Finite State Machines

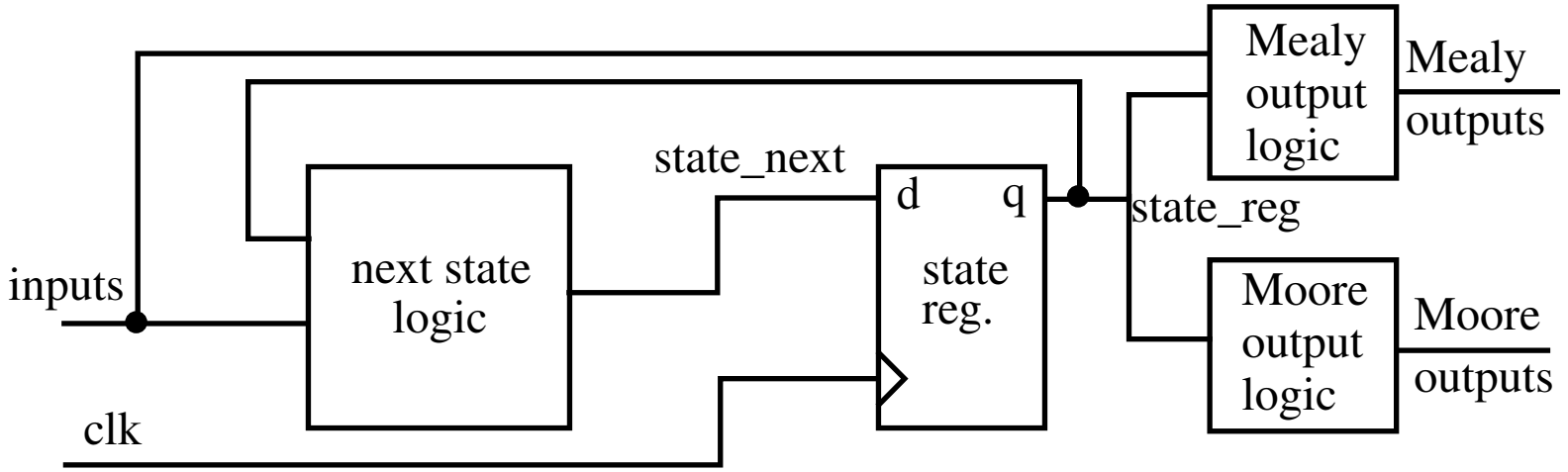
FSMs are sequential machines with "random" next-state logic

Used to implement functions that are realized by carrying out a **sequence of steps** -- commonly used as a controller in a large system

The state transitions within an FSM are more complicated than for regular sequential logic such as a shift register

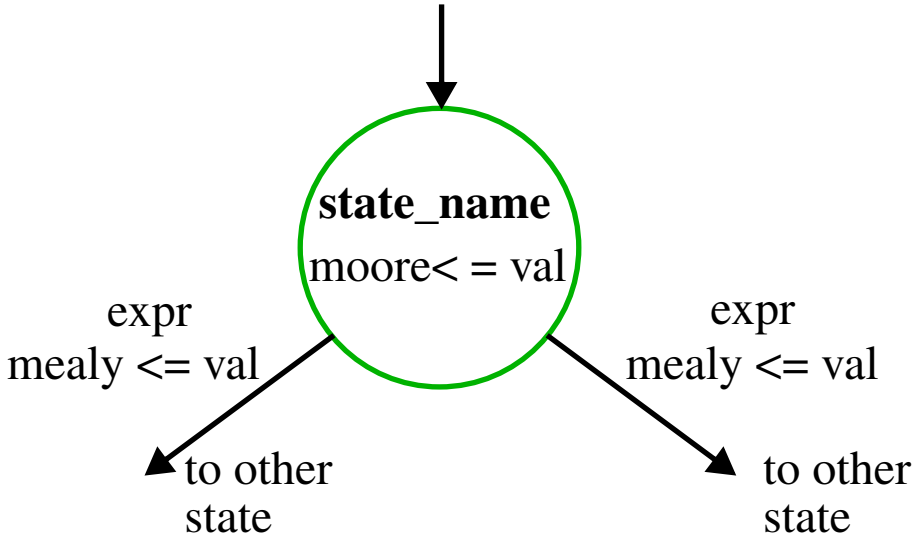
An FSM is specified using five entities: *symbolic states*, *input signals*, *output signals*, *next-state function* and *output function*

• Mealy vs Moore output



Finite State Machines

State diagram



A *node* represents a unique state

An *arc* represents a transition from one state to another
Is labeled with the condition that causes the transition

Moore outputs are shown inside the bubble

Mealy outputs are shown on the arcs

Only asserted outputs are listed

Consider a memory controller that sits between a processor and a memory unit

- Commands include *mem*, *rw* and *burst*

  mem is asserted when a memory access is requested

  rw when '1' indicates a read, when '0' indicates a write

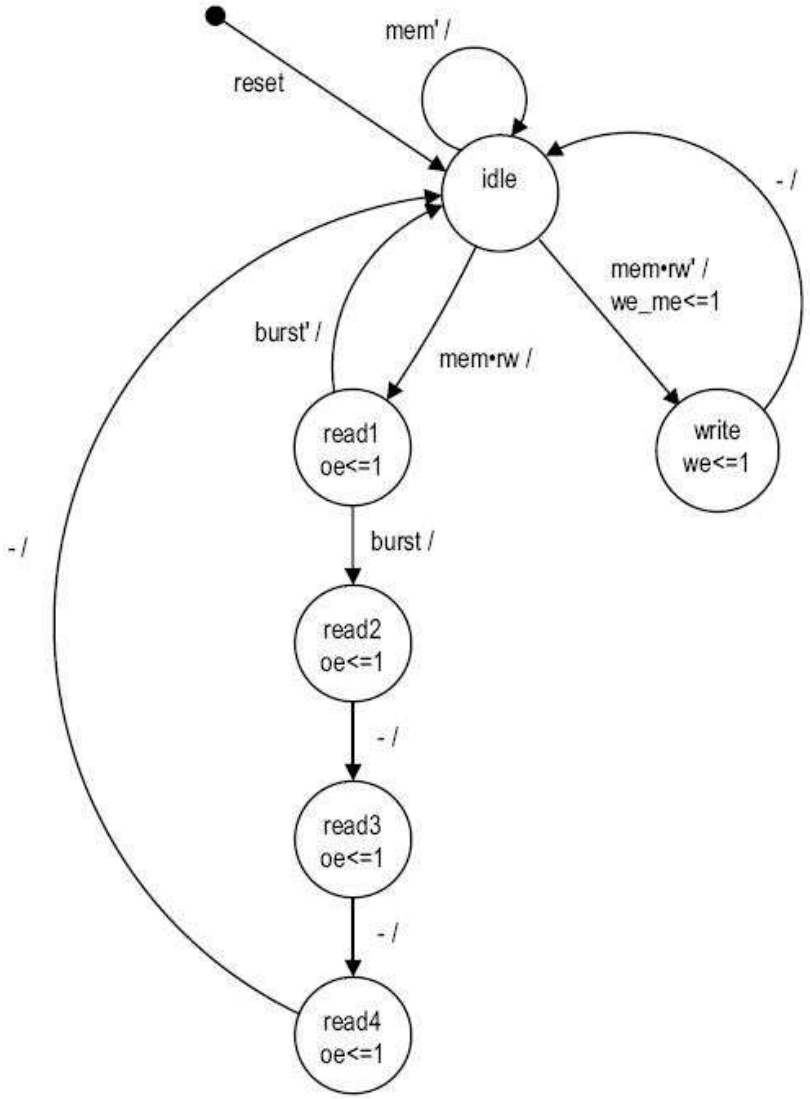
  burst is a special read operation in which 4 consecutive reads occur

- Two control signals *oe* (output enable) and *we* (write enable)

  One Mealy output *we_me*

Finite State Machines

The controller is initially in the *idle* state, waiting for *mem* to be asserted



Once *mem* is asserted, the FSM inspects the *rw* signal and moves to either the *read1* or *write* state on **rising edge of clk**

The logic expressions are given on the arcs

They are checked on the rising edge of the clock

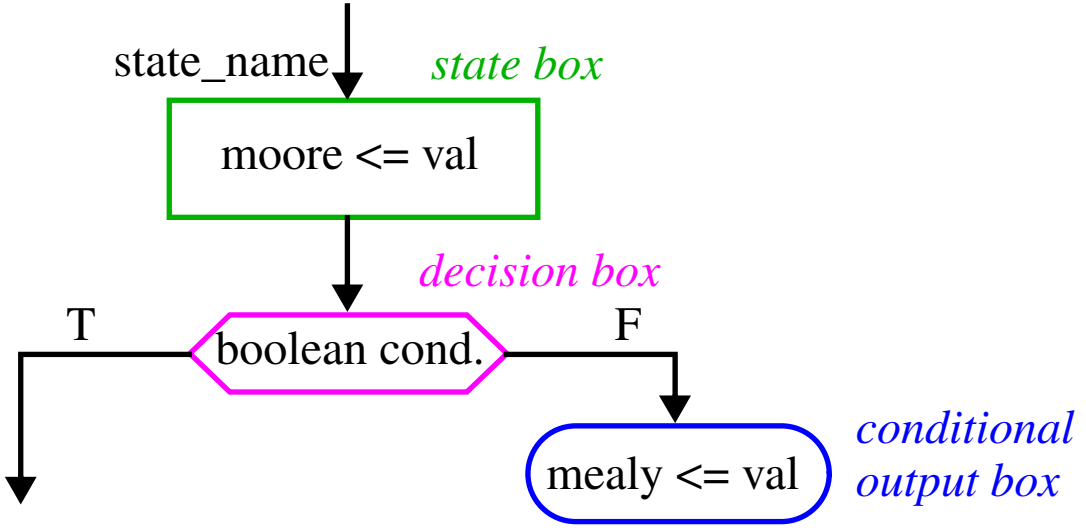
For example, if *mem* is asserted and *rw* is '1', a transition is made to *read1* and the output signal *oe* is asserted

Finite State Machines

Algorithmic State Machine (ASM) chart

Flowchart-like diagram with transitions controlled by the rising edge of clk

More descriptive and better for complex description than state diagrams



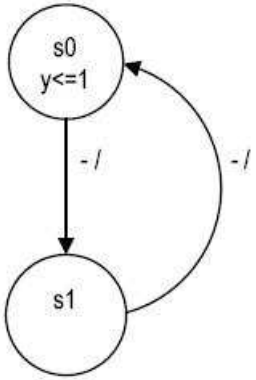
Each *state box* has only one exit and is usually followed by a *decision box*

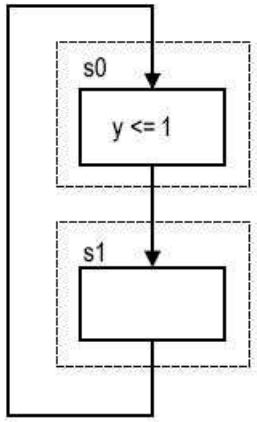
Conditional output boxes can only follow *decision* boxes and list the Mealy outputs that are asserted when we are in this state and the Boolean condition(s) is true

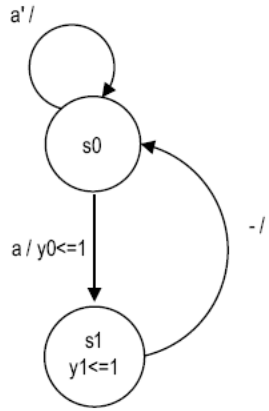
EVERYTHING that follows a state box (to the next state) is **next-state** combo. logic!

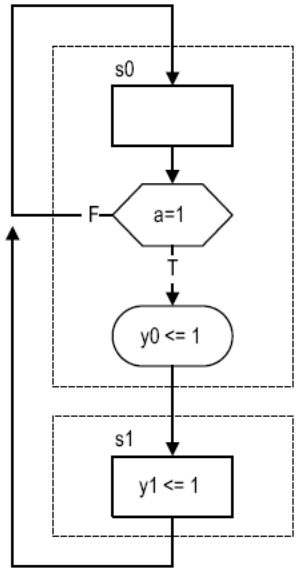
Finite State Machines

Conversion between state diagrams and ASMs









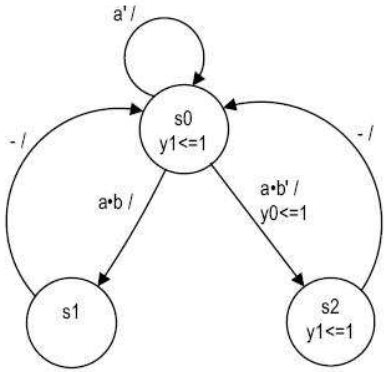
Conversion process is trivial for the left example

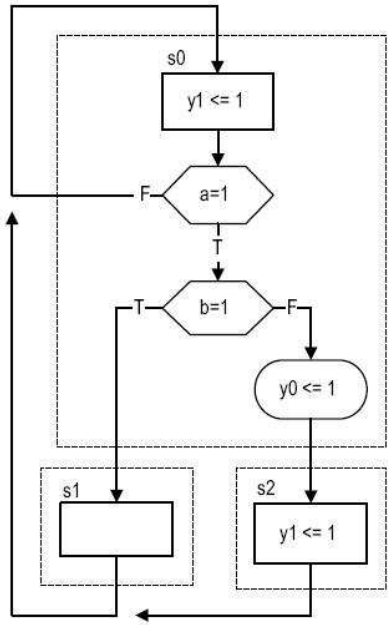
For right example, a decision box is added to accommodate the conditional transition to state *s1* when *a* is true.

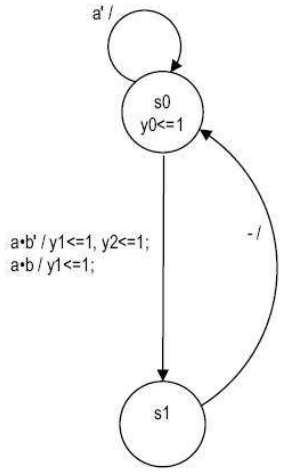
A conditional output box is added to handle the Mealy output that depends on both *state_reg=s0* and *a='1'*

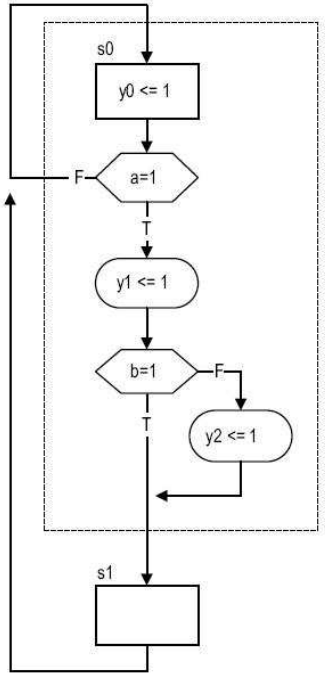
Finite State Machines

More examples







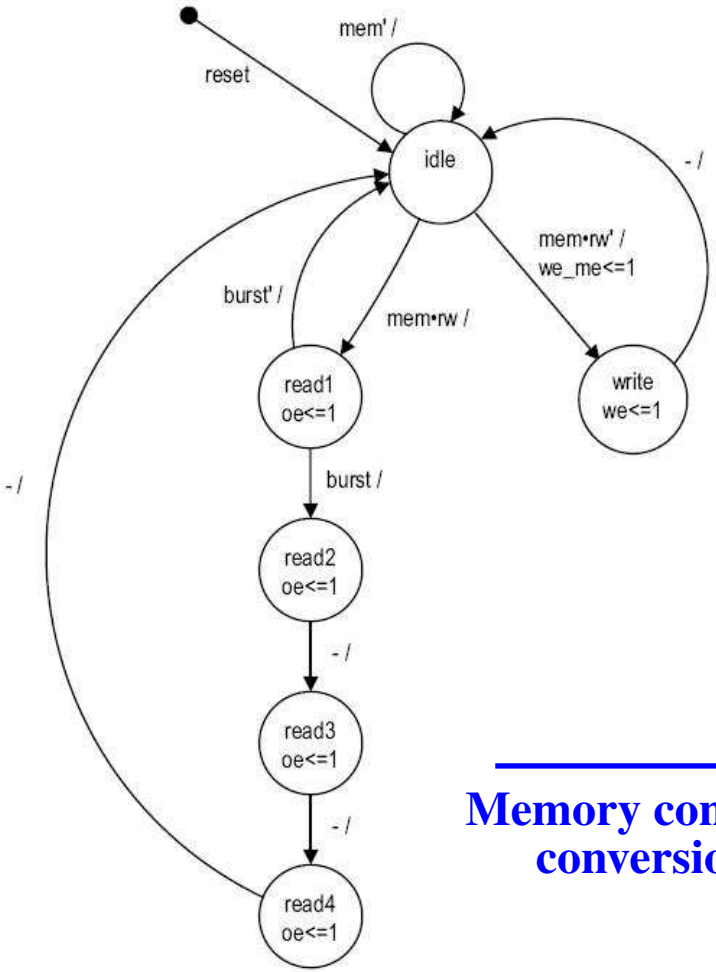


The same general structure is apparent for either state diagrams or ASMs

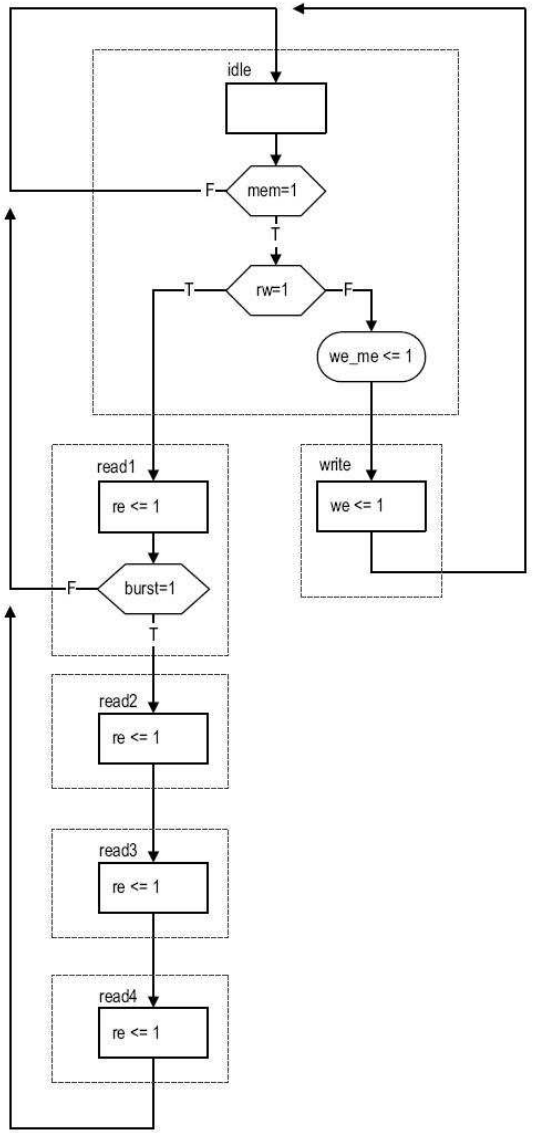
The biggest difference is in how the *decisions* and *conditional outputs* are expressed

When we code this in VHDL, you must view the decision and conditional output logic following a state (up to the next state(s)) as **combinational next-state logic**

Finite State Machines



Memory controller conversion



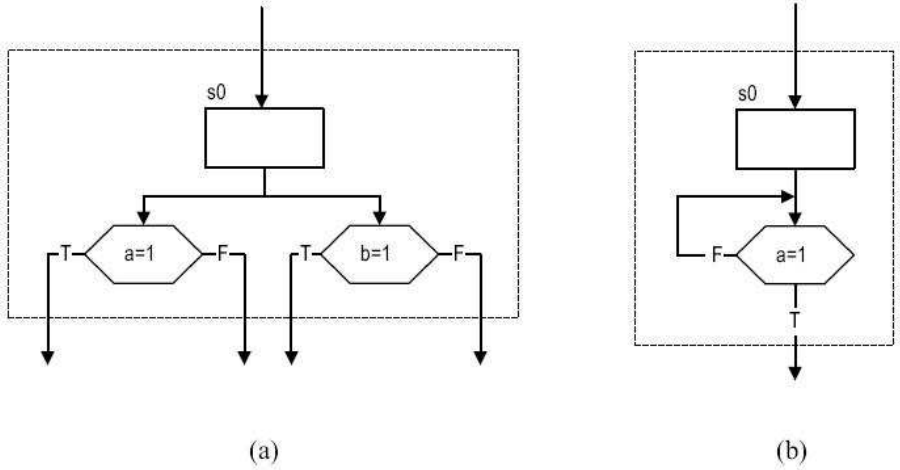
Finite State Machines

Basic rules:

- For a given input combination, there is **one unique exit path** from the current ASM block
- The exit path of an ASM block **must always lead** to a state box.

  The state box can be the state box of the current ASM block or a state box of another ASM block.

Incorrect ASM charts:



(a) (b)

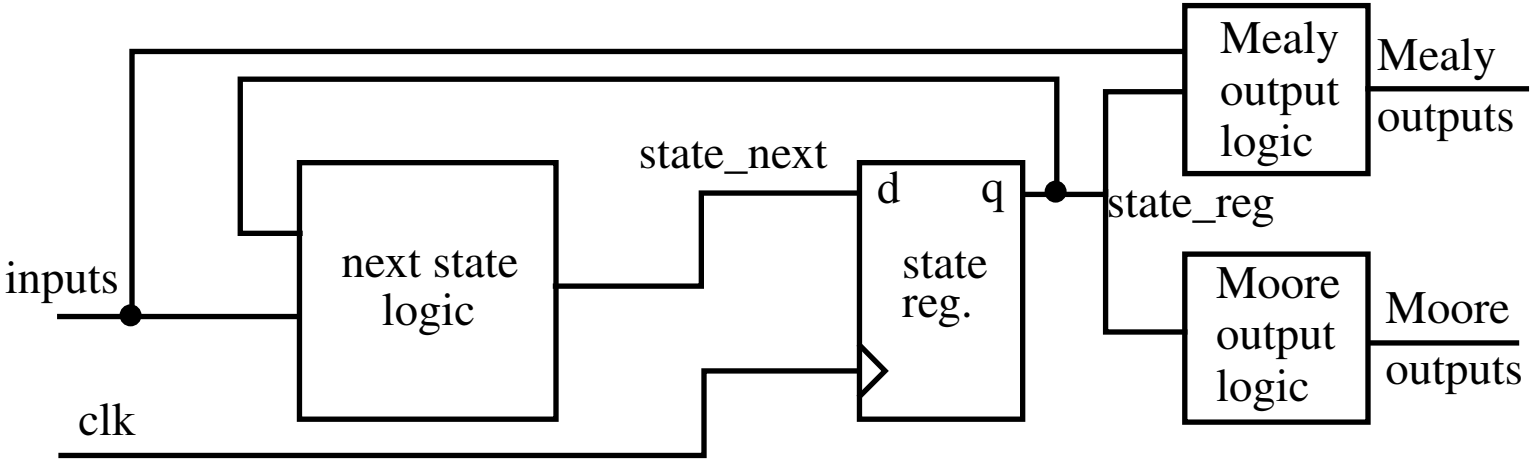
There are two exit paths (on the left) if *a* and *b* are both '1' and NO exit path (on the right) when a is '0'

Finite State Machines

How do we interpret the ASM chart

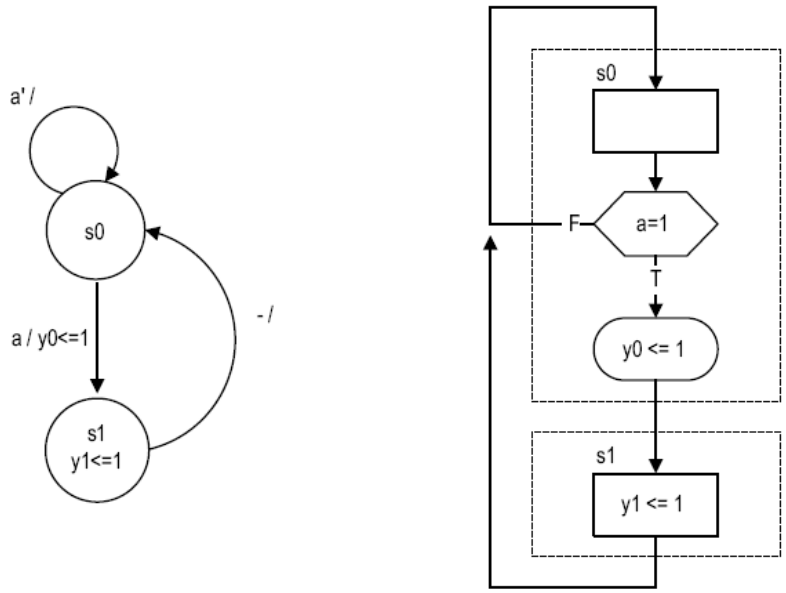
- At the rising edge of clk, the FSM enters a new state (a new ASM block)
- During the clock period, the FSM performs several operations
  - It activates Moore output signals asserted in this new state
  - It evaluates various Boolean expressions of the decision boxes and activates the Mealy output signals accordingly
- At the next rising edge of clk (the end of the current clock period), the results of Boolean expression are examined simultaneously
  - An exit path is determined and the FSM stays or enters a new ASM block

Timing analysis of an FSM (similar to regular sequential circuit)



Timing Analysis of FSMs

Consider a circuit with both a Moore and Mealy output



The timing parameters are

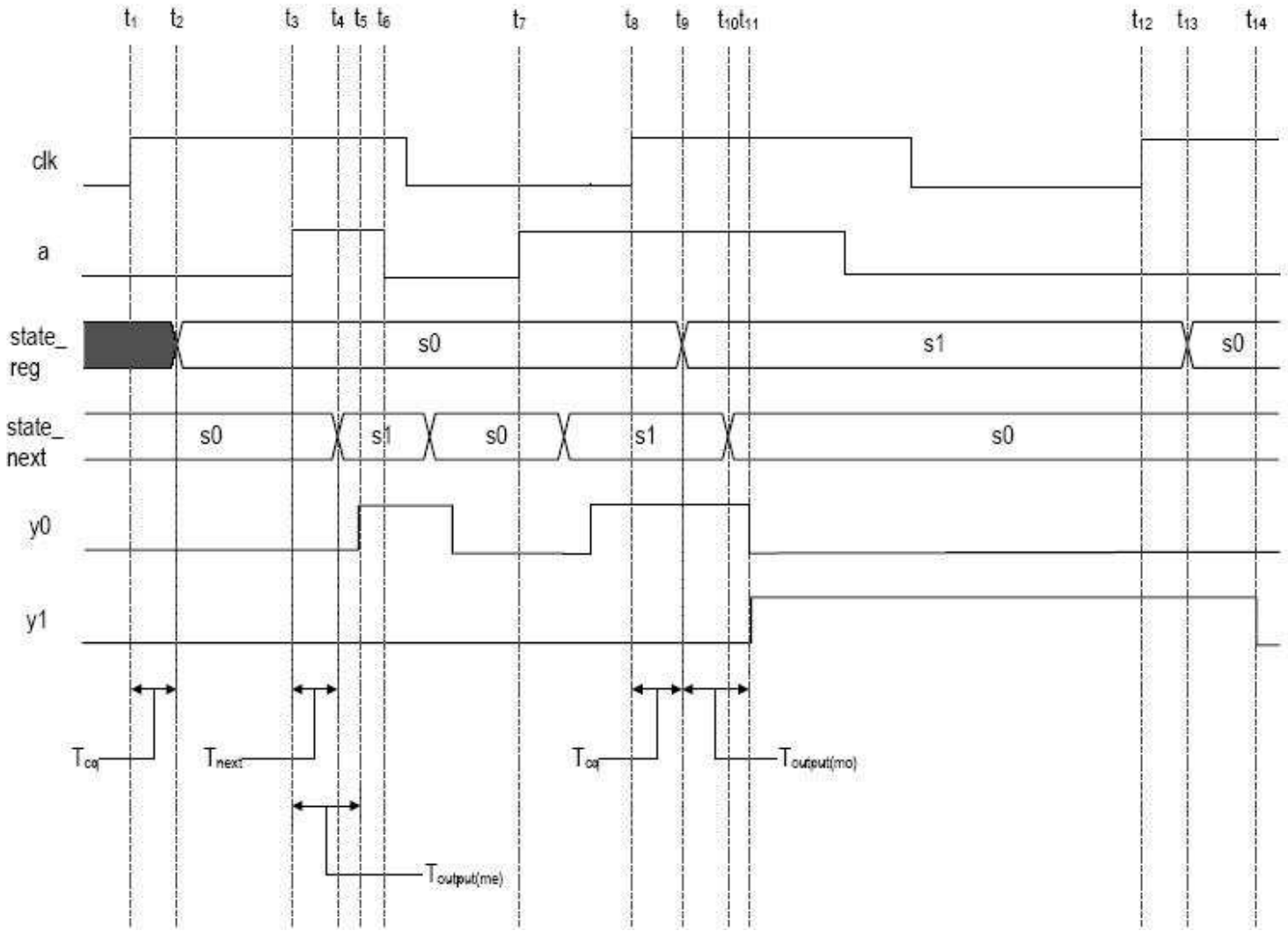
- T_{cq}, T_{setup}, T_{hold}, $T_{next(max)}$
- $T_{output(mo)}$ (Moore logic) and $T_{output(me)}$ (Mealy logic)

Similar to the analysis of a regular sequential circuit, the minimum clock period (max clk freq) of a FSM is given by

$$T_c = T_{cq} + T_{next(max)} + T_{setup}$$

Timing Analysis of FSMs

Sample timing diagram



Timing Analysis of FSMs

Since the FSM is frequently used in a controller application, the delay of the output signals are important

For Moore

$$T_{co(mo)} = T_{cq} + T_{output(mo)}$$

For Mealy (when change is due to a change in state)

$$T_{co(me)} = T_{cq} + T_{output(me)}$$

For Mealy (when change is due to a change in input signal(s))

$$T_{co(me)} = T_{output(me)}$$

Although the difference between a Moore and Mealy output seem subtle, as you can see from the timing diagram, there behaviors can be very different

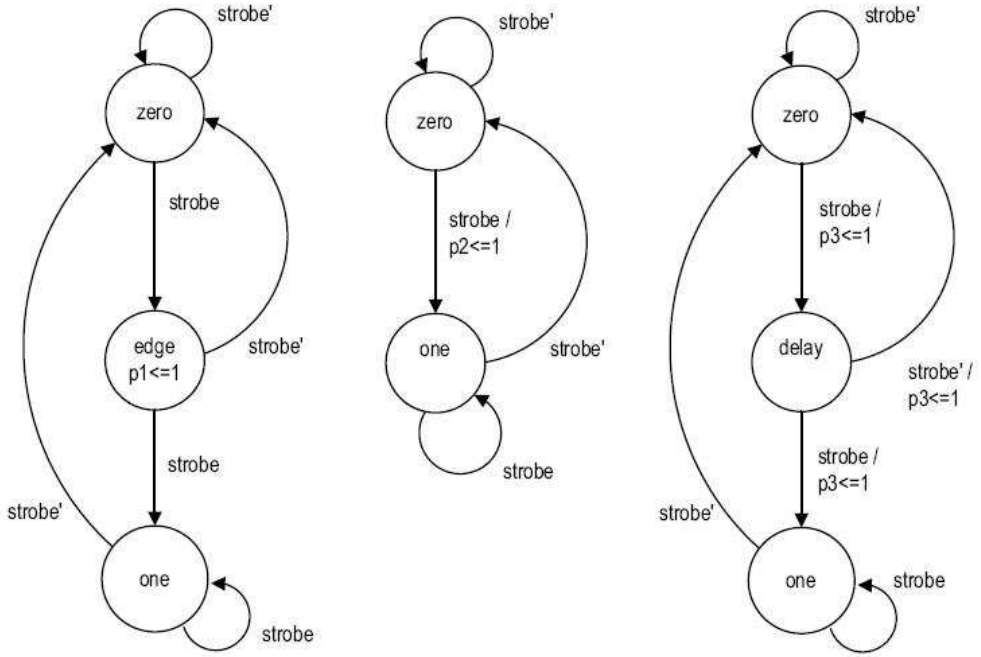
And, in general, it takes fewer states to realize a given function using a Mealy machine (note that both are equivalent in 'power')

But greater care must be exercised

Mealy vs Moore

Consider an **edge detection circuit**

The circuit is designed to detect the rising edge of a slow *strobe* input, i.e., it generates a "short" (1-clock period or less) output pulse

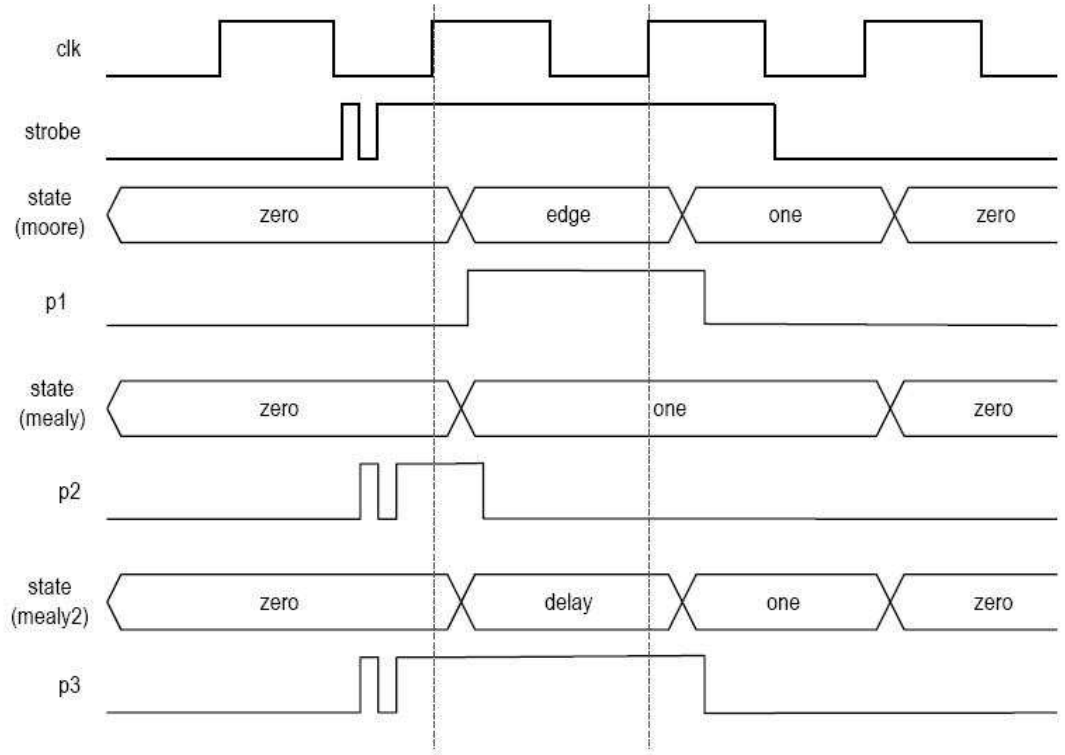


The input signal may be asserted for a long time (think of a pushbutton) -- the FSM has one state for *long duration* '0's and one state for *long duration* '1's

The output, on the other hand, responds only to the rising edge and generates a *pulse* of much shorter duration

Mealy vs Moore

The left-most design above is a Moore implementation, which additionally includes an *edge* state



Middle design is a Mealy machine

The output *p2* goes high in the ***zero*** state when *strobe* becomes '1' (after a small propagation delay), and stays high until the transition to state *one* on the next rising edge

Mealy vs Moore

The right-most design includes both types of outputs and adds a third state *delay*

The state diagram asserts *p3* in the *zero* state (as in second version) when *strobe* goes high and transitions to *delay* state

But since both transitions out of the *delay* state keep *p2* asserted, this has the effect of adding a clock cycle to *p2*'s high state (as in the first version)

Since the assertion is on all outgoing arcs, it is high **independent** of the input conditions (and can be added inside the bubble as a Moore output)

All three designs generate a 'shot pulse' but with subtle differences -- understanding these differences is key to deriving a **correct** and **efficient** FSM

There are **three main differences** between Mealy and Moore:

- Mealy machine uses fewer states -- the input dependency allows several output values to be specified in the same state
- Mealy machine responds faster -- one clock cycle earlier in systems that use output
- Mealy machine may be transparent to glitches, i.e., passing them to the output

Mealy vs Moore

So which one is better?

For control system applications, we can divide control signals into two categories, ***edge sensitive*** and ***level sensitive***

An **edge sensitive** signal (e.g., the enable signal on a counter) is sampled only on the rising edge of clock

Therefore, glitches do NOT matter -- only the setup and hold times must be obeyed

Both Mealy and Moore machines can generate output signals that meet this requirement

However, Mealy machines are preferred because it responds one clk cycle faster and uses fewer states

For a **level sensitive** control signal, the signal must be asserted for a certain interval of time (e.g., the write enable signal of an SRAM chip) and Moore is preferred

While asserted, it MUST remain stable and free of glitches

VHDL Description of FSM

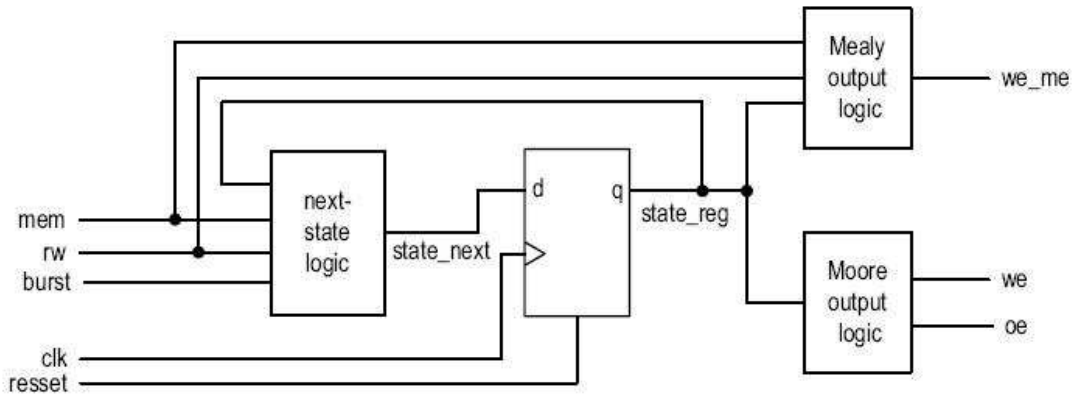
Coding FSMs is similar to regular sequential logic, e.g., separate the memory elements out and derive the next-state/output logic

There are two differences

- Symbolic states are used in an FSM description -- we use the *enumeration* VHDL data type for the state registers
- The next-state logic needs to be constructed according to a state diagram or ASM, as opposed to using regular combinational logic such as a incrementer or shifter

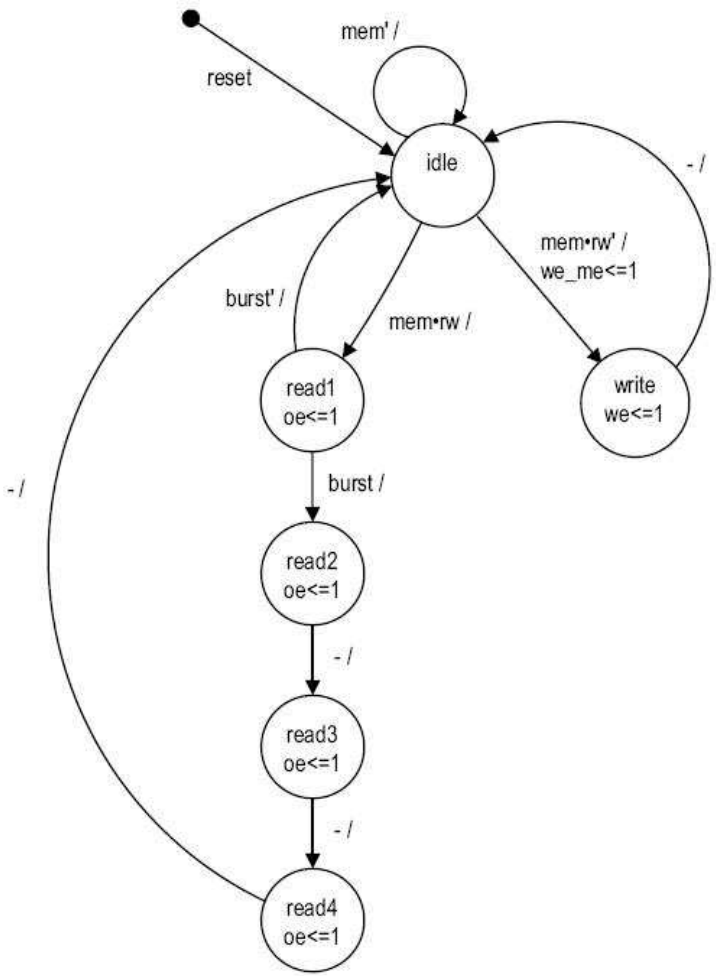
There are several coding styles

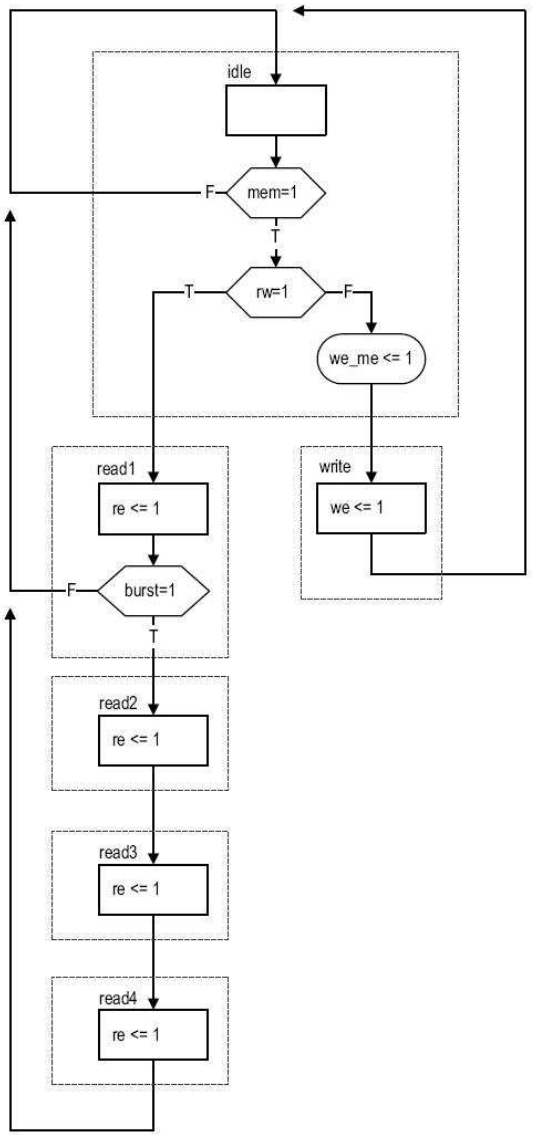
- **Multi-Segment**: Create a VHDL code segment for each block in the block diagram



Memory controller example

VHDL Description of FSM





Multi-Segment VHDL Description of FSM

```
library ieee;
use ieee.std_logic_1164.all;

entity mem_ctrl is
   port(
      clk, reset: in std_logic;
      mem, rw, burst: in std_logic;
      oe, we, we_me: out std_logic
   );
end mem_ctrl ;

architecture mult_seg_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
   begin
```

Multi-Segment VHDL Description of FSM

```
-- state register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;

-- next-state logic
   process(state_reg, mem, rw, burst)
      begin
      case state_reg is
-- When multiple transitions exist out of a state,
-- use an if stmt
         when idle =>
            if (mem = '1') then
```

Multi-Segment VHDL Description of FSM

```
                    if (rw = '1') then
                       state_next <= read1;
                    else
                       state_next <= write;
                    end if;
                 else
                    state_next <= idle;
                 end if;

              when write =>
                 state_next <= idle;

              when read1 =>
                 if (burst = '1') then
                    state_next <= read2;
                 else
                    state_next <= idle;
                 end if;
```

Multi-Segment VHDL Description of FSM

```
            when read2 =>
               state_next <= read3;

            when read3 =>
               state_next <= read4;

            when read4 =>
               state_next <= idle;
         end case;
      end process;

   -- Moore output logic
      process(state_reg)
         begin
         we <= '0'; -- default value
         oe <= '0';  -- default value
```

Multi-Segment VHDL Description of FSM

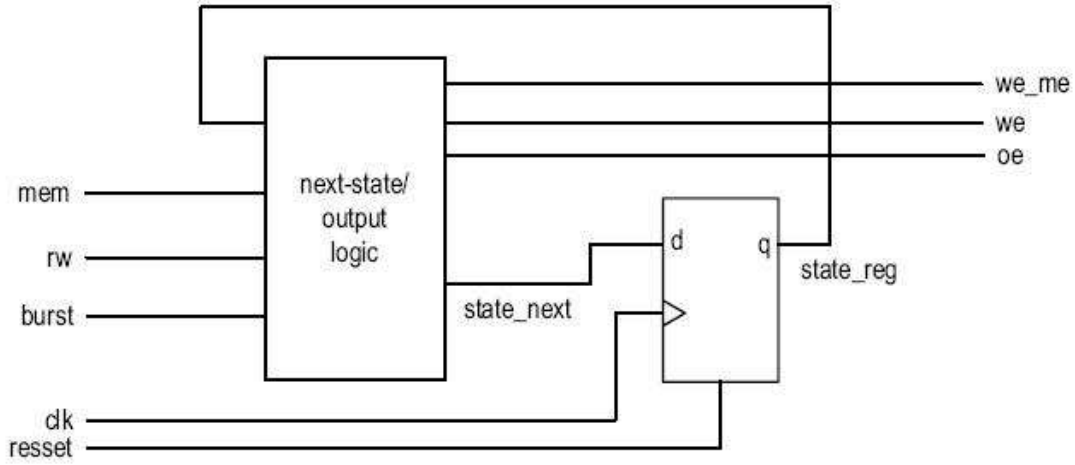
```
case state_reg is
   when idle =>

   when write =>
      we <= '1';

   when read1 =>
      oe <= '1';

   when read2 =>
      oe <= '1';

   when read3 =>
      oe <= '1';

   when read4 =>
      oe <= '1';
end case; end process;
```

Multi-Segment VHDL Description of FSM

```
    -- Mealy output logic
       process(state_reg, mem, rw)
          begin
          we_me <= '0'; -- default value
          case state_reg is
             when idle =>
                if (mem = '1') and (rw = '0') then
                  we_me <= '1';
                end if;
             when write =>
             when read1 =>
             when read2 =>
             when read3 =>
             when read4 =>
          end case;
       end process;
    end mult_seg_arch;
```

Two-Segment VHDL Description of FSM

Combine next-state/output logic into one process




```
architecture two_seg_arch of mem_ctrl is
   type mc_state_type is
        (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
   begin
```

Two-Segment VHDL Description of FSM

```
-- state register
   process(clk, reset)
      begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;

-- next-state logic and output logic
   process(state_reg, mem, rw, burst)
      begin
      oe <= '0';     -- default values
      we <= '0';
      we_me <= '0';
```

Two-Segment VHDL Description of FSM

```
case state_reg is
   when idle =>
      if (mem = '1') then
         if (rw = '1') then
            state_next <= read1;
         else
            state_next <= write;
            we_me <= '1';
         end if;
      else
         state_next <= idle;
      end if;

   when write =>
      state_next <= idle;
      we <= '1';
```

Two-Segment VHDL Description of FSM

```
            when read1 =>
               if (burst='1') then
                  state_next <= read2;
               else
                  state_next <= idle;
               end if;
               oe <= '1';

            when read2 =>
               state_next <= read3;
               oe <= '1';

            when read3 =>
               state_next <= read4;
               oe <= '1';
```

Two-Segment VHDL Description of FSM

```
            when read4 =>
                state_next <= idle;
                oe <= '1';
        end case;
    end process;
end two_seg_arch;
```

State Assignment

State assignment is the process of assigning a **binary** representations to the set of symbolic states

Although any arbitrary assignment works for a synchronous FSM, some assignments reduce the complexity of next-state/output logic and allows faster operation

Typical assignment strategies:

- Binary -- requires *ceiling*($\log_2 n$)-bit register
- Gray -- also minimal size but may reduce complexity of next-state logic
- One-hot or Almost one-hot (includes "0 ...0") -- requires n-bit register

State Assignment

Example for memory controller:

Table 10.1 State assignment example

	Binary assignment	Gray code assignment	One-hot assignment	Almost one-hot assignment
idle	000	000	000001	00000
read1	001	001	000010	00001
read2	010	011	000100	00010
read3	011	010	001000	00100
read4	100	110	010000	01000
write	101	111	100000	10000

State assignment can be controlled in VHDL either ***implicitly*** or ***explicitly***

For *implicit state assignment*, use *user attributes* which acts as a "directive" to guide the CAD synthesis software

The 1076.6 RTL synthesis standard defines an attribute named *enum_encoding* for specifying the values for an enumeration data type

This **attribute** can be used for specifying state assignment, as shown below

State Assignment

```
type mc_state_type is (idle, write, read1, read2,
   read3, read4);
attribute enum_encoding: string;
attribute enum_encoding of mc_state_type:
   type is "0000 0100 1000 1001 1010 1011";
```

This user attribute is very common is should be accepted by most synthesis software

Explicit state assignment is accomplished by replacing the symbolic values with actual binary representations

```
architecture state_assign_arch of mem_ctrl is
   constant idle:  std_logic_vector(3 downto 0):="0000";
   constant write: std_logic_vector(3 downto 0):="0100";
   constant read1: std_logic_vector(3 downto 0):="1000";
   constant read2: std_logic_vector(3 downto 0):="1001";
   constant read3: std_logic_vector(3 downto 0):="1010";
   constant read4: std_logic_vector(3 downto 0):="1011";
```

State Assignment

```
        signal state_reg, state_next:
           std_logic_vector(3 downto 0);
        begin

    -- state register
        process(clk, reset)
           begin
           if (reset = '1') then
              state_reg <= idle;
           elsif (clk'event and clk = '1') then
              state_reg <= state_next;
           end if;
        end process;

    -- next-state logic
        process(state_reg, mem, rw, burst)
           begin
```

State Assignment

```
case state_reg is
   when idle =>
      if (mem = '1') then
         if (rw = '1') then
            state_next <= read1;
         else
            state_next <= write;
         end if;
      else
         state_next <= idle;
      end if;

   when write =>
      state_next <= idle;

   when read1 =>
      if (burst = '1') then
```

State Assignment

```
                    state_next <= read2;
                else
                    state_next <= idle;
                end if;

            when read2 =>
                state_next <= read3;

            when read3 =>
                state_next <= read4;

            when read4 =>
                state_next <= idle;
    -- Need this now to cover other std_logic_vector vals
            when others =>
                state_next <= idle;
        end case;
    end process;
```

State Assignment

```
    -- Moore output logic
       process(state_reg)
          begin
          we <= '0'; -- default value
          oe <= '0';  -- default value
          case state_reg is
             when idle =>
             when write =>
                we <= '1';
             when read1 =>
                oe <= '1';
             when read2 =>
                oe <= '1';
             when read3 =>
                oe <= '1';
             when read4 =>
               oe <= '1';
             when others =>
```

State Assignment

```
            end case;
        end process;

    -- Mealy output logic
       we_me <= '1' when ((state_reg = idle) and
                          (mem = '1') and(rw = '0')) else
                '0';
    end state_assign_arch;
```

Moore Output Buffering

Output buffering involves adding a D FF to drive the output signal

The purpose is to remove glitches (and minimize clock-to-**output** delay (T_{co}))

The disadvantage is that the output is **delayed** by one clock cycle

However, for a Moore output, it is possible to obtain a buffered signal **without** this delay penalty.

Moore Output Buffering

There are two possible solutions

• Buffering by clever state assignment

A Moore output is shielded from *glitches* in the input signals, but not from glitches in the state transition and output logic

Glitches in the state transition can result from **multiple-bit** transitions of the state register, e.g., from the "111" to "000" states

Even though the state registers are controlled by the same clk, variations in the T_{cq} of the D FFs can produce glitches

Recall that T_{co} is the sum of T_{cq} and T_{output}

One way to reduce the effect on T_{co} introduced by the output logic is to eliminate it completely by clever state assignment

To accomplish this, add bits to the state encoding that specify the behavior of the output signals

Moore Output Buffering

You will also need to specify state assignment *explicitly*

Consider the memory controller -- we can specify the state of the outputs *oe* and *we* in bits q_3 and q_2 and the actual state in bits q_1 and q_0.

Table 10.2 Clever assignment

	q_3q_2 (oe) (we)	q_1q_0	$q_3q_2q_1q_0$
idle	00	00	0000
read1	10	00	1000
read2	10	01	1001
read3	10	10	1010
read4	10	11	1011
write	01	00	0100

This encoding scheme was used in the previous code segment

So, we see that *oe* and *we* are given directly by *state_reg(3)* and *state_reg(2)*

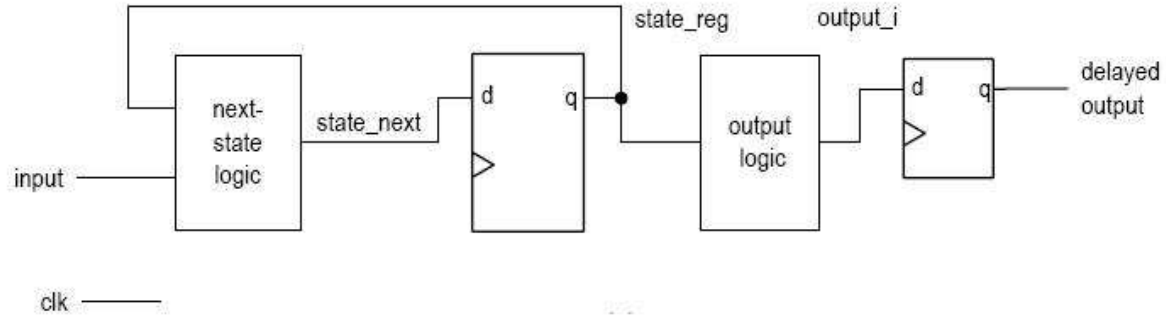
```
oe <= state_reg(3); -- modify the previous code seg by
we <= state_reg(2); -- replacing output logic with these
```

Therefore, the output logic is eliminated and T_{co} is reduced to T_{cq}

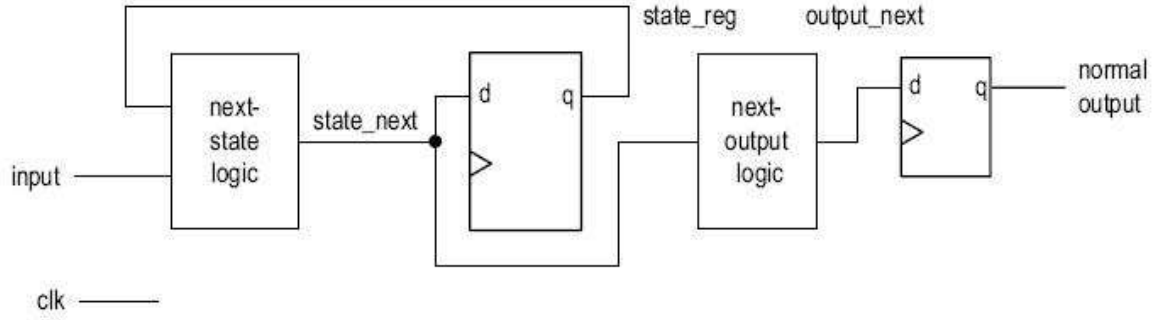
Unfortunately, this scheme is difficult to modify and maintain

Look-Ahead Output Circuit

A more systematic approach to eliminate the one-clock output buffer delay is to use the *state_next* signal instead of the *state_reg* signal



(a) Moore output with a regular output buffer



(b) Moore output with a look-ahead buffer

This works because the next output signal is a function of the next state logic

Only drawback is that the critical path is likely extended through the *next output* logic

Look-Ahead Output Circuit

```
architecture look_ahead_buffer_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
   signal oe_next, we_next, oe_buf_reg, we_buf_reg:
      std_logic;
   begin

   -- state register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;
```

Look-Ahead Output Circuit

```
-- output buffer
process(clk, reset)
   begin
   if (reset = '1') then
      oe_buf_reg <= '0';
      we_buf_reg <= '0';
   elsif (clk'event and clk = '1') then
      oe_buf_reg <= oe_next;
      we_buf_reg <= we_next;
   end if;
end process;

-- next-state logic
process(state_reg, mem, rw, burst)
   begin
   case state_reg is
```

Look-Ahead Output Circuit

```
when idle =>
   if (mem = '1') then
      if (rw = '1') then
         state_next <= read1;
      else
         state_next <= write;
      end if;
   else
      state_next <= idle;
   end if;

when write =>
   state_next <= idle;

when read1 =>
   if (burst = '1') then
      state_next <= read2;
```

Look-Ahead Output Circuit

```
                else
                    state_next <= idle;
                end if;

            when read2 =>
                state_next <= read3;

            when read3 =>
                state_next <= read4;

            when read4 =>
                state_next <= idle;
        end case;
    end process;
```

Look-Ahead Output Circuit

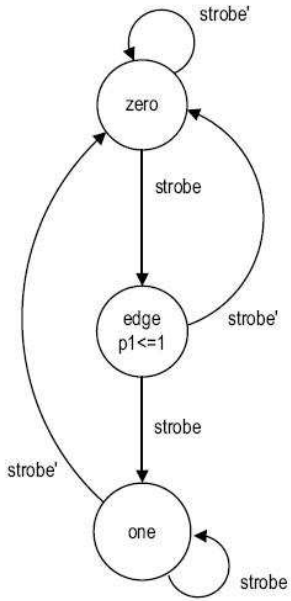
```
-- look-ahead output logic
process(state_next)
   begin
   we_next <= '0'; -- default value
   oe_next <= '0'; -- default value
   case state_next is
      when idle =>
      when write =>
         we_next <= '1';
      when read1 =>
         oe_next <= '1';
      when read2 =>
         oe_next <= '1';
      when read3 =>
         oe_next <= '1';
      when read4 =>
         oe_next <= '1';
   end case; end process;
```

Look-Ahead Output Circuit

```
        -- output
        we <= we_buf_reg;
        oe <= oe_buf_reg;
    end look_ahead_buffer_arch;
```

FSM Design Examples

Edge detecting circuit (Moore)



The VHDL code for version 1 of edge detection circuit we saw earlier

Edge Detection Circuit

```
library ieee;
use ieee.std_logic_1164.all;

entity edge_detector1 is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      p1: out std_logic
   );
end edge_detector1;

architecture moore_arch of edge_detector1 is
   type state_type is (zero, edge, one);
   signal state_reg, state_next: state_type;
   begin
```

Edge Detection Circuit

```
-- state register
process(clk, reset)
   begin
   if (reset = '1') then
      state_reg <= zero;
   elsif (clk'event and clk = '1') then
      state_reg <= state_next;
   end if;
end process;

-- next-state logic
process(state_reg, strobe)
   begin
   case state_reg is
      when zero=>
         if (strobe = '1') then
            state_next <= edge;
         else
```

Edge Detection Circuit

```
                    state_next <= zero;
                end if;

            when edge =>
                if (strobe = '1') then
                    state_next <= one;
                else
                    state_next <= zero;
                end if;

            when one =>
                if (strobe = '1') then
                    state_next <= one;
                else
                    state_next <= zero;
                end if;
        end case;
    end process;
```

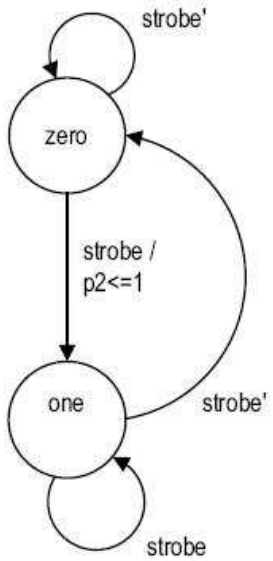
```
    -- Moore output logic
    p1 <= '1' when state_reg = edge else
          '0';
end moore_arch;
```

If we need to the output to be glitch-free, we can use the *clever state assignment* shown below or the *look-ahead* output scheme

	state_reg(1) (p1)	state_reg(0)
zero	0	0
edge	1	0
one	0	1

Edge Detection Circuit

Edge detecting circuit (Mealy)




```
library ieee;
use ieee.std_logic_1164.all;

entity edge_detector2 is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      p2: out std_logic
```

```
    );
end edge_detector2;

architecture mealy_arch of edge_detector2 is
   type state_type is (zero, one);
   signal state_reg, state_next: state_type;
   begin

   -- state register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= zero;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;
```

Edge Detection Circuit

```
-- next-state logic
process(state_reg, strobe)
   begin
   case state_reg is
      when zero=>
         if (strobe = '1') then
            state_next <= one;
         else
            state_next <= zero;
         end if;
      when one =>
         if (strobe = '1') then
            state_next <= one;
         else
            state_next <= zero;
         end if;
   end case;
end process;
```

Edge Detection Circuit

```
        -- Mealy output logic
        p2 <= '1' when (state_reg = zero) and (strobe = '1')
                    else
              '0';
    end mealy_arch;
```

An alternative to deriving the edge detection circuit is to treat it as a regular sequential circuit and design it in an *ad hoc* manner

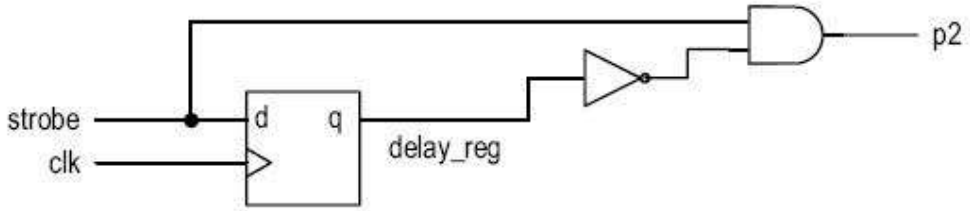


Figure 10.19 Direct implementation of an edge detector.

Output *p2* is asserted when the previous value in FF is '0' and the new value is (*strobe*) is '1' -- this represents an edge

Note that the output is a Mealy output (subject to glitches) -- what does the timing diagram look like?

Edge Detection Circuit

```
architecture direct_arch of edge_detector2 is
   signal delay_reg: std_logic;
   begin

   -- delay register
   process(clk, reset)
      begin
      if (reset = '1') then
         delay_reg <= '0';
      elsif (clk'event and clk = '1') then
         delay_reg <= strobe;
      end if;
   end process;

   -- decoding logic
   p2 <= (not delay_reg) and strobe;
end direct_arch;
```

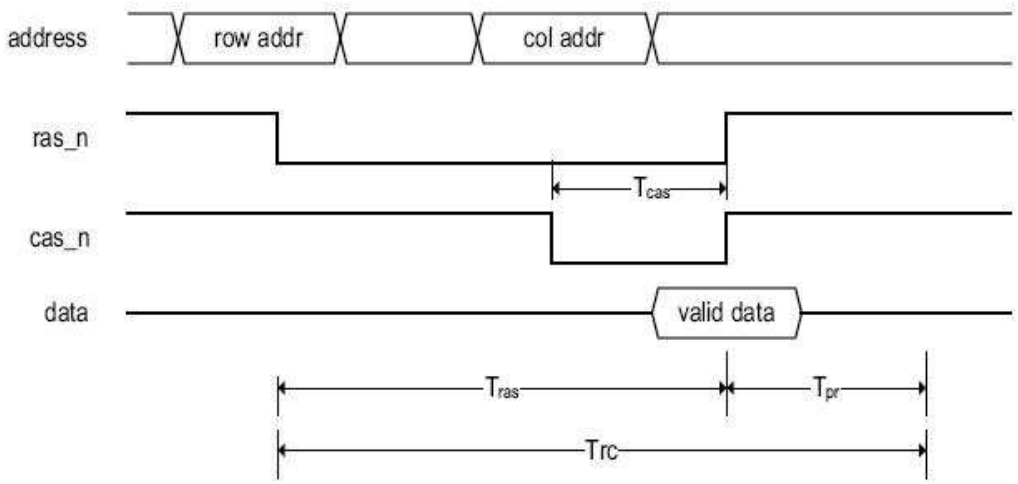
Text covers an **Arbiter** circuit

DRAM Strobe Signal Generation

The address signals of a DRAM are split into two parts, *row* and *column*

They are sent to the DRAM from the controller in a time-multiplexed manner

Two signals, *ras_n* (active low row access strobe) and *cas_n* are **de-asserted** to instruct the DRAM to latch the addresses internally



(a) Simplified timing of a DRAM read cycle

There are several timing parameters associated with a (simplified) DRAM

- T_{ras} and T_{cas}: *ras*/*cas* access time -- time required to obtain output data after *ras_n*/*cas_n* are de-asserted

DRAM Strobe Signal Generation

- T_{pr}: precharge time -- the time to recharge the DRAM cell to restore the destroyed original value after a read
- T_{rc}: read cycle -- minimum elapsed time between two read operations

DRAMs are asynchronous (do not have a clk input)

Instead the strobe signals have to de-asserted in a proper sequence and be held long enough to allow for decoding, multiplexing and recharging

A memory controller is the interface between a DRAM device and a **synchronous** system

Its primary function is to generate the proper strobe signals

A full blown read controller should contain registers to store address and data, plus extra control signals to coordinate the address and data bus operations

Assume our DRAM card has the following parameters

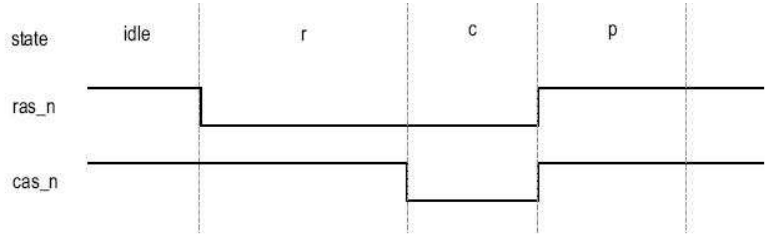
- 120 ns DRAM (T_{rc} = 120 ns):
- T_{ras} = 85 ns, T_{cas} = 20 ns, T_{pr} = 35 ns

DRAM Strobe Signal Generation

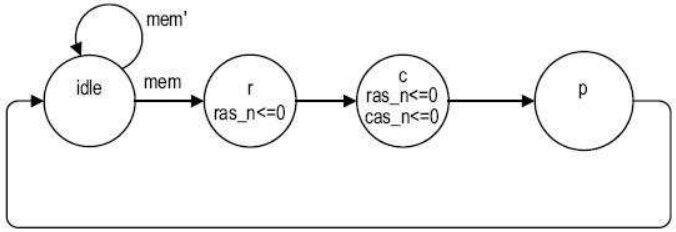
Our task is to design an FSM that generates the strobe signals, *ras_n* and *cas_n* after the input command signal *mem* is asserted

From the timing diagram

- *ras_n* is de-asserted first for 65 ns (output pattern of FSM is "01" in this interval
- *cas_n* is then de-asserted for at least 20 ns (output pattern is "00")
- The *ras_n* and *cas_n* signals are *re-asserted* for at least 35 ns ("11")



(b) State of the strobe signals



Three states + *idle* for no-op

(c) State diagram of slow strobe generation

First design uses *state* to generate the pattern and divides a read cycle into three states, *r*, *c* and *p*

DRAM Strobe Signal Generation

We also use a Moore machine because it has *better control* over the width of the intervals (level-sensitive) and the outputs can be easily made *glitch-free*

For this design, clock cycle needs to be at least 65 ns to satisfy the timing constraints

Therefore, this is a slow design because read cycle time is 195 ns (3*65 ns)

```
library ieee;
use ieee.std_logic_1164.all;

entity dram_strobe is
   port(
      clk, reset: in std_logic;
      mem: in std_logic;
      cas_n, ras_n: out std_logic
   );
end dram_strobe;
```

DRAM Strobe Signal Generation

```
architecture fsm_slow_clk_arch of dram_strobe is
   type fsm_state_type is (idle, r, c, p);
   signal state_reg, state_next: fsm_state_type;
   begin

-- state register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;
```

DRAM Strobe Signal Generation

```
-- next-state logic
   process(state_reg, mem)
      begin
      case state_reg is
         when idle =>
            if (mem = '1') then
               state_next <= r;
            else
               state_next <= idle;
            end if;

         when r =>
            state_next <=c;

         when c =>
            state_next <=p;
```

DRAM Strobe Signal Generation

```
            when p =>
               state_next <=idle;
         end case;
      end process;

   -- output logic
      process(state_reg)
         begin
         ras_n <= '1';
         cas_n <= '1';
         case state_reg is
            when idle =>
            when r =>
               ras_n <= '0';
            when c =>
               ras_n <= '0';
               cas_n <= '0';
```

DRAM Strobe Signal Generation

```
            when p =>
        end case;
    end process;
end fsm_slow_clk_arch;
```

Since the strobe signals are **level-sensitive**, we have to ensure that these signals are glitch-free by, e.g., adding a *look-ahead* output buffer

A faster design must use a clock period that is **smaller** to accommodate the differences in the three intervals

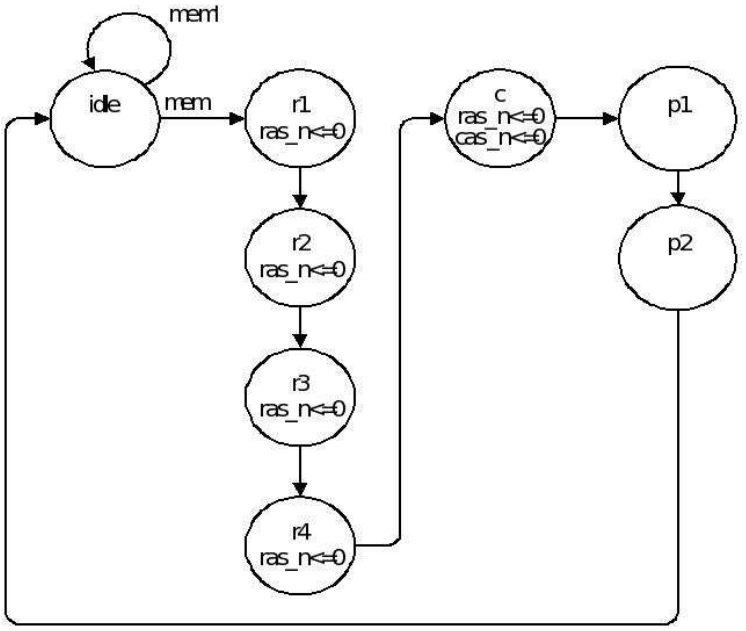
For example, if we use a 20 ns clock period then the three output patterns need

- *ceiling(65/20)* or 4 states for *r*
- *ceiling(20/20)* or 1 state for *c*
- *ceiling(35/20)* or 2 states for *p*

This reduces the read cycle to 140 ns (7*20 ns) -- down from 195 ns

DRAM Strobe Signal Generation

One way to implement this is to split the *r* and *p* states -- make multiple states where one existed originally



The minimum read cycle time for the memory can be achieved using a clock period of 5 ns (largest factor evenly divisible into all three parameters)

This would yield 13 states + 4 states + 7 states for *r*, *c* and *p*, respectively

A better approach is to use *counters* in each state as we will see later

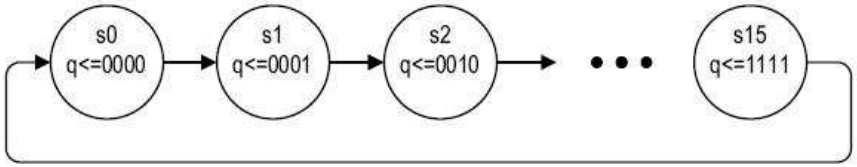
DRAM Strobe Signal Generation

Text covers a **Manchester encoding** circuit

In reality, all sequential circuits, including *regular sequential* circuits, can be modeled by FSMs

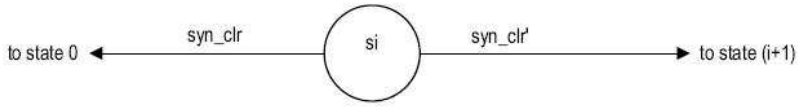
Consider a free-running *mod-16* binary counter consider earlier

Expressed as an FSM, it is an extremely *regular* structure with 16 states



We can modify this easily to add 'features' as we did earlier

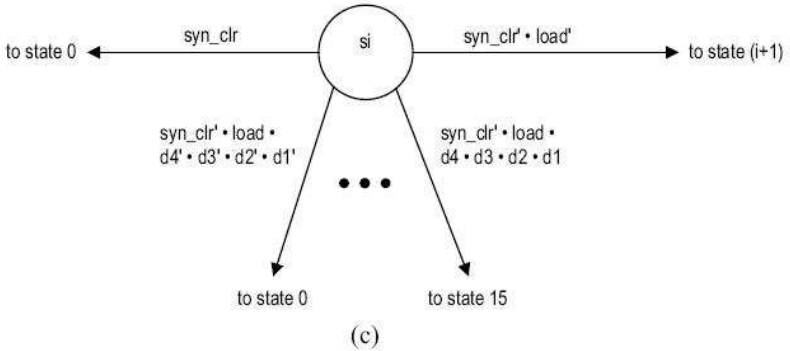
(a) ← Original



← To add synchronous *clear*

DRAM Strobe Signal Generation

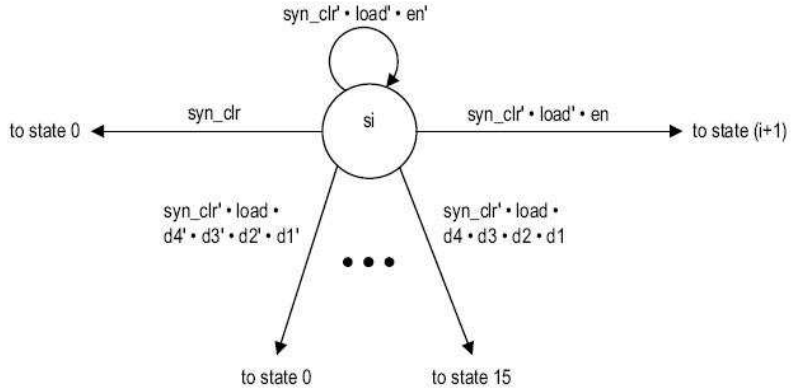
To add the *load* operation, need to add 1 control signal and a *4*-bit data signal



(c)

To add *load*

Note: 16 additional transitions are needed here



To add *enable*

Note: Logic expression establish priority with *syn_clr* highest, followed by *load* and then *enable*

This becomes extremely tedious, especially for larger counters

Therefore, for regular sequential circuits, we do NOT employ this strategy