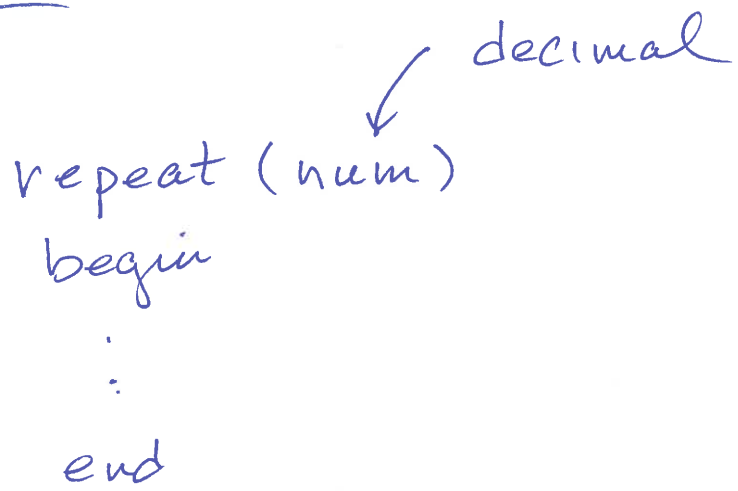
Example 7-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;

initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end


//Illustration 2: Find the first bit with a value 1 in flag (vector
variable)
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;

  while((i < 16) && continue ) //Multiple conditions using operators.
  begin
    if (flag[i])
    begin
      $display("Encountered a TRUE bit at element number %d", i);
      continue = 'FALSE;
    end
    i = i + 1;
  end
end
```

Example 7-23 For Loop

```
integer count;

initial
    for ( count=0; count < 128; count = count + 1)
            $display("Count = %d", count);
```

Repeat Loop



Example 7-24 Repeat Loop

```
//Illustration 1 : increment and display count from 0 to 127
integer count;

initial
begin
   count = 0;
   repeat(128)
   begin
        $display("Count = %d", count);
        count = count + 1;
   end
end
```

forever loop

from before,

```
initial
  clock = 1'b0;

always
  #5 clock = ~clock; // clock with a
                        10 ns period
```

```
initial
  begin
    clock = 1'b0;
      forever #5 clock = ~clock;
  end
```

Synchronizing 2 reg values

```
reg clock;
reg A, B;

initial
    forever @(posedge clock) A = B;
```

So far we've only discussed sequential blocks

Defn (parallel blocks)

a block where all statements execute concurrently (subject to any # delays)

Notes:

1) statements execute concurrently even though blocking assignments are used

2) delays are with respect to the time the parallel block was entered

3) "fork" and "join" are Keywords

sequential blocks

```
initial
  begin
    x = 1'b0;
    y = 1'b1;
    z = {x, y}; // z = 01
    w = {y, x}; // w = 10
  end

. . . . . .

initial
  begin
    x = 1'b0;      // t = 0
    #5 y = 1'b1;   // t = 5
    #10 z = {x, y}; // t = 15
    #20 w = {y, x}; // t = 35
  end
```

parallel blocks

```
initial
  fork
    x = 1'b0;   // t = 0
    #5 y = 1'b1;   // t = 5
    #10 z = {x, y}; // t = 10
    #20 w = {y, x}; // t = 20
  join
```

Example 7-35 *Behavioral 4-to-1 Multiplexer*

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//output declared as register
reg out;

//recompute the signal out if any input signal changes.
//All input signals that cause a recomputation of out to
//occur must go into the always @(...)  sensitivity list.
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
  2'b00: out = i0;
  2'b01: out = i1;
  2'b10: out = i2;
  2'b11: out = i3;
  default: out = 1'bx;
  endcase
end


endmodule
```

Example 7-36 *Behavioral 4-bit Counter Description*

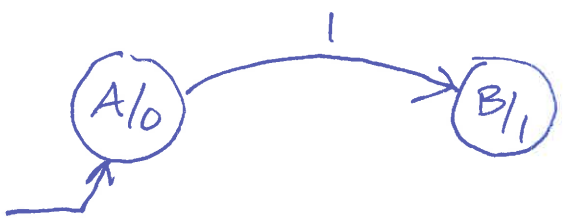
```
//4-bit Binary  counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;
//output defined as register
reg [3:0] Q;

always @( posedge clear  or negedge clock)
begin
  if (clear)
     Q <= 4'd0;  //Nonblocking assignments are recommended
                 //for creating sequential logic such as flipflops
  else
     Q <= Q + 1;// Modulo 16 is not necessary because Q is a
                // 4-bit value and wraps around.
end

endmodule
```

Note:
Initial value
for regs is "x"

Finite State Machines

- Mealy
- Moore

State Diagram (MOORE)



Initial state: A

State Diagram (Mealy)

Initial state: M

e.g.

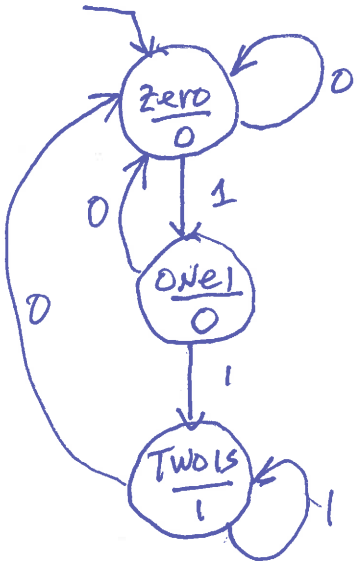
Specification: Change the 1st '1' in a subsequence to '0'

for example

r = 1 0 0 1 1 0 1 1 1 0 1 0

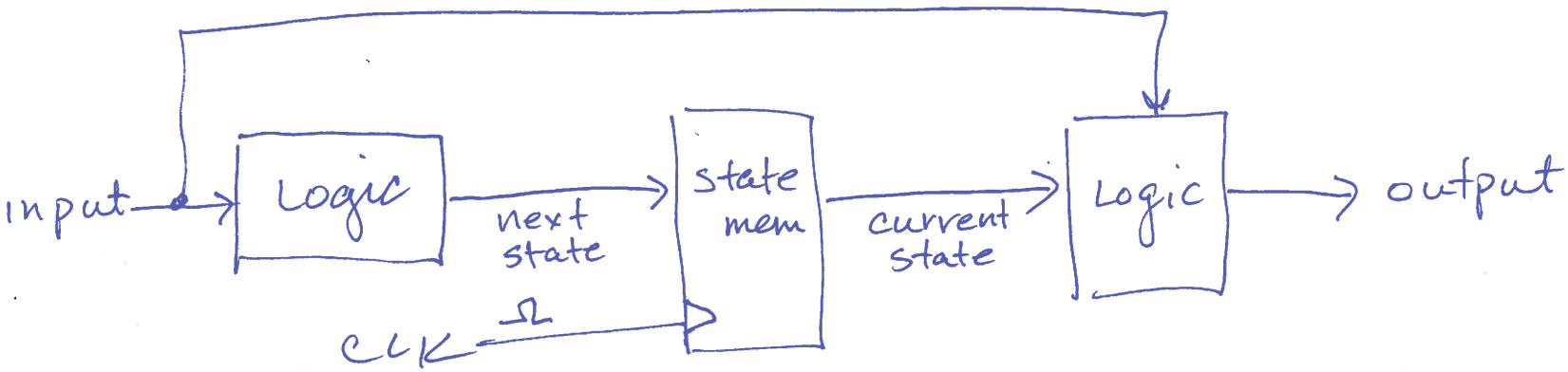
FSM output = 0 0 0 0 1 0 0 1 1 0 0 0

MOORE



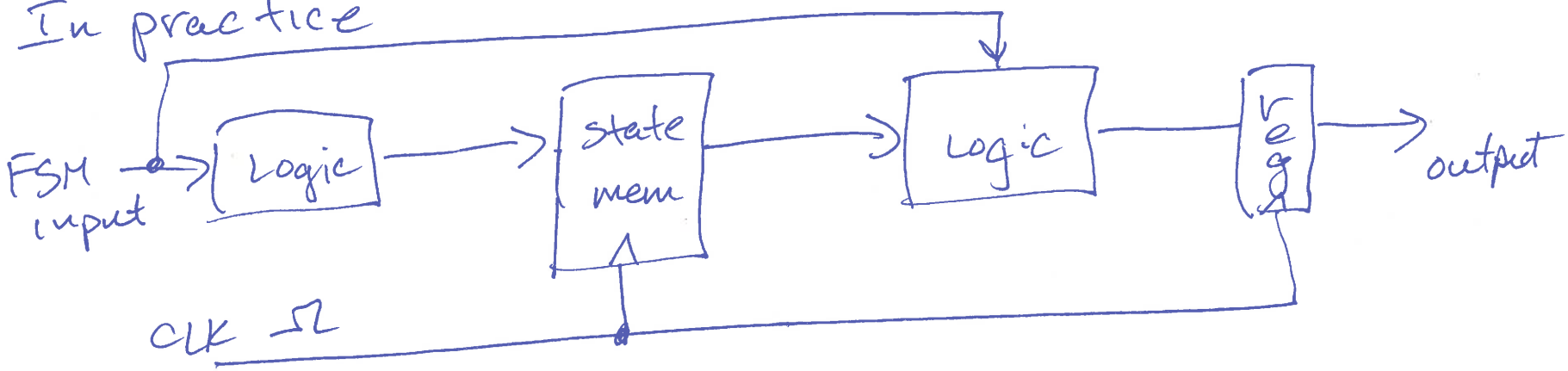

```
Module moore (out, r, clk, reset);
 output out; reg out;
 input r, clk, reset;
 parameter zero=0, one1=1, twoIs=2;
 // state assignments
 reg [1:0] state;  // state registers
 reg [1:0] next_state;

 always @(r OR state)
   case (state)
     zero: begin out=0;
           if (r) next_state = one1;
           else next_state = zero; end

     one1: begin out=0;
           if(!r) next_state = zero;
           else next_state = twoIs; end

     twoIs; begin out=1;
           if (r) next_state = twoIs;
           else next_state = zero; end

   endcase

always @(posedge clk
          or reset)

 if (reset) begin
     state = zero;
      out=0; end
 else begin
     state = next_state;
     out=0;
     end
endmodule
```

Mealy

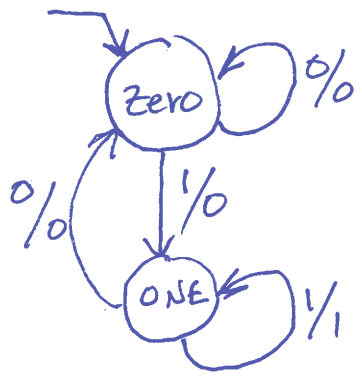


But FSM output only is supposed to change at ~~the~~ the state changes

In practice



Mealy




```
module mealy (clk, r, out, reset);
input clk, r, reset;
output out; reg out;
reg state; // state register
reg next_out; reg next_state;
parameter zero=0; parameter one=1; // state assign.
always @(r or state)
  case (state)
zero: begin
      if (r) next_state = one;
       else next_state = zero;
       next_out = 0; end

one: begin
      if (r) begin
            next_state = one;
            next_out = 1; end

       else begin
            next_state = zero;
            next_out = 0; end

    end
endcase

always @ (posedge clk)
  if (reset) begin
      state = zero;
      out = 0; end

  else begin
      state = next_state;
      out = next_out
      end
endmodule
```

in the moore FSM we did

parameter one1 = 1, two1s = 2, zero = 0;

Suppose we did

parameter one1 = 3'b001, two1s = 3'b100, zero = 3'b010;

Suggest one-hot encoded FSM

Tasks & functions

In C

Subroutines =>

```
//Subr template
float foo(- - - )
{
{

main (    )
  -
  =
  y = foo(- - - )
      :
  w = foo(- - - - )
      :
```

In Verilog we have

Tasks
and
Function

Declared in the module they are used

Table 8-1

Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output**, or **inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value, but can pass multiple values through **output** and **inout** arguments.