More Verilog

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

N-bit Register with Asynchronous Reset
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) begin
    if (clr) Dout <= 0;
    else if (ld) Dout <= Din;
    else if (shift) Dout <= { Dout[6:0], SI };
end

endmodule // shiftReg

Verilog - 4
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

```verilog
always @(posedge CLK)
begin
  temp = B;
  B = A;
  A = temp;
end
```
```verilog
always @(posedge CLK)
begin
  A <= B;
  B <= A;
end
```
Swap (continued)

- The real problem is parallel blocks
  - one of the blocks is executed first
  - previous value of variable is lost

```verilog
always @(posedge CLK) begin
    A = B;
end

always @(posedge CLK) begin
    B = A;
end
```

- Use delayed assignment to fix this
  - both blocks are scheduled by `posedge CLK`

```verilog
always @(posedge CLK) begin
    A <= B;
end

always @(posedge CLK) begin
    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

```verilog
// this implements 3 parallel flip-flops
always @(posedge clk)
begin
    B = A;
    C = B;
    D = C;
end

// this implements a shift register
always @(posedge clk)
begin
    {D, C, B} = {C, B, A};
end

// this implements a shift register
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```

Verilog - 7
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
module count8 (CLK, clr, cntEn, Dout);
input CLK;
input clr; // clear counter
input cntEn; // enable count
output [7:0] Dout; // counter value
reg [7:0] Dout;

always @(posedge CLK)
  if (clr) Dout <= 0;
  else if (cntEn) Dout <= Dout + 1;
endmodule
Finite State Machines

- Recall FSM model

![Diagram of FSM model](image)

- Recommended FSM implementation style
  - Implement combinational logic using a one always block
  - Implement an explicit state register using a second always block
Verilog FSM - Reduce 1s example

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

```verilog
// State assignment
parameter zero = 0, one1 = 1, two1s = 2;

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

  // Implement the state register
  always @(posedge clk)
    if (reset) state = zero;
    else state = next_state;
endmodule
```

Verilog FSM - Reduce 1s example
Moore Verilog FSM (cont’d)

always @(in or state) 
  case (state)
    out = 0; // defaults
    next_state = zero;
    zero: begin // last input was a zero
      if (in) next_state = one1;
    end

    one1: begin // we've seen one 1
      if (in) next_state = twols;
    end

    twols: begin // we've seen at least 2 ones
      out = 1;
      if (in) next_state = twols;
    end
    // Don’t need case default because of default assignments
  endcase
endmodule
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state;     // state register
  reg next_state;
  parameter zero = 0, one = 1;

  always @(posedge clk)
    if (reset) state = zero;
    else state = next_state;

  always @(in or state)
    out = 0;
    next_state = zero;
    case (state)
      zero: begin // last input was a zero
        if (in) next_state = one;
        end
      one:     // we've seen one 1
        if (in) begin
          next_state = one; out = 1;
        end
      endcase
  endmodule
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
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg [1:0] state; // state register
parameter zero = 0, one1 = 1, twols = 2;
Single-always Moore Machine
(Not Recommended!)

```verilog
always @(posedge clk)
  case (state)
    zero: begin
      out = 0;
      if (in) state = one1;
      else state = zero;
    end
    one1:
      if (in) begin
        state = two1s;
        out = 1;
      end else begin
        state = zero;
        out = 0;
      end
    two1s:
      if (in) begin
        state = two1s;
        out = 1;
      end else begin
        state = zero;
        out = 0;
      end
    default: begin
      state = zero;
      out = 0;
    end
  endcase
endmodule
```

All outputs are registered

This is confusing: the output does not change until the next clock cycle
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

```verilog
assign #5 c = a | b;

assign #4 {Cout, S} = Cin + A + B;

always @(A or B or Cin)
    #4 S = A + B + Cin;
    #2 Cout = (A & B) | (B & Cin) | (A & Cin);

assign #3 zero = (sum == 0) ? 1 : 0;

always @(sum)
    if (sum == 0)
        #6 zero = 1;
    else
        #3 zero = 0;
```
Initial Blocks

- Like always blocks
  - execute once at the very beginning of simulation
  - not synthesizeable
    - use reset instead
Tri-State Buffers

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

```verilog
module tstate (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 all the language features
  - initial, delays, read/write files, etc.
Verilog Clocks

ClockGenerator

module clockGenerator (CLK);
    parameter period = 10;
    parameter howlong = 100;
    output CLK;
    reg CLK;

    initial begin
        CLK = 0;
        #(period/2);
        repeat (howlong) begin
            CLK = 1;
            #(period-period/2);
            CLK = 0;
            #(period/2);
        end
        $finish;
    end
endmodule // clockGenerator

Values assigned in blocks have to be declared “reg”

Stops the simulation
Verilog Clocks

- Another clock generator

```verilog
module clock_gen (masterclk);

`define PERIOD = 10;

output masterclk;
reg masterclk;

initial masterclk = 0;

always begin
    #`PERIOD/2
    masterclk = ~masterclk;
end

endmodule
```

- Use `define to make constants easier to find and change
- Use of initial and always blocks
Example Test Fixture

module stimulus (a, b, c);
    parameter delay = 10;
    output a, b, c;
    reg [2:0] cnt;

    initial begin
        cnt = 0;
        repeat (8) begin
            #delay cnt=cnt+1;
        end
        #delay $finish;
    end

    assign {c, a, b} = cnt;
endmodule

module full_addr1 (A, B, Cin, S, Cout);
    input A, B, Cin;
    output S, Cout;

    assign {Cout, S} = A + B + Cin;
endmodule

module driver;  // Structural Verilog connects test-fixture to full adder
    wire a, b, cin, sum, cout;
    stimulus stim (a, b, cin);
    full_addr1 fa1 (a, b, cin, sum, cout);

    initial begin
        $monitor (@ time=%0d cin=%b, a=%b, b=%b, cout=%d, sum=%d,
                  $time, cin, a, b, cout, sum);
    end
endmodule
module stimulus (a, b);
    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

2-bit vector

initial block executed only once at start of simulation

directive to stop simulation

bundles two signals into a vector
module testData(clk, reset, data);
    input clk;
    output reset, data;
    reg [1:0] testVector [100:0];
    reg reset, data;
    integer count;

    initial begin
        $readmemb("data.b", testVector);
        count = 0;
        { reset, data } = testVector[0];
    end

    always @(posedge clk) begin
        count = count + 1;
        #1 { reset, data } = testVector[count];
    end
endmodule

Test Vectors
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?
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 now functions associated with elements are simply executed and directly place events on queue
    - overhead of compilation must be amortized over total simulation time and its harder to make changes - need dynamic linking
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)
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 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