Outstanding Material from Lecture 1

EL GY 6463: Advanced Hardware Design

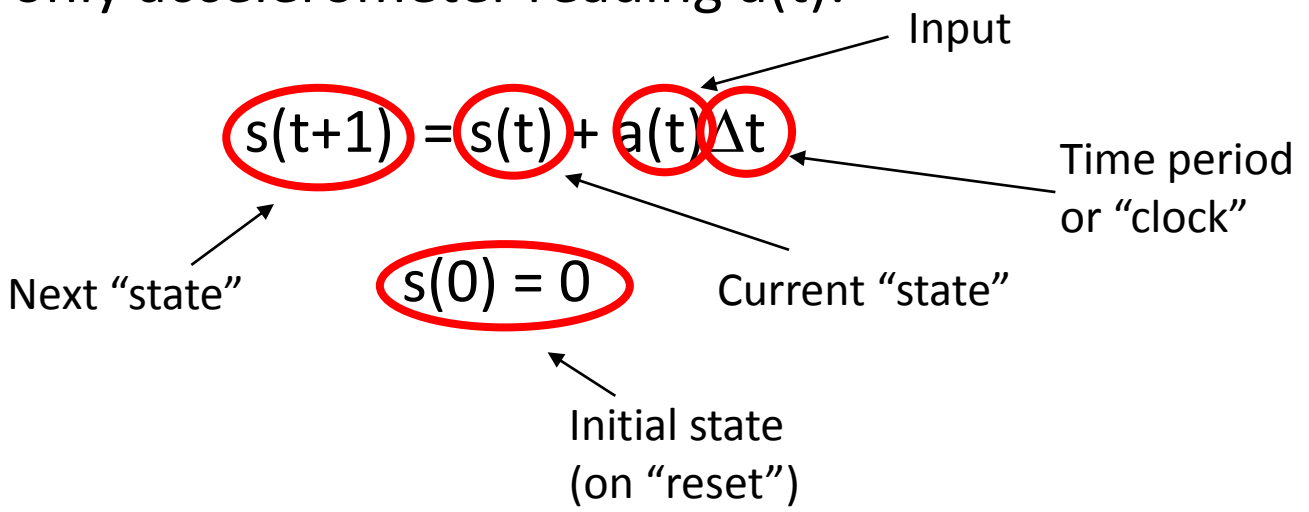
Instructor: Siddharth Garg

Finite State Machines: Introduction

- Combinational logic: output depends on the current input only
- Finite State Machine: output depends on input *and* current state

Example: what is my current speed?

- Given only speedometer reading s(t): output = s(t)
- Given only accelerometer reading a(t):



But is this a finite state machine? Why or why not?

FSM: Formal Model

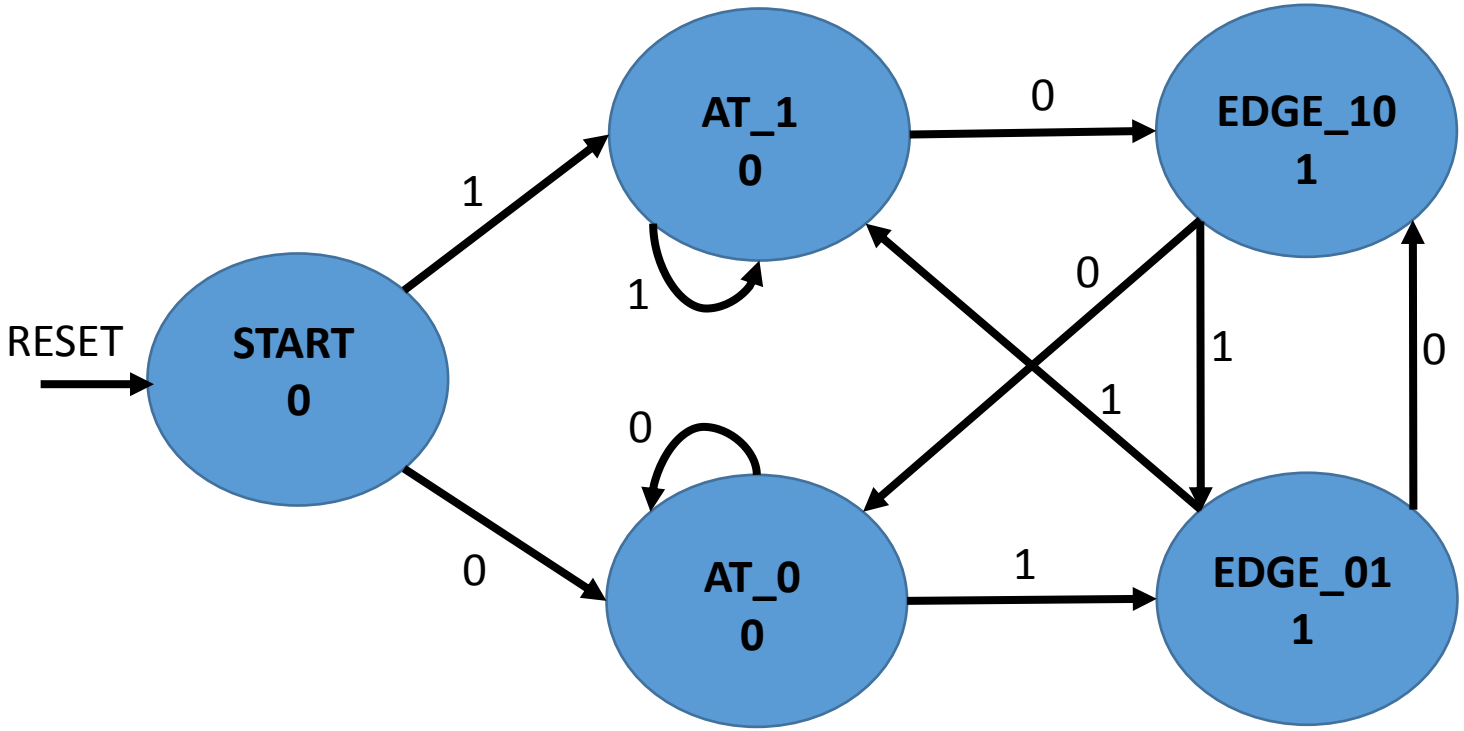
- A finite set of *states: S*
  - An initial state: $s_0 \in S$
- A finite set of *inputs:* Σ
- A finite set of *outputs:* Γ
- A state transition function $\delta : S \times \Sigma \rightarrow S$
- An output function ω
  - **Moore machine** $\omega : S \rightarrow \Gamma$ ← Output depends only on current state
  - **Mealy machine** $\omega : S \times \Sigma \rightarrow \Gamma$ ← Output depends on current state and input

FSM Example: Edge Detector

- Goal: detect 0->1 and 1->0 transitions in a sequence of inputs. Output a 1 when a transition occurs, zero otherwise.

  Input: 00101111100001000

  Output: 00111000010001100

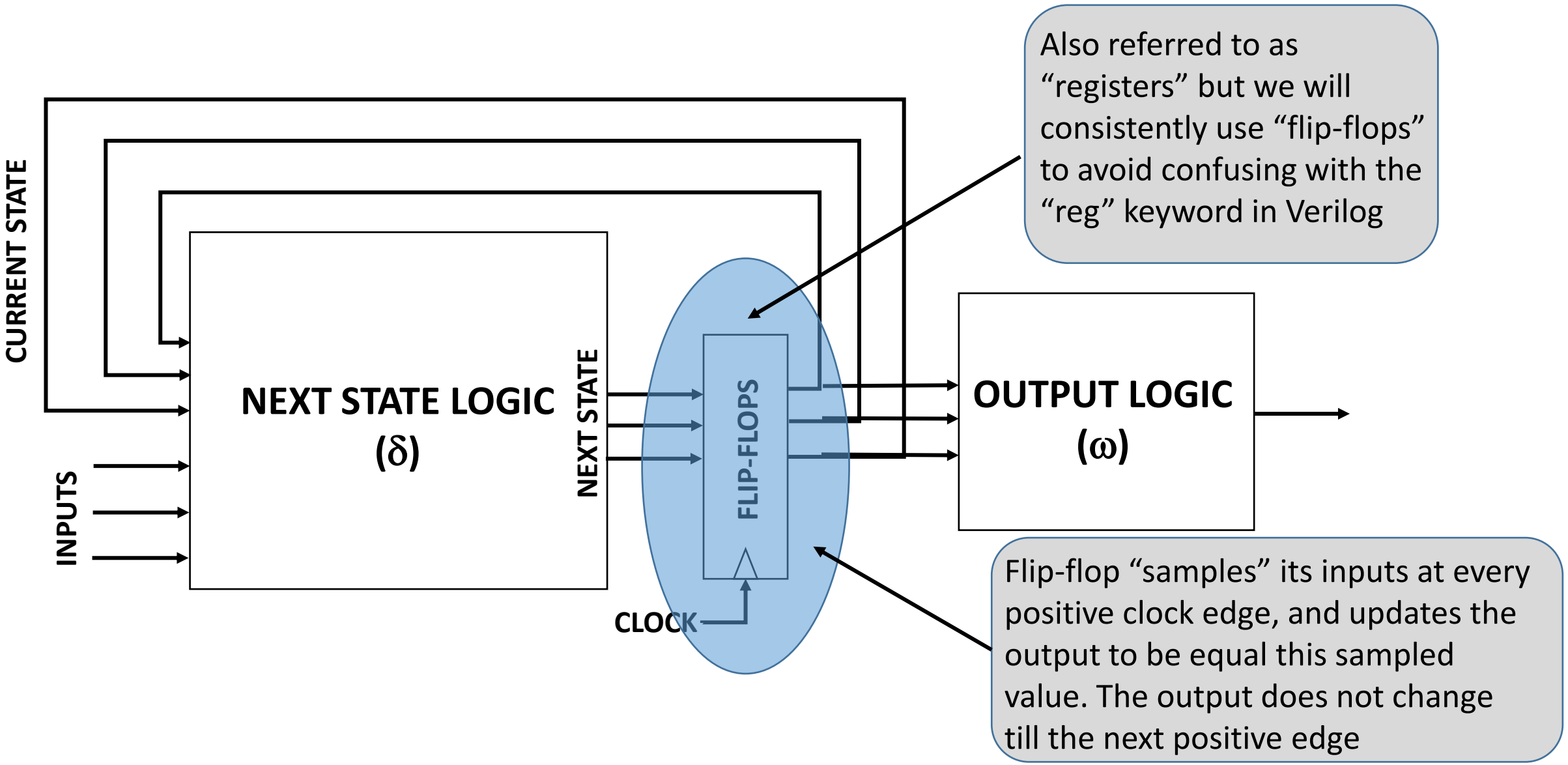


- *states S = {START, AT_1, AT_0, EDGE_10, EDGE_01}*
  - Initial state: s_0 = START
- *Inputs* $\Sigma = \{0,1\}$
- *Outputs* $\Gamma = \{0,1\}$

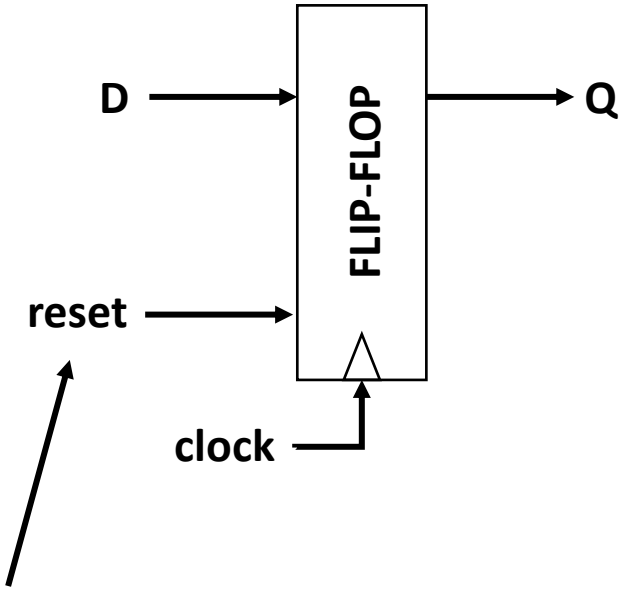
Is this a Mealy or a Moore machine?

http://courses.cs.washington.edu/courses/cse370/10sp/pdfs/lectures/15-VerilogIIPrint.pdf

Clocked Implementation of a Moore FSM



Flip-flops in Verilog



If reset = 1, output Q = 0.

- Asynchronous: Q = 0 as soon as reset becomes 1
- Synchronous: Q = 0 at next positive edge

D Flip-flop with Asynchronous Reset.

```
module dff ( input D, input reset,
             input clock, output reg Q);

always @ (posedge clock, posedge reset) begin
  if (reset)
     Q <= 0;
  else
     Q <= D;
end

endmodule
```

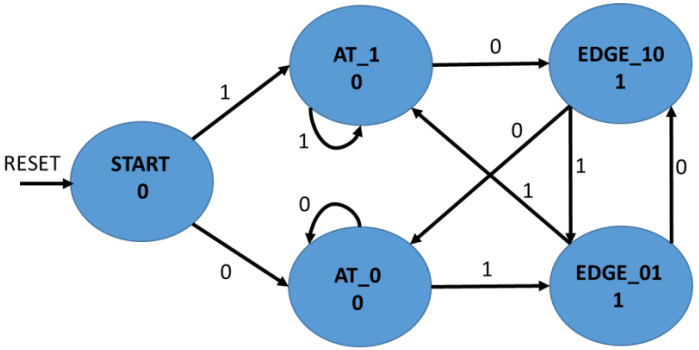
when clock transitions from 0 to 1

or reset transitions from 0 to 1

- **Non-blocking assignment** (remember that = is a blocking assignment)
- Rule of thumb: use non-blocking when always block is edge triggered. More on this later.

What about synchronous reset?

Edge Detector FSM in Verilog



- Let's implement the **sequential portion** first
  - At positive clock edge, current state <= next state

```
module edge_detect ( input in, input clock, input reset, output reg out);


                    Three bits. Why?
reg [2:0] current_state, next_state;

localparam START = 0, AT_1 = 1, AT_0 = 2, EDGE_10 = 3, EDGE_01 = 4;
//localparam START = 3'b000, AT_1 = 3'b001, AT_2 = 3'b010 … ;
always @ (posedge clock, posedge reset) begin
  if (reset)                                          Three bits.
     current_state <= START;
  else
     current_state <= next_state;
end

endmodule
```

Constant defined only within the scope of current module

Edge Detector FSM in Verilog (In-Class Exercise)

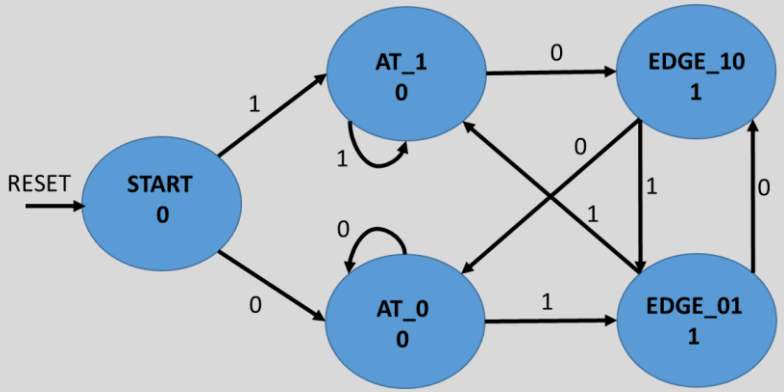
- Now let's implement the **next state and output logic**

```
module edge_detect (input in, input clock, input reset, output reg out);
// sequential code from previous slide
always @ (current_state, in) begin //always @ (*)
  out = 0;
  next_state = current_state;
  case (current_state)
    START: if (in) next_state = AT_1;
           else next_state = AT_0;
    AT_1:  if (~in) next_state = EDGE_10;
    AT_0:  if (in) next_state = EDGE_01;
    EDGE_10: begin
               out = 1;
               if (in) next_state = EDGE_01;
                else next_state = AT_0;
             end
    EDGE_01: //DO THIS
    default:  begin out = 1'bX; next_state = 3'bX; end

  endcase
end
endmodule
```

Similar to C case statement

Current_state = 5,6,7

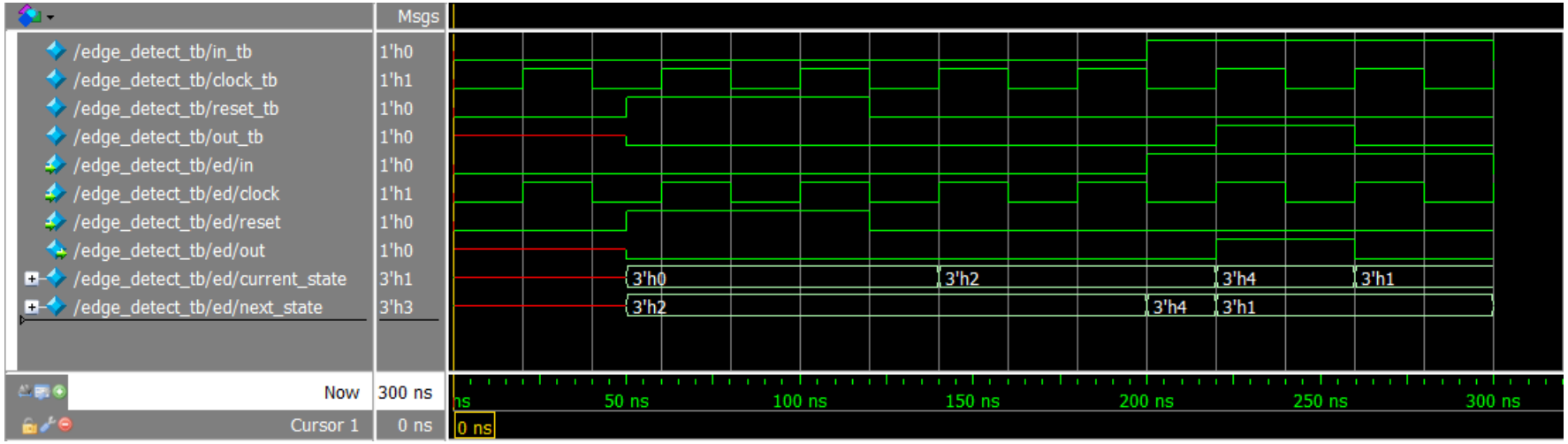
X: don't care



Solution

```
EDGE_01: begin
                out = 1;
                if (~in) next_state = EDGE_10;
                 else next_state = AT_1;
          end
```

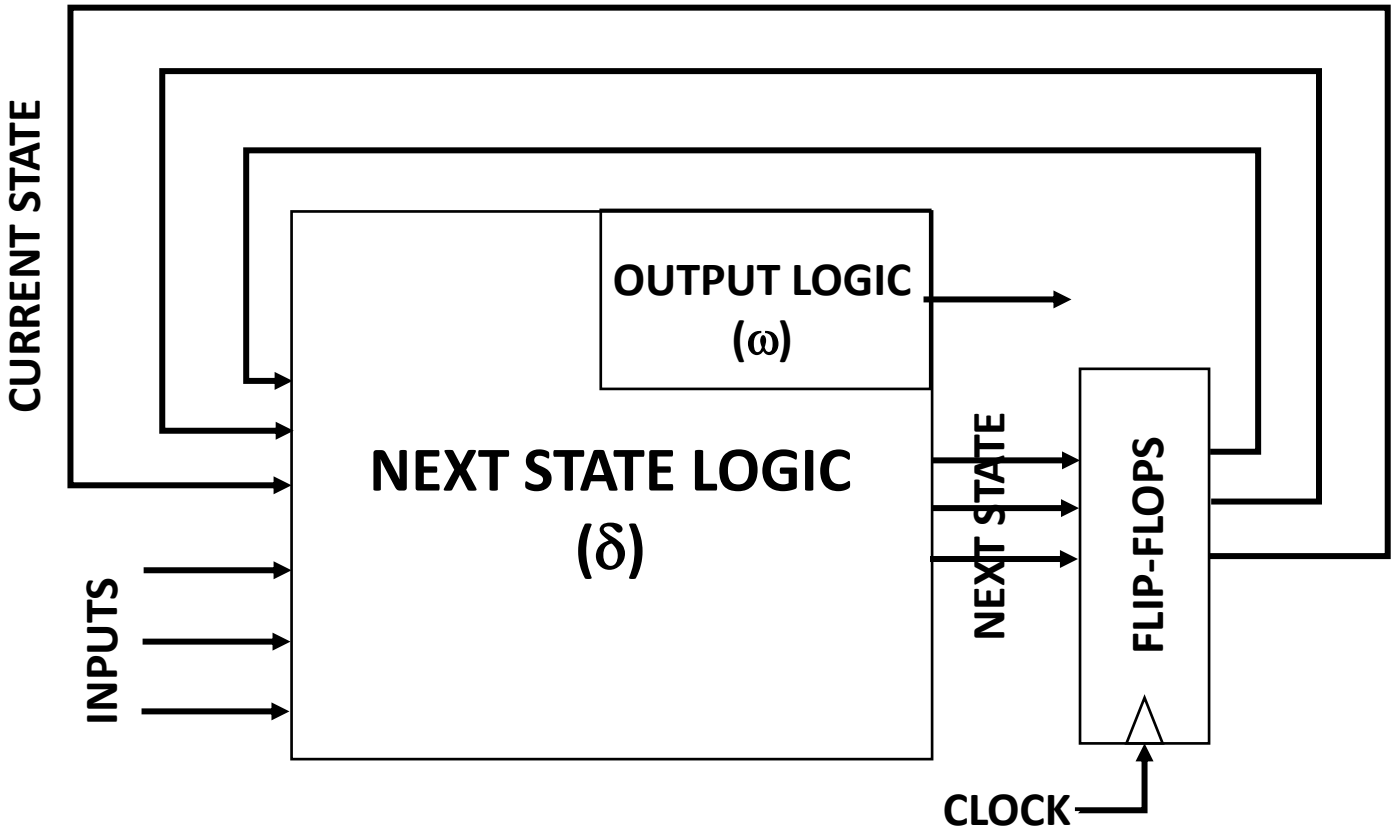
Simulation Output




```
VSIM 61> run
#                    0 in=0, clock=0, reset=0, out=x
#                   20 in=0, clock=1, reset=0, out=x
#                   40 in=0, clock=0, reset=0, out=x
#                   50 in=0, clock=0, reset=1, out=0
#                   60 in=0, clock=1, reset=1, out=0
#                   80 in=0, clock=0, reset=1, out=0
#                  100 in=0, clock=1, reset=1, out=0
#                  120 in=0, clock=0, reset=0, out=0
#                  140 in=0, clock=1, reset=0, out=0
#                  160 in=0, clock=0, reset=0, out=0
#                  180 in=0, clock=1, reset=0, out=0
#                  200 in=1, clock=0, reset=0, out=0
#                  220 in=1, clock=1, reset=0, out=1
#                  240 in=1, clock=0, reset=0, out=1
#                  260 in=1, clock=1, reset=0, out=0
#                  280 in=1, clock=0, reset=0, out=0
```

Mealy Machine

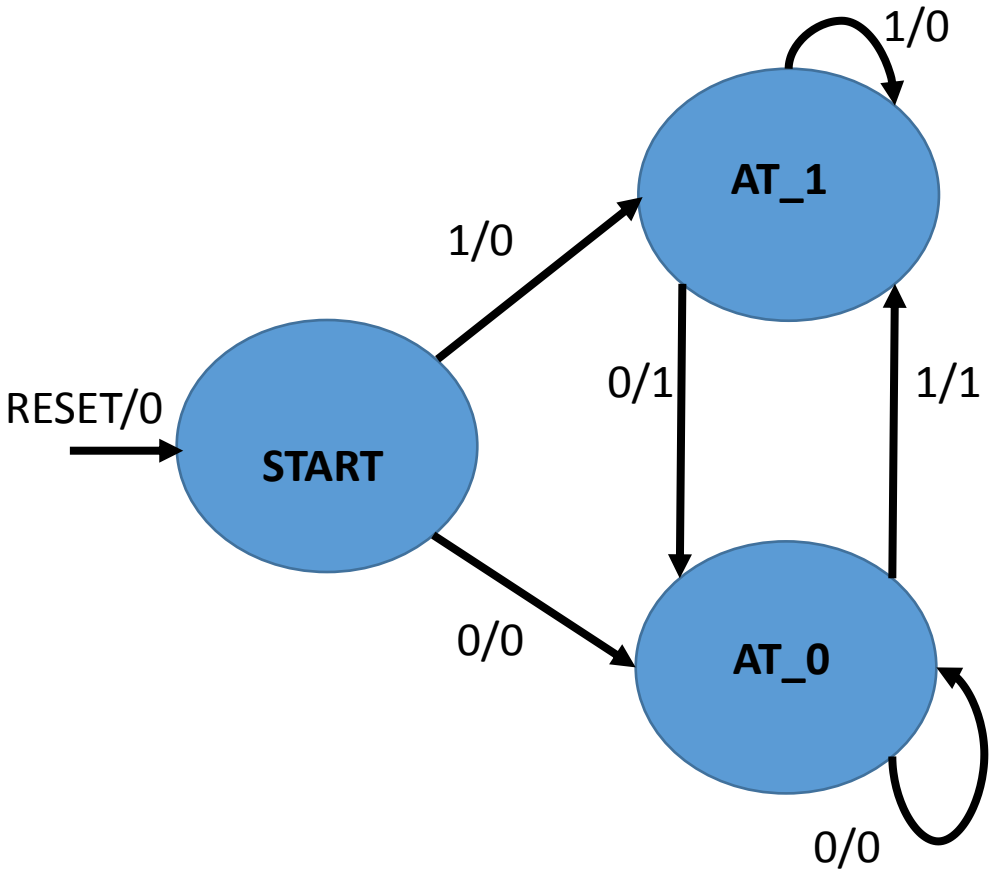
- Mealy machine: output depends on state *and* input



Direct path between inputs and outputs that does not pass through a flip-flop

- Pros:
  - Fewer states
  - Output more "responsive" to input
- Cons:
  - Unstable outputs
  - Cascading Mealy machines?

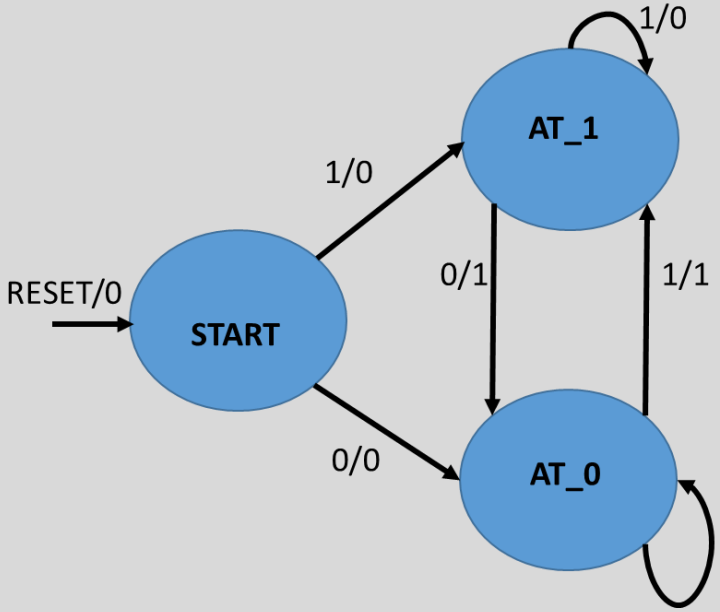
Mealy Machine: Verilog Implementation




```
module edge_detect ( input in, input clock,
                     input reset, output reg
out);


reg [1:0] current_state, next_state;

localparam START = 0, AT_1 = 1, AT_0 = 2;

always @ (posedge clock, posedge reset) begin
  if (reset)
     current_state <= START;
  else
     current_state <= next_state;
end
……
endmodule
```

Mealy Machine: Verilog Implementation (In Class Exercise)

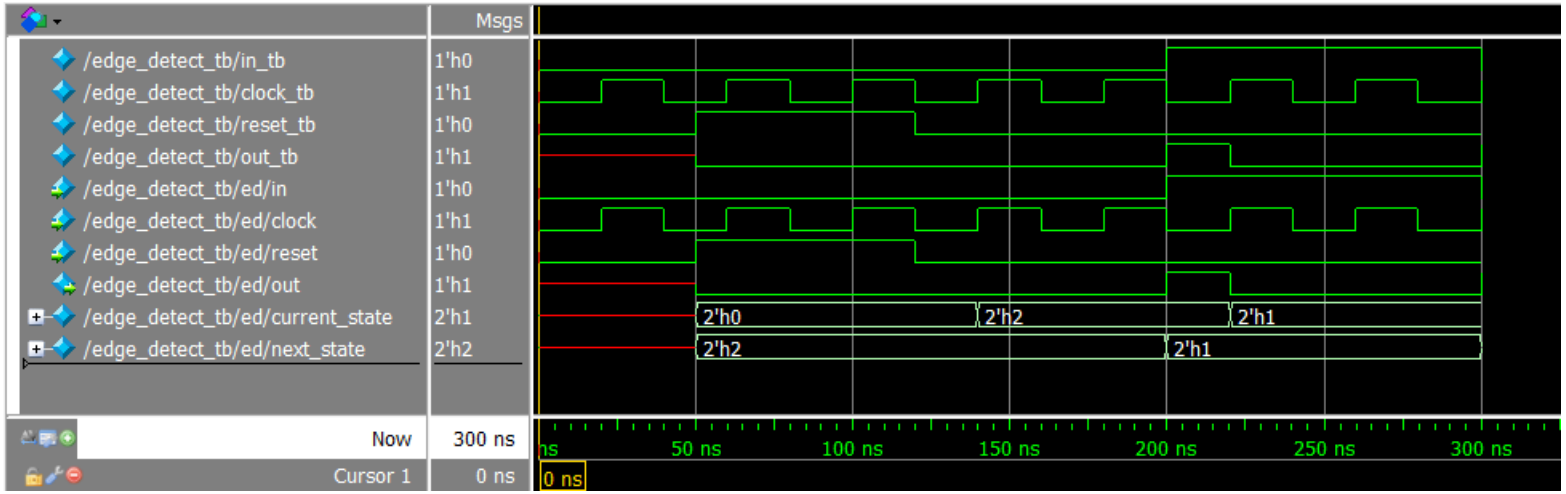
```
module edge_detect (input in, input clock, input reset, output reg out);
// sequential code from previous slide
always @ (current_state, in) begin //always @ (*)
  out = 0;
  next_state = current_state;
  case (current_state)
    START: if (in) next_state = AT_1;
           else    next_state = AT_0;
    AT_1:  begin
             if (~in) begin
                out = 1;
                next_state = AT_0;
             end
           end
   AT_0:          //DO THIS

   default:
endcase
end
endmodule
```



Solution

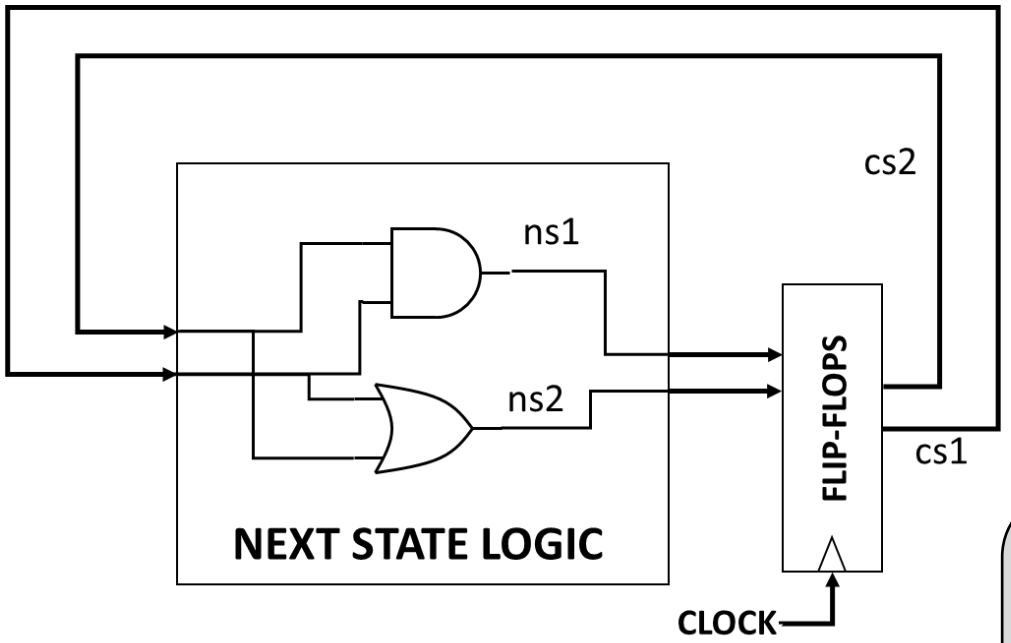
```
AT_0:  begin
              if (in) begin
                 out = 1;
                 next_state = AT_1;
              end
           end
```

Simulation Output




```
VSIM 68> run
#                    0 in=0, clock=0, reset=0, out=x
#                   20 in=0, clock=1, reset=0, out=x
#                   40 in=0, clock=0, reset=0, out=x
#                   50 in=0, clock=0, reset=1, out=0
#                   60 in=0, clock=1, reset=1, out=0
#                   80 in=0, clock=0, reset=1, out=0
#                  100 in=0, clock=1, reset=1, out=0
#                  120 in=0, clock=0, reset=0, out=0
#                  140 in=0, clock=1, reset=0, out=0
#                  160 in=0, clock=0, reset=0, out=0
#                  180 in=0, clock=1, reset=0, out=0
#                  200 in=1, clock=0, reset=0, out=1
#                  220 in=1, clock=1, reset=0, out=0
#                  240 in=1, clock=0, reset=0, out=0
#                  260 in=1, clock=1, reset=0, out=0
#                  280 in=1, clock=0, reset=0, out=0
```

Blocking Vs. Non-blocking Assignments



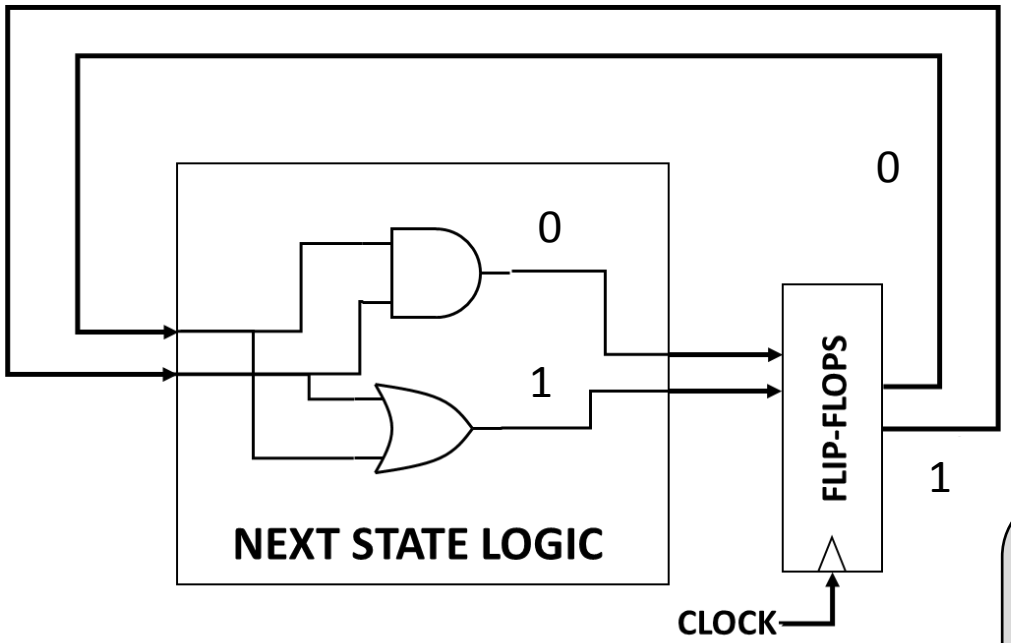

```
// combinational part
always @(cs1, cs2) begin
    ns1 = cs & cs2;
    ns2 = cs1 | cs2;
end
//sequential part
always @ (posedge clock) begin
    cs1 <= ns1;
    cs2 <= ns2;
end
```

OR

```
// sequential + combinational part
always @(posedge clcok) begin
    cs1 <= cs & cs2;
    cs2 <= cs1 | cs2;
end
```

Both LHSs evaluated first; then assigned to RHSs. As if the two ops happened in parallel

Blocking Vs. Non-blocking Assignments




```
// combinational and sequential part
always @(posedge clock) begin
    cs1 = cs & cs2;
    cs2 = cs1 | cs2;
end
// Blocking assignments operate in sequence.
// So first cs1 will become 0 (1&0).
// Now both cs1 and cs2 are 0.
// The next statement executes, and cs2 remains
zero.
```

DOES NOT WORK

OR

```
// sequential + combinational part
always @(posedge clcok) begin
    cs1 <= cs & cs2;
    cs2 <= cs1 | cs2;
end
```

Will operate correctly regardless of order in which these statement are written!!

Lecture 2: Modeling Complex Designs

EL GY 6463: Advanced Hardware Design

Instructor: Siddharth Garg

Procedural Modeling of Combinational Functions

- Expressing control flow
  - Ternary operator
  - If-else construct
  - Case construct

```
module mux( input a,
            input b,
            input sel,
            output c);

assign c = (sel)? a : b;

endmodule
```

condition

If true

If false

```
module mux( input a,
            input b,
            input sel,
            output reg c);

always @ (*) begin

  if (sel==1)
      c = 1;
  else
      c = 0;

end
endmodule
```

Logical equality (more later)

```
module mux( input a,
            input b,
            input sel,
            output reg c);


always @ (*) begin
  case (sel)
    1'b0: c = a;
    1'b1: c = b;
  endcase
end
endmodule
```

All cases must be specified (use default case)

For Loops

- Similar to for loops in C/C++. Can only be used inside timed blocks (always, initial)

Array of 8 bits. MSB = 7, LSB = 0

```
module xor8( input [7:0] a,
             input [7:0] b,
             output reg [7:0] out);


reg [7:0] i;

always @(*) begin
  for (i=0; i<8; i=i+1) begin
      out[i] = a[i] ^ b[i];
  end
end
endmodule
```

Loop counter

From i=0 to 7 in steps of 1

```
module xor8( input [7:0] a,
             input [7:0] b,
             output [7:0] out);

assign out = a ^ b;

endmodule
```

Bit-wise xor

For Loops: a more complex example

- Correlator: out = $\sum_{i=1}^{N} a[i]b[i] + (1 - a[i])(1 - b[i])$
  - Counts number of occurrences where a[i] == b[i]

```
module corr8( input [7:0] a, input [7:0] b, output reg [3:0] out);


reg [7:0] i;

always @(*) begin
  out = 0;
  for (i=0; i<8; i=i+1) begin
      out = out + (a[i] ~^ b[i]);
  end
end
endmodule
```

Out can be at most 8.

Xnor operator

Arithmetic addition

A Note on Verilog Operators

Don't synthesize. Avoid unless using in test bench or for simulation only.

- Arithmetic operators
  - Add (+), subtract (-), multiply (*), divide (/), modulus (%)
  - reg and wire datatypes are interpreted as unsigned numbers

Examples:

```
module add(
      input [3:0] a,
      input [3:0] b,
      output [4:0] out);

assign out = a + b;
//a=4'b0101, b=4'b0001
// => c=00110 (5+1=6)
//a=4'b1000, b=4'b1000
// => c= 10000 (8+8=16)
endmodule
```

Carry out bit

```
module sub(
      input [3:0] a,
      input [3:0] b,
      output [4:0] out);

assign out = a - b;
//a=4'b0101, b=4'b0001
// => c=00100 (5-1=4)
//a=4'b0001, b=4'b0010
// => c= 11111 (1-2=?)
endmodule
```

What's happening here?

```
module mult(
      input [3:0] a,
      input [3:0] b,
      output [7:0] out);

assign out = a * b;
//a=4'b1111, b=4'b1111
// => c=11100001
// (15*15=225)
endmodule
```

Verilog Operators (Bit-wise Vs. Logical)

- Example: `&' (bit-wise) Vs. `&&' (logical)

Two n-bit operands results in an n-bit output

```
module bitwise_and(
      input [3:0] a,
      input [3:0] b,
      output [3:0] out);

assign out = a & b;
//a=4'b0101, b=4'b0001
// => out =0001
//a=4'b1001, b=4'b1011
// => c= 1001
endmodule
```

| (or)
^(xor)
~^(xnor)
~ (negation)

Always results in a **single bit** output. Output is 1 if both operands are non-zero.

```
module logical_and(
      input [3:0] a,
      input [3:0] b,
      output out);

assign out = a && b;
//a=4'b0101, b=4'b0001
// => out = 1
//a=4'b0001, b=4'b0000
// => out = 0
endmodule
```

|| (or)
== (equality)
!= (inequality)
! (logical not)
>=, <=, <, >

Typical Use
If (a || b)
If (a==b)
If (a != b)
If (!a)
If (a > b)

For more details:
1. Appendix C of T&M
2. http://web.engr.oregonstate.edu/~traylor/ece474/lecture_verilog/beamer/verilog_operators.pdf

Verilog Operators (Reduction and Concatenation)

- Compute the parity of an 8 bit string. (Why is this useful)?

```
module parity(input [7:0] str,
              output reg par,
              output reg [8:0] par_str);

reg [2:0] i;
always @(*) begin
  par = str[0];
  for (i=1; i<8; i=i+1) begin
      par = par ^ str[i];
  end
  par_str = {str, par};
end

endmodule
```

Concatenation

```
module parity(input [7:0] str,
              output par);

assign par = ^str;
assign par_str = {str, par};

endmodule
```

Computes (str[0]^str[1]...str[7])

| (or)
~| (nor)
& (and)
~& (nand)
~^ (xnor)

Homework: Read up on Verilog shift operator (>>, <<).

In-class Exercise

- Write a combinational logic module that takes a 16-bit value as input and counts the number of times the pattern ‘0101’ occurs in it.
  - Tip: x[base+:width] gives x[base+width-1:base]

How big should count be?

```
module pattern_count (input [15:0] str, output reg [4:0] count);

reg [4:0] i;

always @ (*) begin
  count = 0;
  for (i=0; i<16; i=i+1) begin
     if (str[i+:4] == 4'b0101)
       count = count + 1;
  end
end
endmodule
```

Parameterized Modules

- Parameters allow you to declare constants that can be *externally modified*
  - Unlike local parameters (localparam) that cannot be modified from outside the module

```
module parity #(parameter W = 8)
               (input [W-1:0] str,
                output reg par,
                output reg [W:0] par_str);
reg [2:0] i;

always @(*) begin
  par = str[0];
  for (i=1; i<W; i=i+1) begin
      par = par ^ str[i];
  end
  par_str = {str, par};
end

endmodule
```

Default value of W = 8

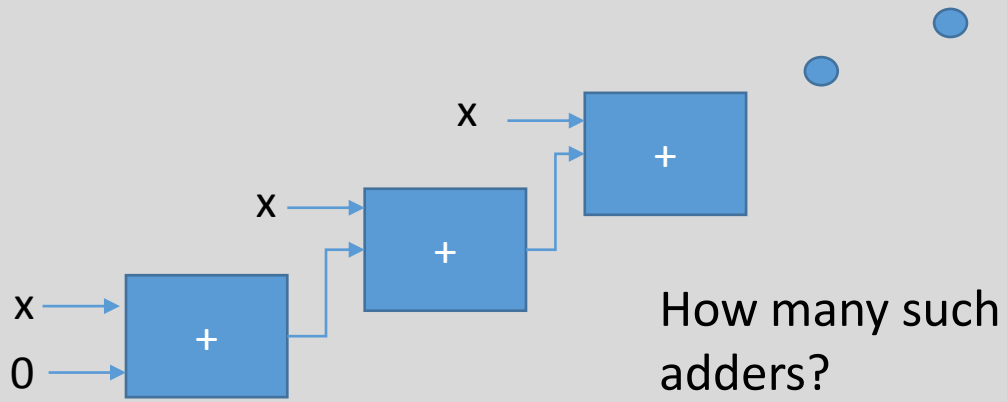
```
parity #(8) p1 (str, par, par_str);

parity #(12) p1 (str, par, par_str);
```

Finite State Machines (Data path and Control path Synthesis)

- Say you want to build a multiplier but only have access to adders and comparators.
  - Solution: repeated addition. Simple Verilog combinational description

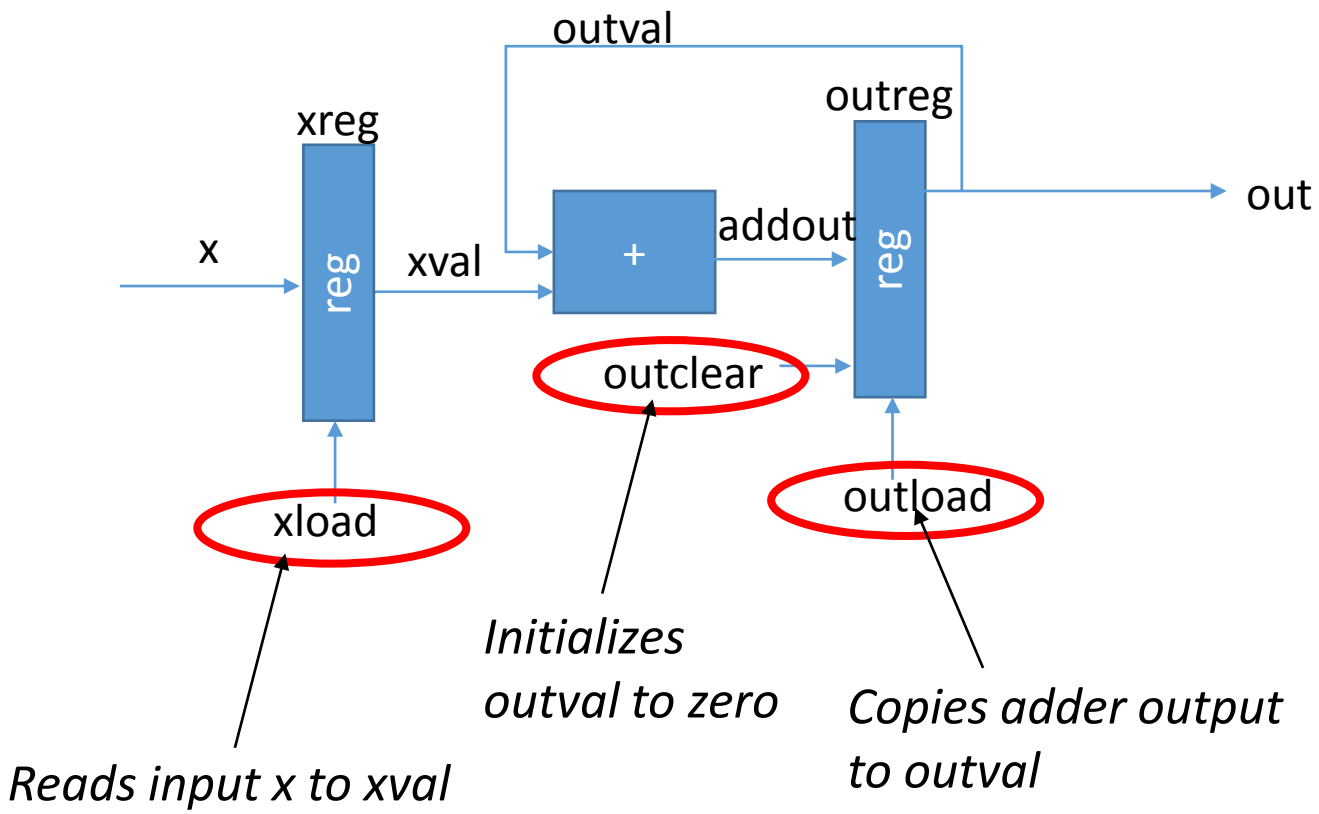
```
module mult #(parameter W = 8)
             (input [W-1:0] x, input [W-1:0] y, output reg [2*W-1:0] out);

reg [W-1:0] i;

always @(*) begin
   out = 0;
   for (i=0; i<y; i=i+1) begin
      out = out + x;
   end
end
endmodule
```

Will work in simulation but does not synthesize?



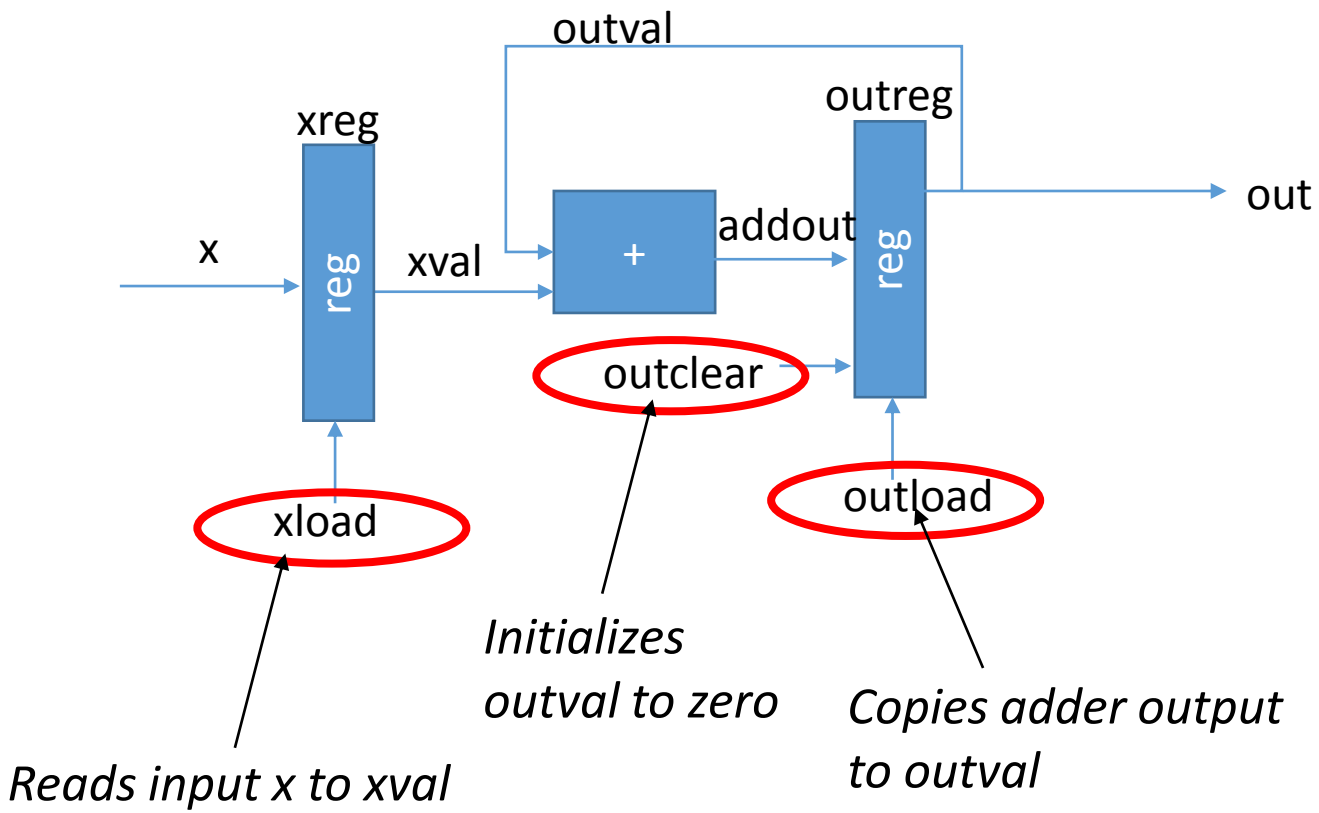
How many such adders?

Better Solution: Use One Adder Repeatedly



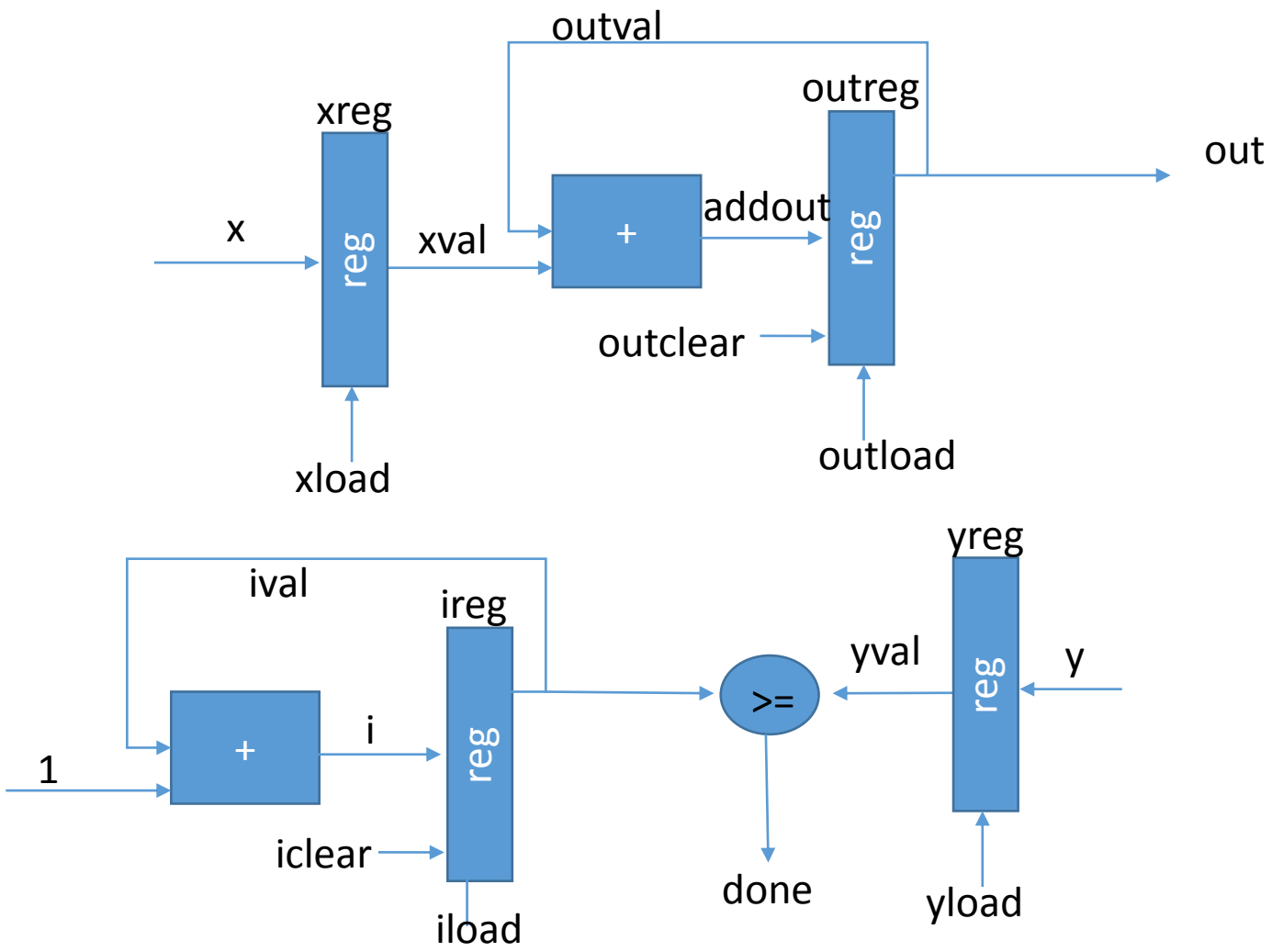

```
module myreg
# (parameter  W=8)
(input [W-1:0] xin,
 input xload,
 input xclear,
 input clk,
 output reg [W-1:0] xout);

always @ (posedge clk) begin
  if(xclear)
        xout = 0;
  else if (xload)
        xout = xin;
end

endmodule
```

Better Solution: Use One Adder Repeatedly




```
module adder
# (parameter W = 8)
(input [W-1:0] in1,
 input [2*W-1:0] in2,
 output reg  [2*W-1:0] out);


//the first operand is 0-extended
always @ (*) begin
   out = in1 + in2;
end

endmodule
```

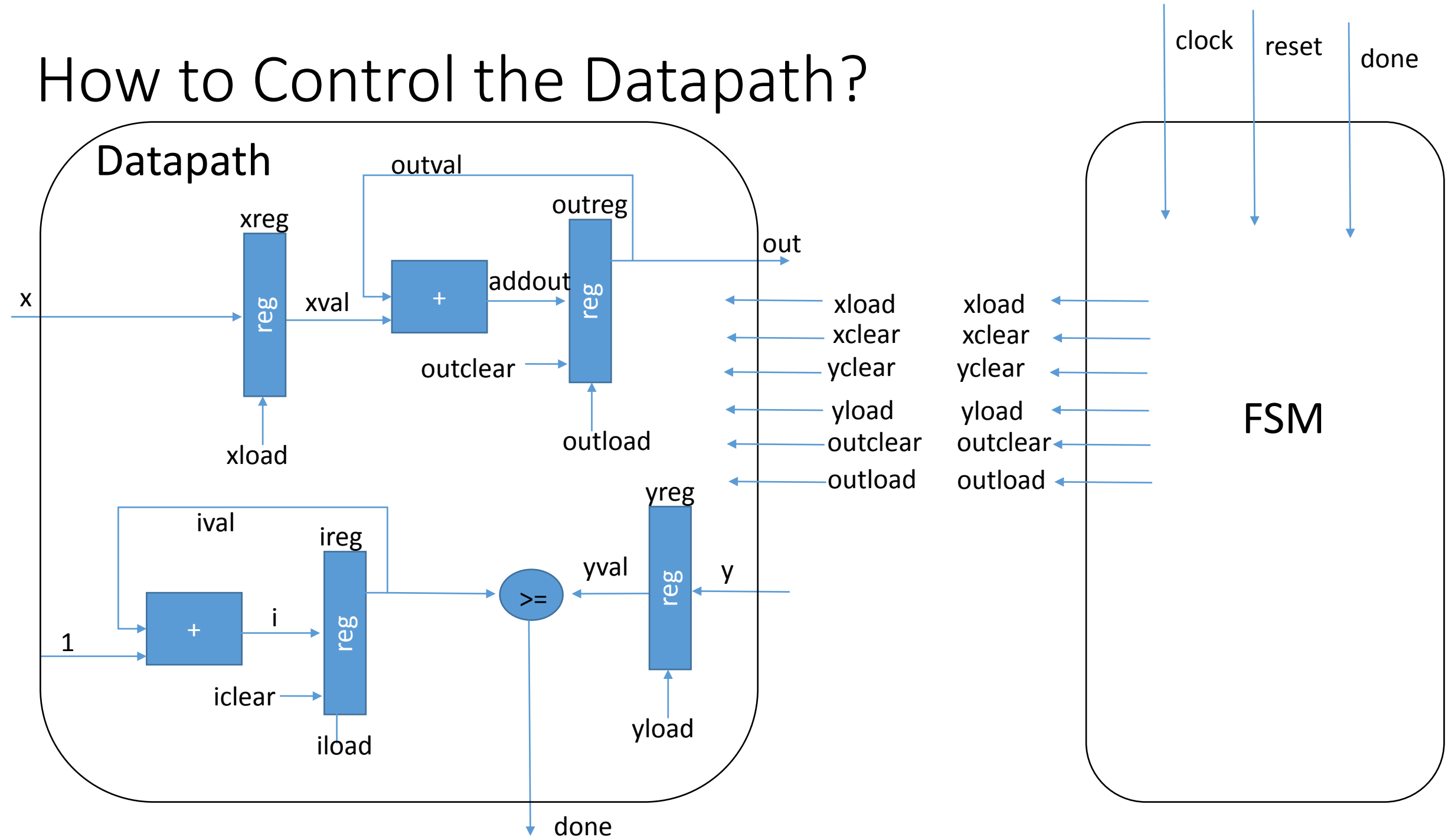
Better Solution: Use One Adder Repeatedly



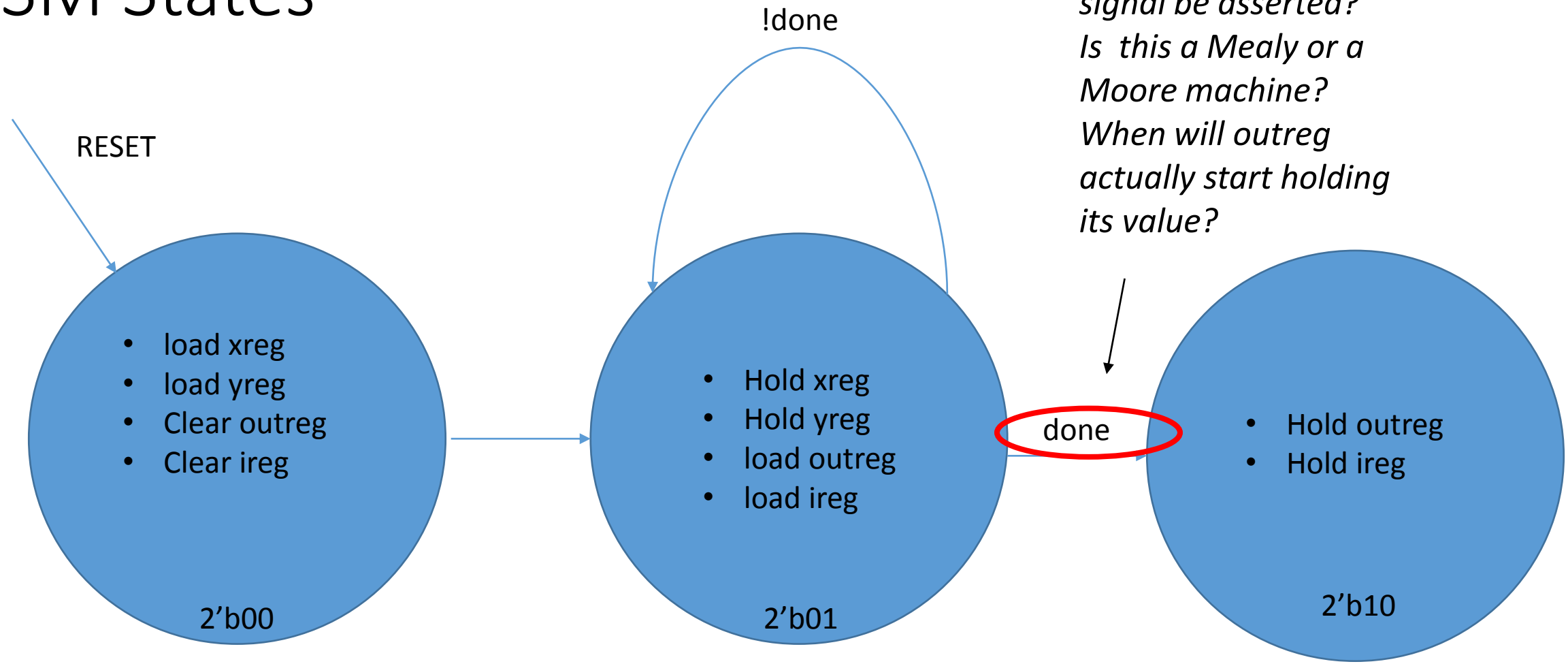

```
module increment
# (parameter W = 8)
(input [W-1:0] in1,
output reg [W-1:0] out);


always @ (*) begin
   out = in1 + 1;
end
endmodule
```

```
module comparator
# (parameter W = 8)
(input [W-1:0] ival,
 input [W-1:0] yval,
 output reg done);

always @ (*) begin
  done = 0;
  if (ival   >= yval ) //Yes?
    done=1;
end
endmodule
```

How to Control the Datapath?



FSM States



Question: for what value of i should done signal be asserted?
Is this a Mealy or a Moore machine?
When will outreg actually start holding its value?

Increment Module

```
module comparator
# (parameter W = 8)
(input [W-1:0] ival,
 input [W-1:0] yval,
 output reg done);

always @ (*) begin
  done = 0;
  if (ival +1   >= yval -1 )
    done=1;
end
endmodule
```

Note:

- ival starts from 0 instead of 1
- When done signal is asserted, outreg only stops loading in the next clock cycle

Putting it All Together

```
module multiplier
# (parameter W = 8)
(input [W-1:0] x, input [W-1:0] y, input clk, input reset, output [2*W-1:0] out);

//variable declarations. Question: what datatype for xval, xclear, xload?

assign  out = addout;

myreg #(W) xreg(x, xload, xclear, clk, xval); //xreg
myreg #(W) yreg(y, yload, yclear, clk, yval); //yreg
myreg #(2*W) outreg(addout, outload, outclear, clk, outval); //outreg
myreg #(W) ireg(i, iload, iclear, clk, ival); //ireg

adder #(W) add1 (xval, outval, addout); //out = out+x
increment inc1 (ival, i);  //i = i+1
comparator #(W) comp1 (ival, yval, done); //i+1 >= y-1

myfsm fsm1 ( clk, reset, done, xload, xclear, yload, yclear, iload, iclear,
outload, outclear); //controller

endmodule
```

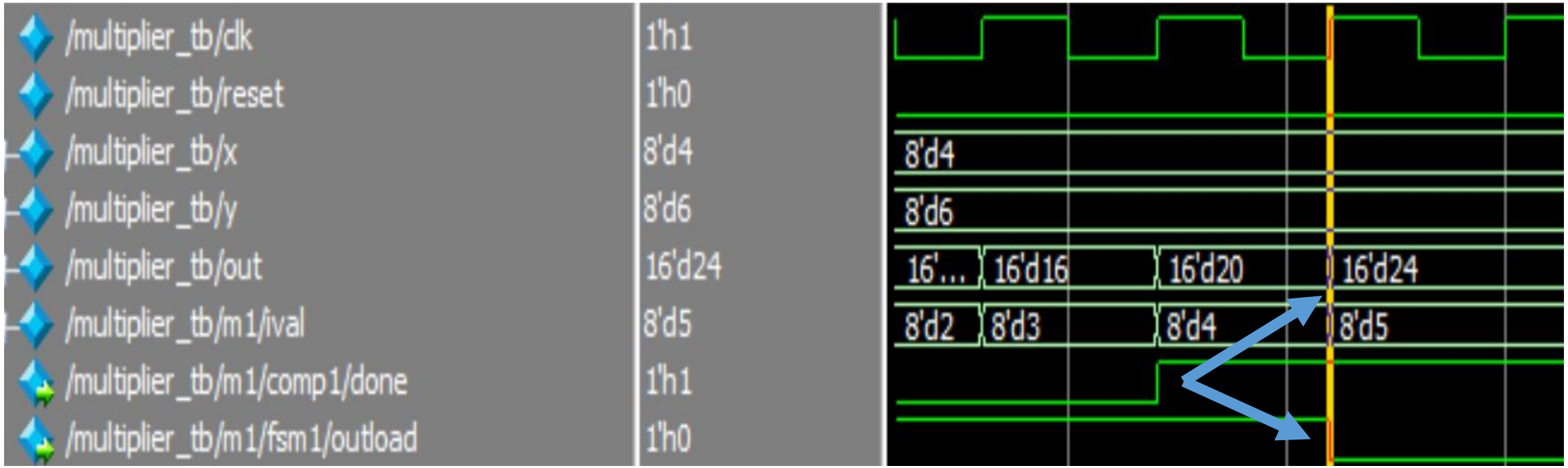
FSM Design (In-Class Exercise)

```
module myfsm (input clk, input reset, input done, output reg xload, output….);

reg [2:0] cs, ns;

always @ (posedge clk, posedge reset) begin
   //SEQUENTIAL PORTION: YOUR CODE HERE
end

always @(*) begin
  ns = cs;
  xload = 0; yload = 0;  iload =  0;  outload = 0;
  xclear = 0; yclear = 0; iclear = 0; outclear = 0;
  case (cs)
     2'b00 :  begin
     // YOUR CODE HERE
     end
     2'b01: begin
     //YOUR CODE HERE
     end
     2'b10 : begin
     // YOUR CODE HERE
     end
     default :
  endcase
end
endmodule
```

Solution

```
case (cs)
     //START
     2'b00 :  begin
         xload = 1; yload = 1;  iclear = 1; outclear = 1;
         ns = 2'b01;
     end
     //working
     2'b01: begin
         xload = 0; yload = 0; iload = 1; outload = 1;
         if (done)
             ns = 2'b10;
     end
     //finished
     2'b10 : begin
         iload = 0; outload = 0;
     end

     default :  // put in don't cares
```

Desired Simulation Output



One clock cycle difference. Think of done as “almost done”