to process
    - infinite loops are easy to write
    - use explicit $finish
    - use specified number of clock periods
Inertial and Transport Delays

- Inertial Delay
  - #3 X = A;
  - Wait 3 time units, then assign value of A to X
  - The usual way delay is used in simulation
  - Models logic delay reasonably
- Transport Delay
  - X <= #3 A;
  - Current value of A is assigned to X, after 3 time units
  - Better model for transmission lines and high-speed logic

A few requirements for CSE467...

- Draw data-flow diagrams
  - Algorithm ⇒ dataflow ⇒ datapath and control
  - Then instance Verilog modules into Xilinx schematics
- Draw state diagrams
  - And do the state encoding
  - One-hot
  - Simplifies combinational logic (reduces # of inputs)
  - More CLBs for state, but less for logic
  - Output based ⇒ use same bits for outputs and state
  - Don’t need CLBs to decode output from state
- Modularize your designs