Design of Digital Circuits and Systems
Finite State Machine Review

Instructor: Justin Hsia

Teaching Assistants:

Colton Harris

Gayathri Vadhyan

Lancelot Wathieu

Deepti Anoop

Jared Yoder

Matthew Hung

Relevant Course Information

❖ hw1 due on Monday (4/1)
  - Homework can be completed in groups of up to **4**

❖ Lab 1 report due Friday (4/5)
  - Labs can be completed in groups of up to **2**

❖ Lab demos:
  - Lab demo sign up sheet sent out soon (check with partner)
  - 15 minutes for demos, early labs will be quicker
  - Make sure LabsLand is set up and synthesized *beforehand*

❖ Quiz 1 is Thursday, April 4 in last 25 min of lecture
  - Draw FSM state diagram & make design decisions

Lecture 1 Review

❖ Useful operators:
  ▪ Ternary operator: `<cond> ? <then> : <else>`
  ▪ Concatenation: `{sig, …, sig}`
  ▪ Replication: `{n{m}}`

❖ A **parameter** is a named constant

```
parameter N = 8;          // bus width
parameter period = 100;   // timing constant
```

❖ A parameterized module:
  ▪ `module <name> #(<parameter list>) (<port list>);`
  ▪ Parameters can be given default values
    • *e.g.,* `#(parameter N = 8)`

Review Question

❖ There are two forms of bit extensions: zero-extension (add 0s) and sign-extension (copy MSB)

❖ Write out SystemVerilog pseudocode for a parameterized *extender* module

  ▪ Inputs `sign` (1 bit), `in` (M bits); output `out` (N bits > M)

  ▪ `out` should either be the sign-extended version of `in` (`sign` = 1) or the zero-extended version of `in` (`sign` = 0)

Lecture Outline

❖ **SystemVerilog Review & Tips (Cont.)**

❖ FSMs

❖ Testbenches

Structural vs. Behavioral Revisited

❖ Not a strict definition of these terms, so exact classification is not that important

❖ Structural:

  ▪ Instantiating modules (library and user-defined) and defining port connections

  ▪ `assign`: continuous assignment

    • Used with nets

Verilog Procedural Blocks

❖ A *procedural block* is made up of behavioral code in the form of procedural statements whose effects are interpreted sequentially

  ▪ The block itself is awakened/triggered in a non-sequential manner

❖ `initial`: block triggered once at time zero

  ▪ Non-synthesizable (*i.e.*, for simulation/testbenches only)

❖ `always`: block triggered by a *sensitivity list*

  ▪ Any object that is assigned a value in an `always` statement must be declared as a variable (*e.g.,* `logic` or `reg`).

UNIVERSITY *of* WASHINGTON

SystemVerilog Procedural Blocks

❖ SystemVerilog introduced variants on always that are generally more robust and more specialized

❖ `always_comb`: intended for combinational logic
  ▪ Sensitivity list is automatically built

❖ `always_latch`: intended for latch-based logic
  ▪ Sensitivity list is automatically built

❖ `always_ff`: intended for sequential logic (*i.e.,* synchronous/clocked)
  ▪ Sensitivity list must be specified

Latch vs. Flip-Flop

❖ Both are bistable multivibrators (2 stable states) that can store information

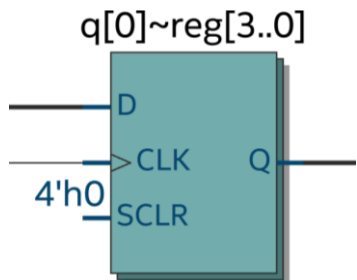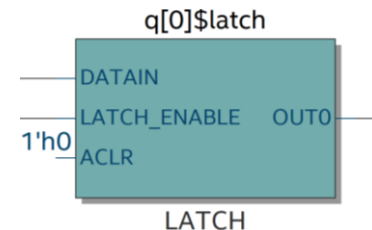❖ A latch is *asynchronous*; a flip-flop is *edge-triggered*

```systemverilog
module my_latch(input  logic       clk,
                input  logic [3:0] d,
                output logic [3:0] q);

    always_latch
        if (clk) q <= d;

endmodule
```


q[0]$latch

DATAIN
LATCH_ENABLE   OUT0
1'h0 ACLR

LATCH

q[0]~reg[3..0]

D
>CLK   Q
4'h0 SCLR

```systemverilog
module my_ff(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

    always_ff @(posedge clk)
        q <= d;

endmodule
```

Inferred Latches

❖ *Warning:* easy to write code with inadvertent latches

  ▪ Check your synthesis output for "Inferred latch"

  ▪ Usually from *incomplete assignments* – unspecified branch infers latch behavior

❖ **Question:** which of the following will synthesize and, if so, what will the hardware look like?

```
always_latch
    if (clk) q <= d;
```

```
always_comb
    if (clk) q = d;
```

```
always_latch
    if (clk) q <= d;
    else     q <= ~d;
```

```
always_comb
    if (clk) q = d;
    else     q = ~d;
```

  ▪ <u>Demo</u>: Tools → "Netlist Viewers" → "RTL Viewer"

case **Statement**

- ❖ Create combinational logic and is easier to read than lots of if/else statements
  - Must always be inside an always block
  - Each case has an implied C-style break

```systemverilog
module seven_seg(bcd, segs);

    input  logic [3:0] bcd;
    output logic [6:0] segs;

    always_comb
        case (bcd)
        //              abc_defg
        0: segs = 7'b011_1111;
        1: segs = 7'b000_0110;
        2: segs = 7'b101_1011;
        3: segs = 7'b100_1111;
        4: segs = 7'b110_0110;
        5: segs = 7'b110_1101;
        6: segs = 7'b111_1101;
        7: segs = 7'b000_0111;
        8: segs = 7'b111_1111;
        9: segs = 7'b110_1111;

        endcase

endmodule
```

case **Statement**

❖ Create combinational logic and is easier to read than lots of if/else statements

- Must always be inside an always block

- Each case has an implied C-style break

- Remember to use default to avoid incomplete assignments!

```systemverilog
module seven_seg(bcd, segs);

   input  logic [3:0] bcd;
   output logic [6:0] segs;

   always_comb
      case (bcd)
      //              abc_defg
      0: segs = 7'b011_1111;
      1: segs = 7'b000_0110;
      2: segs = 7'b101_1011;
      3: segs = 7'b100_1111;
      4: segs = 7'b110_0110;
      5: segs = 7'b110_1101;
      6: segs = 7'b111_1101;
      7: segs = 7'b000_0111;
      8: segs = 7'b111_1111;
      9: segs = 7'b110_1111;
  ⟶   default:  segs = 7'bX;
      endcase

endmodule
```

Other SystemVerilog Resources

❖ SystemVerilog Language Reference Manual

- On website, Verilog → Reference Manual
- 586 pages…

❖ SystemVerilog articles

- https://www.systemverilog.io/
- http://www.verilogpro.com/
- https://www.chipverify.com/systemverilog/systemverilog-tutorial

❖ One style guide for SystemVerilog

- https://www.systemverilog.io/styleguide
- We won't enforce, but good guidelines

UNIVERSITY *of* WASHINGTON

Technology Break

Lecture Outline

- ❖ SystemVerilog Review & Tips (Cont.)

- ❖ **Finite State Machine Design**

- ❖ Testbenches

Finite State Machines (FSMs)

❖ A convenient way to conceptualize computation over time using a *state transition diagram*

  ▪ Consists of a *set of states*, an *initial state*, and a *transition function*

❖ FSM implementations come in 3 blocks:

  ▪ State register (SL)

  ▪ Next state logic (CL)

  ▪ Output logic (CL)

17
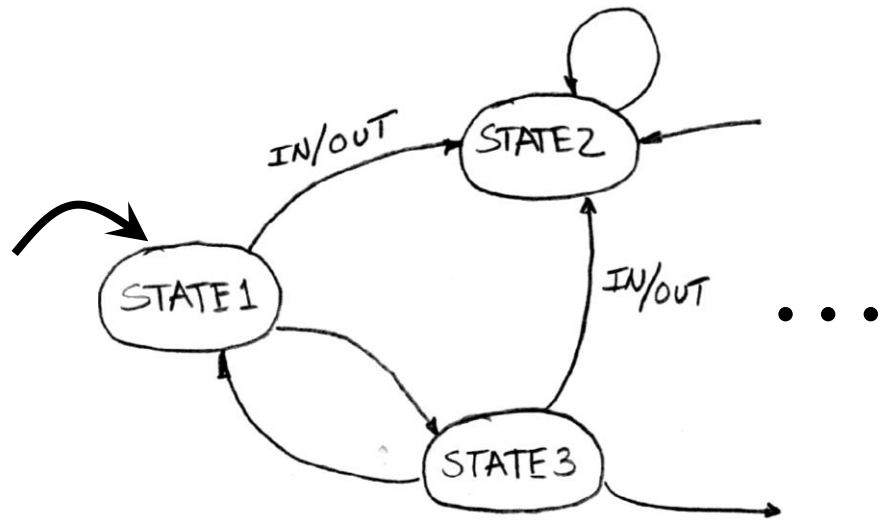
FSM Implementation Notes

- ❖ States must be assigned a binary encoding
  - ▪ More readable by using parameters or an `enum`
  - ▪ Encoding choices can affect logic simplification

- ❖ Reset signal can be synchronous (responds to `clk`) or asynchronous (responds to `reset`)
  - ▪ Determined by whether or not `reset` is in sensitivity list

- ❖ State logic (next state logic + state update) can be written as 1 combined block or 2 separate blocks

- ❖ If input is asynchronous, may want to add a two-flip-flop *synchronizer* to deal with metastability

FSM SystemVerilog Design Pattern

❖ Which, if any, construct(s) would you expect to use for each of the following basic sections of a module that implements an FSM?

▪ *// define states and state variables*
  initial   assign   always_comb   always_ff   None

▪ *// next state logic (ns)*
  initial   assign   always_comb   always_ff   None

▪ *// output logic*
  initial   assign   always_comb   always_ff   None

▪ *// state update logic (ps)*
  initial   assign   always_comb   always_ff   None

FSM Example: String Manipulator

❖ Takes in a stream of inputs and removes the *second* 1 from every consecutive string of 1's.
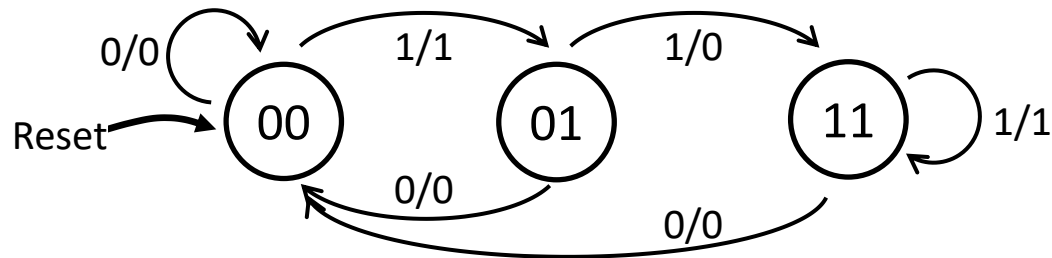
- Example inputs: 0 1 0 1 1 0 1 1 1 0 1 1 1 1 …
  outputs:

String Manipulator FSM

```systemverilog
module fsm (input  logic clk, reset, in,
            output logic out);

    // present and next state
    enum logic [1:0] {S0, S1, S3} ps, ns;

    // next state logic
    always_comb
        case (ps)
            S0: if (in) ns = S1;
                else    ns = S0;
            S1: if (in) ns = S3;
                else    ns = S0;
            S3: if (in) ns = S3;
                else    ns = S0;
        endcase

    // output logic
    assign out = in & (ps[1] | ~ps[0]);

    ...
```

```systemverilog
    ...

    // sequential logic (DFFs)
    // synchronous reset
    always_ff @(posedge clk)
        if (reset)
            ps <= S0;  // reset state
        else
            ps <= ns;

endmodule  // fsm
```
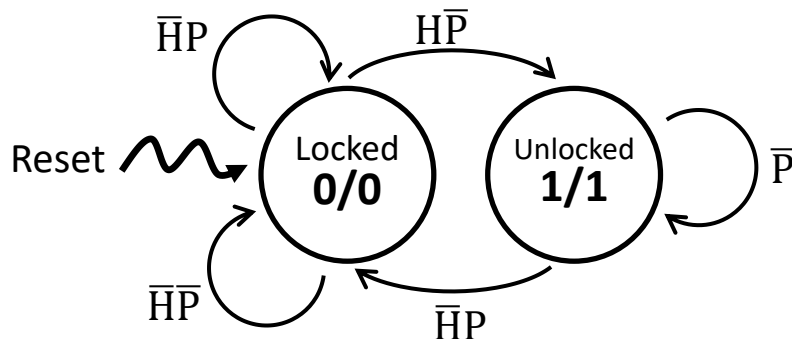
Moore vs. Mealy

❖ **Moore** machines define their outputs based on states (⬭ 00/1) and **Mealy** machines define outputs based on transitions (―0/1→)

- Mealy machines are more *flexible*
  - Moore outputs are function of state; Mealy outputs are function of state *and inputs*
- All FSMs can be expressed in either form, but some systems are more naturally expressed one way versus the other
  - Feel free to use either in this class if not specified
  - However, there *are* implementation differences!

Mealy ↔ Moore Conversions

Not testable material

- ❖ **Moore → Mealy:** copy the state output to every transition *entering* the state

- ❖ Example: FSM for a *turnstile*, which is locked until someone swipes their Husky ID (input H) and then locks once you push through (input P) the unlocked gate. Outputs a light that glows red (0) or green (1).

Mealy ↔ Moore Conversions
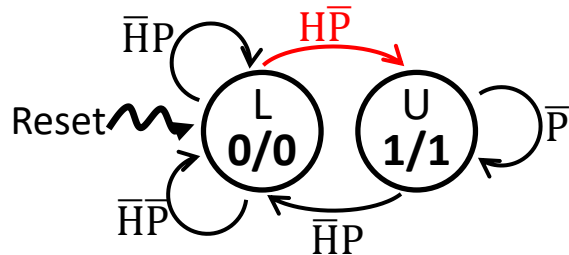
❖ **Mealy → Moore**: more complicated process; if incoming transitions differ in output, may need to "split" the state

❖ <u>Example</u>: the threeOnes FSM from Lecture 1
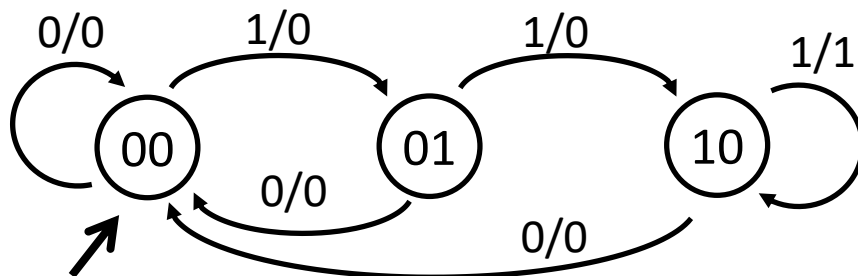
Moore vs. Mealy Outputs

❖ Compare a Moore and Mealy FSM for the turnstile. Complete the statements and waveform below, assuming no delays:
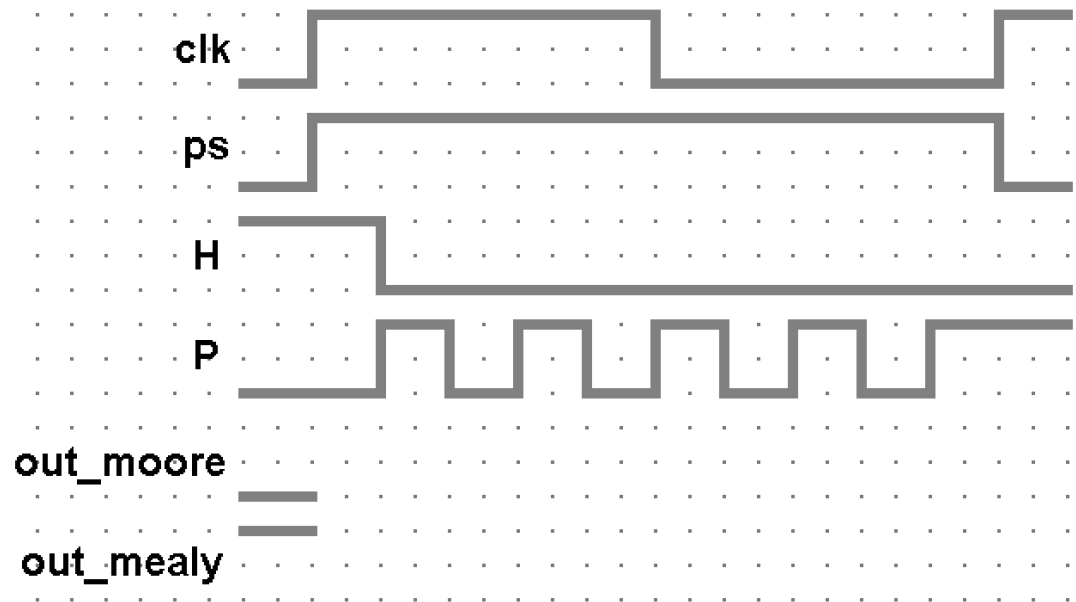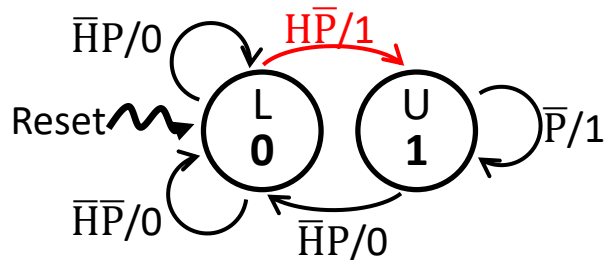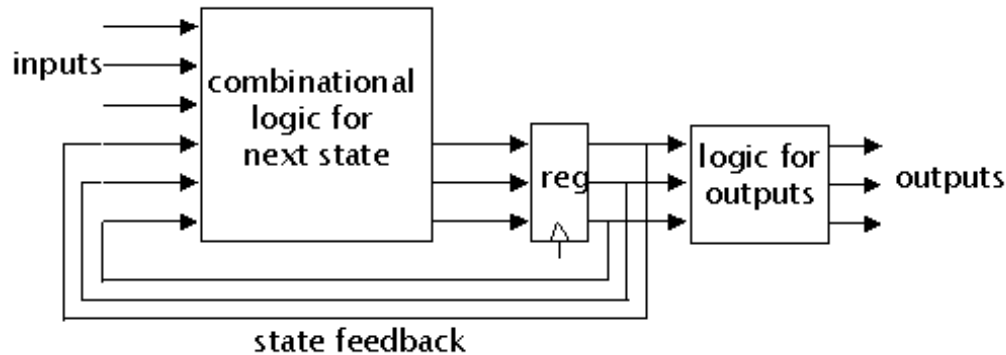

Moore:

```
assign out_moore =
assign out_mealy =
```
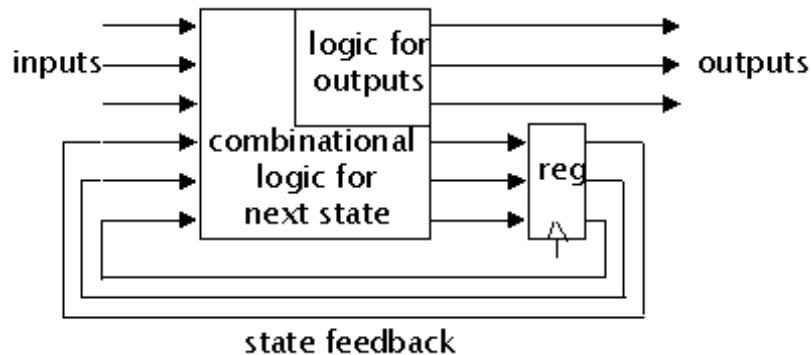
Mealy:

Moore vs. Mealy Outputs

❖ Moore:

- Outputs change synchronously with state changes

❖ Mealy:

- Input changes can cause *immediate* output changes

Lecture Outline

❖ SystemVerilog Review & Tips (Cont.)

❖ Finite State Machine Design

❖ **Testbenches**

Testbenches

❖ Special modules *needed for simulation only!*

  ▪ Software constraint to mimic hardware

❖ ModelSim runs entirely on your computer

  ▪ Tries to simulate your FPGA environment without actually using hardware – no physical signals available

  ▪ Must create fake inputs for FPGA's physical connections

    • *e.g.,* LEDR, HEX, KEY, SW, CLOCK_50

  ▪ Unnecessary when code is loaded onto FPGA

❖ Need to define both input signal combinations as well as their *timing*

Testbench Timing Controls

❖ Delay: `#<time>`
  - Delays by a specific amount of simulation time

❖ Edge-sensitive: `@(<pos/neg>edge <signal>)`
  - Delays next statement until specified transition on signal

❖ Level-sensitive Event: `wait(<expression>)`
  - Waits until `<expression>` evaluates to TRUE

❖ Stop simulation: `$stop;`

❖ Timescale: `` `timescale <time unit> / <precision> ``
  - *e.g.,* `` `timescale 1 ns / 1 ps ``

Extender Testbench
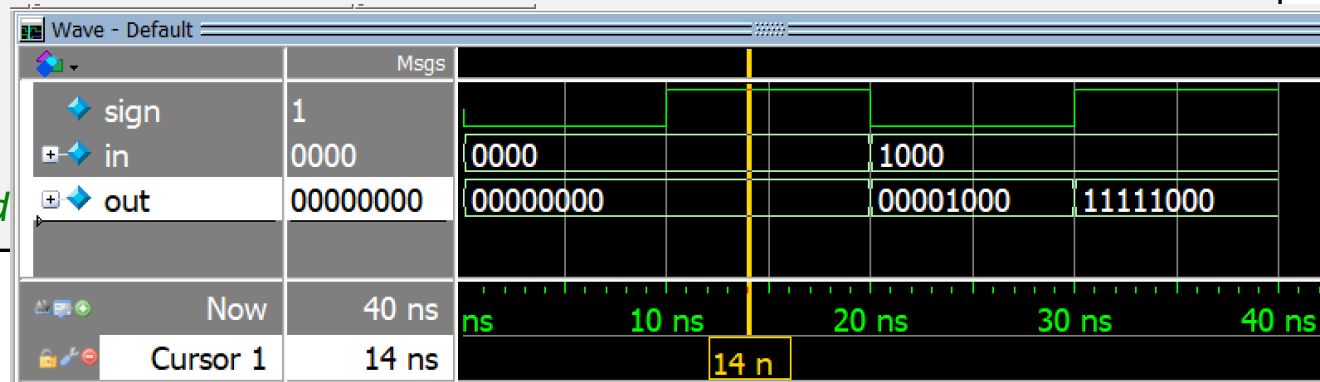
```systemverilog
`timescale 1 ns / 1 ns
module extender_tb();

    parameter M = 4, N = 8;
    logic [M-1:0] in;
    logic [N-1:0] out;
    logic sign;

    extender #(M, N) dut (.*);

    int i;
    initial begin
        for (i = 0; i < 2**2; i++) begin
            sign = i[0]; in = {i[1], {(M-1){1'b0}}}; #10;
        end  // for
        $stop;
    end  // initial

endmodule  // extend
```

FSM Testbench Notes

❖ Your main goal is to test *every transition* that we care about – may take extra clock cycles

❖ For simulation, you need to generate a clock signal

- Assume we have parameter clock_period;

Explicit Edges:

```
initial
    clk = 0;

always_comb begin
    #(clock_period/2) clk <= 1;
    #(clock_period/2) clk <= 0;
end
```

Toggle:

```
initial begin
    clk <= 0;
    forever #(clock_period/2) clk <= ~clk;
end
```

String Manipulator Testbench

```systemverilog
module fsm_tb();

    logic clk, reset, in, out;

    fsm dut (.*);

    // simulated clock
    parameter period = 100;
    initial begin
        clk <= 0;
        forever
            #(period/2)
            clk <= ~clk;
    end  // initial clock

    ...
```

```systemverilog
    ...

    initial begin
        reset <= 1; in <= 0; @(posedge clk);
        reset <= 0; in <= 0; @(posedge clk);
                    in <= 0; @(posedge clk);
                    in <= 1; @(posedge clk);
                    in <= 0; @(posedge clk);
                    in <= 1; @(posedge clk);
                    in <= 1; @(posedge clk);
                    in <= 0; @(posedge clk);
                    in <= 1; @(posedge clk);
                    in <= 1; @(posedge clk);
                    in <= 1; @(posedge clk);
                             @(posedge clk);
        $stop;  // end simulation
    end  // initial signals

endmodule  // fsm_tb
```
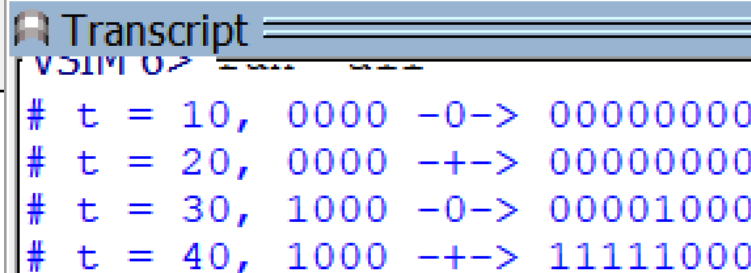
String Manipulator Waveforms
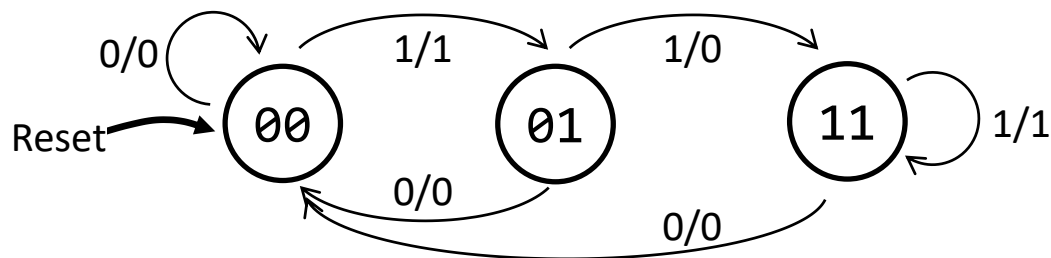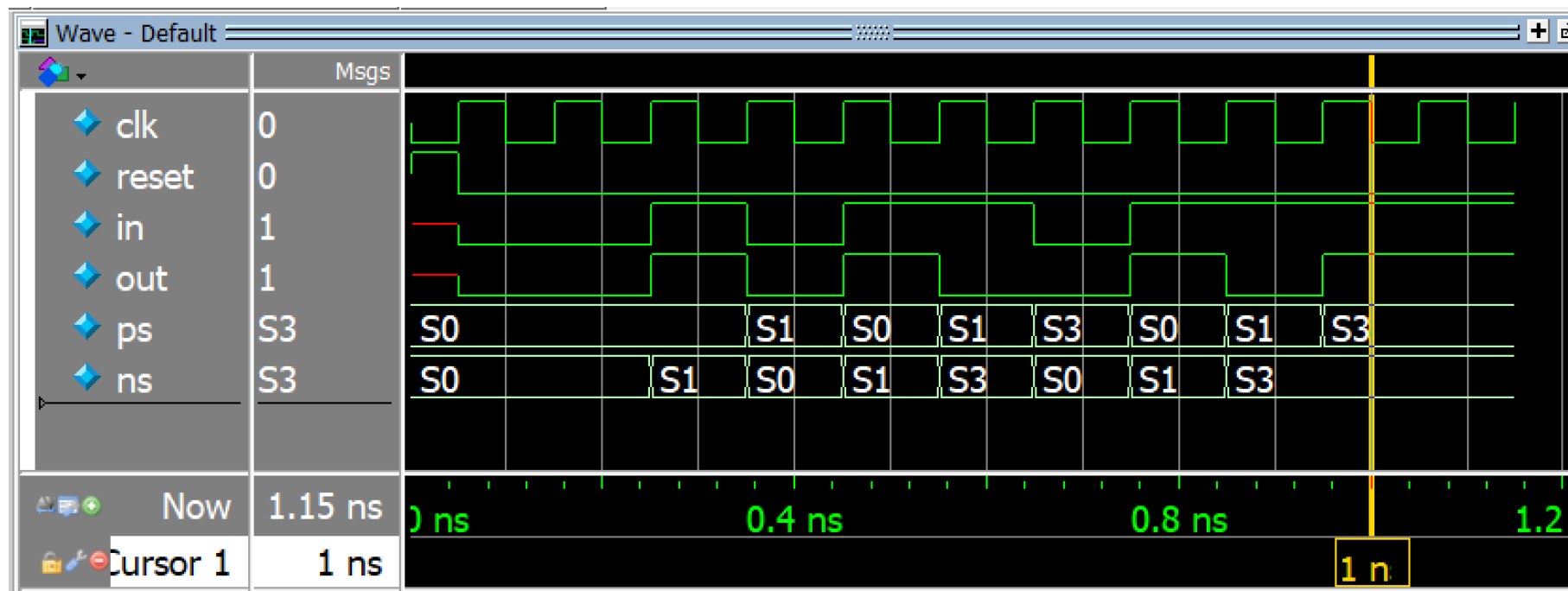
Checking Responses

❖ **Visually checking simulated waveforms quickly becomes impractical for large designs simulated over thousands of clock cycles**

  ▪ Displaying and explaining your waveforms for labs can be tedious

❖ **There are simulator-independent system tasks to write messages to the user/tester!**

  ▪ Look similar to `printf()` in C

    • `$<system_task>(<format_string>, <sig_1>, <sig_2>, …)`

  ▪ Will look at `$display` today and others later on

Checking Responses: `$display`

❖ Triggers once when encountered, prints the given format string and adds a new line:

```
// define test inputs
int i;
initial begin
   for (i = 0; i < 2**2; i++) begin
      sign = i[0]; in = {i[1], {(M-1){1'b0}}}; #10;
      $display("t = %0t, %b %s %b",
               $time, in, sign ? "-+->" : "-0->", out);
   end  // for
   $stop;
end  // initial
```

```
Transcript
VSIM 0>
# t = 10,  0000  -0->  00000000
# t = 20,  0000  -+->  00000000
# t = 30,  1000  -0->  00001000
# t = 40,  1000  -+->  11111000
```

Format Specifiers

Table 5.7: Format Specifiers.

Specifier	Meaning
%h	Hexadecimal format
%d	Decimal format
%o	Octal format
%b	Binary format
%c	ASCII character format
%v	Net signalstrength
%m	Hierarchical name of current scope
%s	String
%t	Time
%e	Real in exponential format
%f	Real in decimal format
%g	Real in exponential or decimal format

Table 5.8: Special characters.

Symbol	Meaning
\n	New line
\t	Tab
\\	\character
\"	" character
\xyz	Where xyz is are octal digits - the character given by that octal code
%%	% character

- Warning: these differ from the specifiers for `printf`
- The *minimum* field width is specified by numbers between the '%' and specifier letter
  - *e.g.*, %3d will pad out to 3 digits if necessary, %0d will show just the minimum number of digits needed