ECE381(CAD), Lecture 10:

FSM and FSMD

Mehdi Modarressi

Department of Electrical and Computer Engineering,

University of Tehran

Some pictures and examples are taken from "FPGA Prototyping By VHDL Examples- Xilinx Spartan-3 Version"

FSM and FSMD

- Reading:
  - "FPGA Prototyping By VHDL Examples- Xilinx Spartan-3 Version", Chapters 5 and 6

Introduction

- FSM (finite state machine)
- FSMD (finite state machine with datapath)
- Some design examples
  - Pipeline
  - Bus
  - Debouncer
  - ….

Datapath and control path

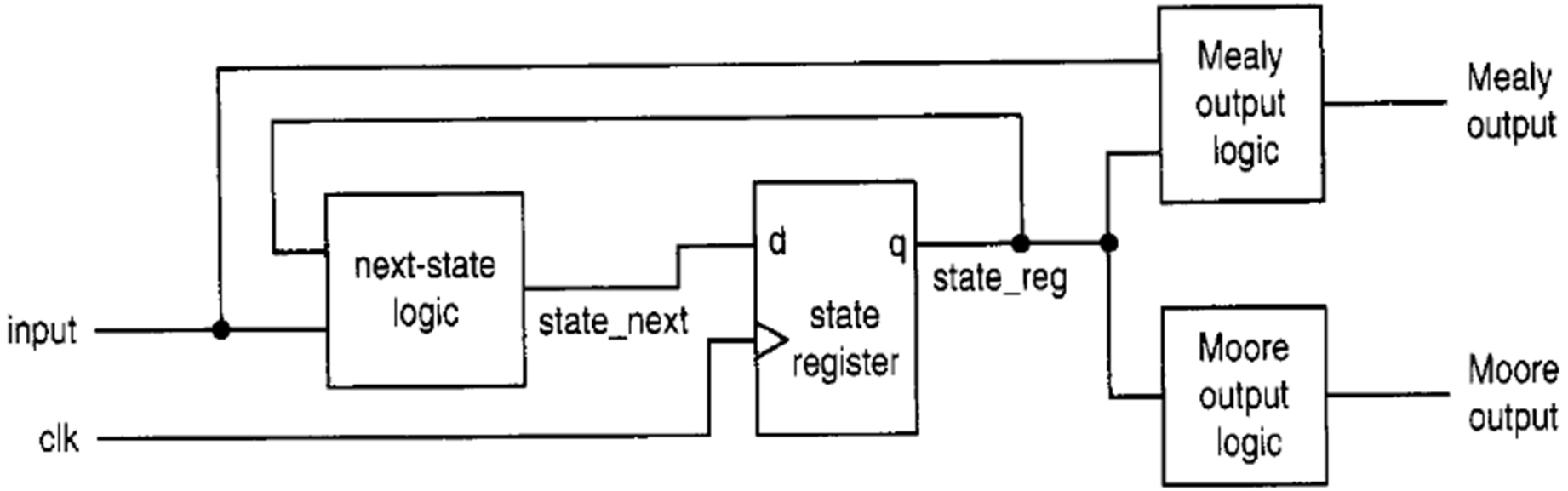
- Each digital circuit is composed of two parts
  - Control path
  - Data path
- Datapath is the functional units that implement the computation or required operations and services
  - Like ALU in a CPU
- Control path controls the behavior of the functional units
  - Activating datapath components, feeding data to them,…

FSM

- Control path operation: Examines the external commands and also the current status of the system and activates proper control signals to control operation of a datapath
- FSMs are used to describe the control path of a digital circuit

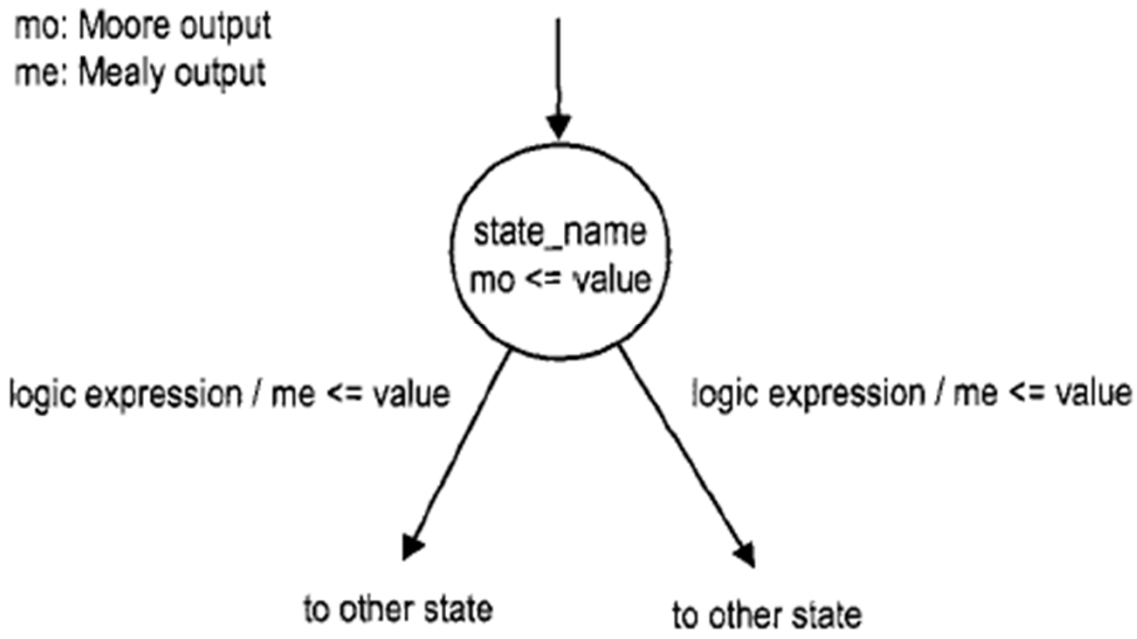
FSM

- Moore: the output is only a function of state
- Mealy : the output is a function of state and external input



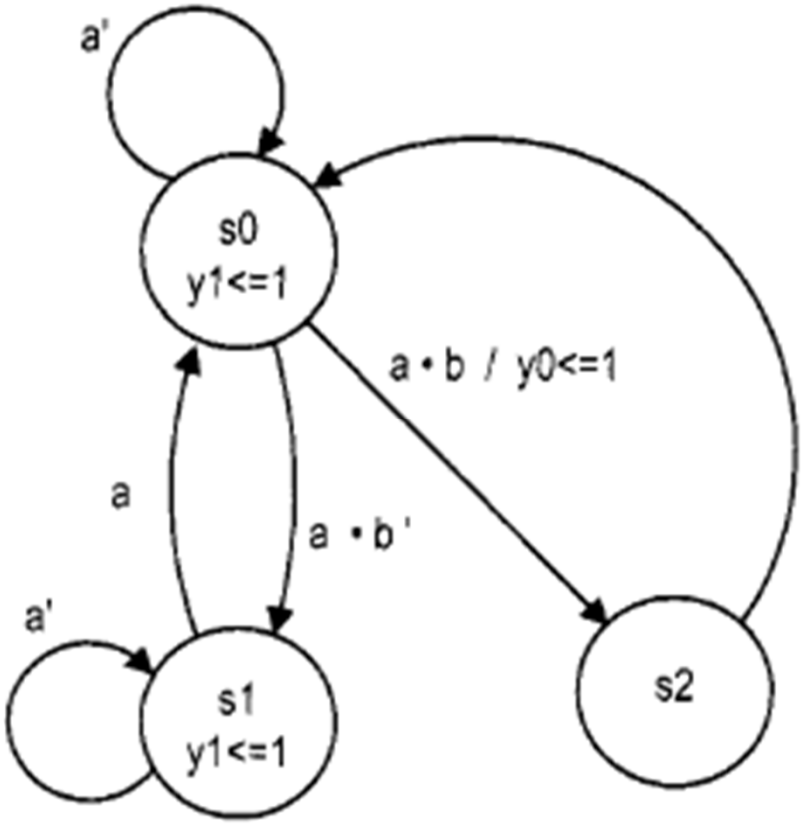
FSM representation

- An FSM is usually specified by a state diagram
- A state diagram is composed of nodes, which represent states and are drawn as circles, and annotated transitional arcs



FSM modeling by VHDL

- Example from "The designers guide to VHDL", page 110
- An FSM with both Mealy and Moore outputs



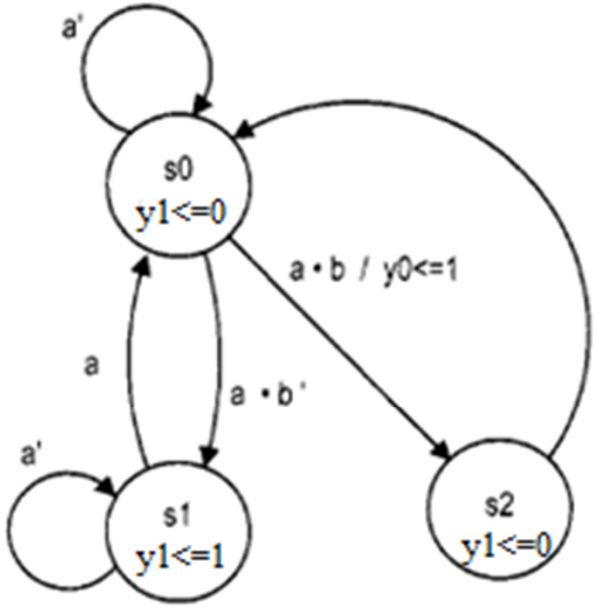
FSM modeling by VHDL

- Use the VHDL's enumerated data type to represent the FSM's states
- 3-process coding method
  - Use one process for state transition, one for next state calculation, and one for output generation

FSM modeling by VHDL

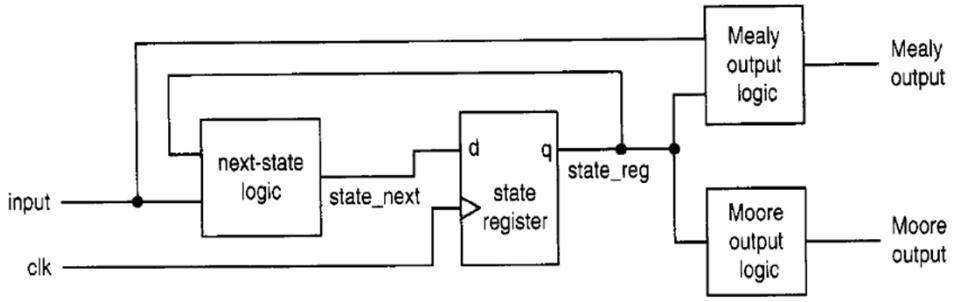
```
library ieee;
use ieee.std_logic_1164.all;
entity fsm_eg is
    port(
        clk, reset: in std_logic;
        a, b: in std_logic;
        y0, y1: out std_logic
    );
end fsm_eg;

architecture mult_seg_arch of fsm_eg is
    type eg_state_type is (s0, s1, s2);
    signal state_reg, state_next: eg_state_type;
begin
```

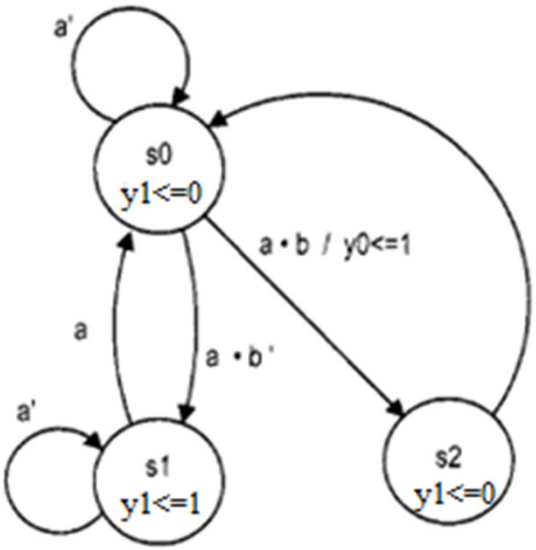


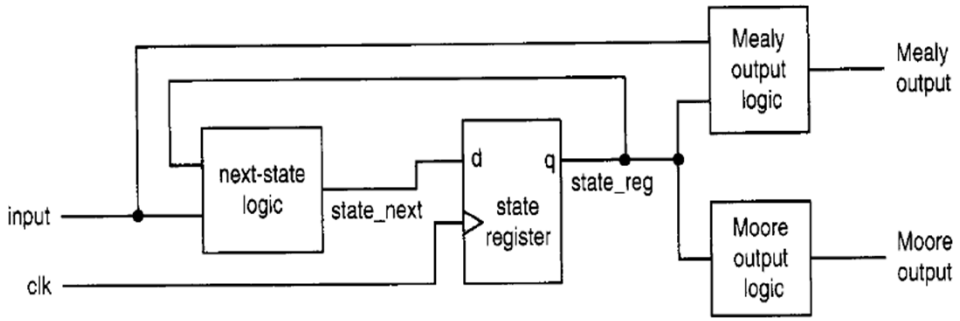
PROCESS-1: State transition

```
-- state register
process(clk,reset)
begin
   if (reset='1') then
      state_reg <= s0;
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
   end if;
end process;
```



PROCESS-2: State calculation



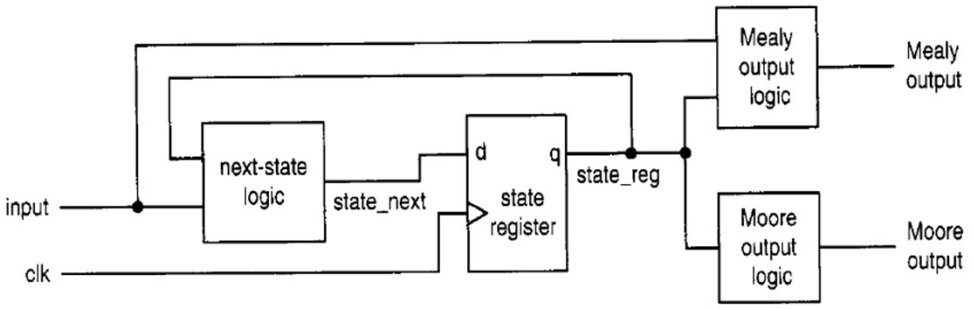



```
-- next-state logic
process(state_reg,a,b)
begin
   case state_reg is
      when s0 =>
         if a='1' then
            if b='1' then
               state_next <= s2;
            else
               state_next <= s1;
            end if;
         else
            state_next <= s0;
         end if;
      when s1 =>
         if (a='1') then
            state_next <= s0;
         else
            state_next <= s1;
         end if;
      when s2 =>
         state_next <= s0;
   end case;
end process;
```

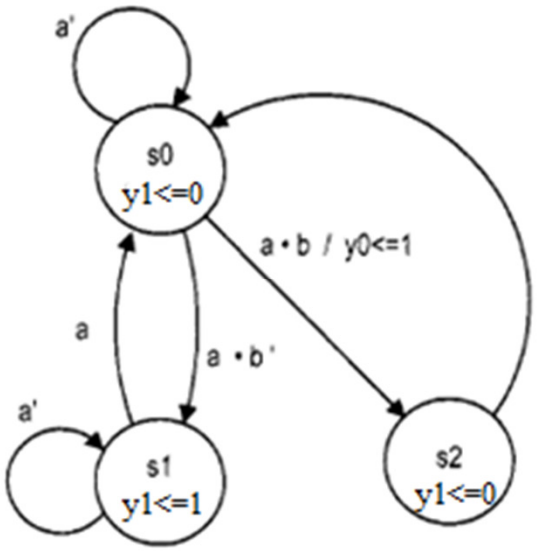
PROCESS-3: Output generation

```
-- Mealy output logic
process(state_reg,a,b)
begin
   case state_reg is
      when s0 =>
         if (a='1') and (b='1') then
           y0 <= '1';
         else
           y0 <= '0';
         end if;
      when s1 | s2 =>
         y0 <= '0';
   end case;
end process;
end mult_seg_arch;
```

```
-- Moore output logic
process(state_reg)
begin
   case state_reg is
      when s0|s2 =>
         y1 <= '0';
      when s1 =>
         y1 <= '1';
   end case;
end process;
```

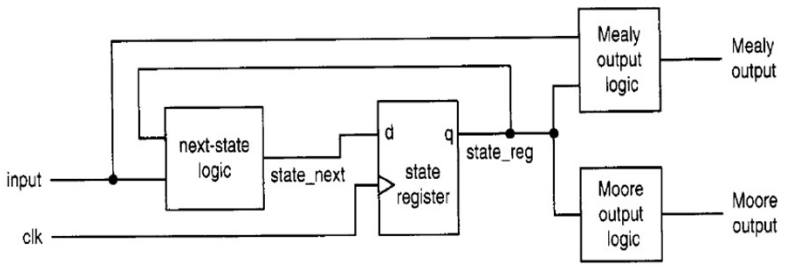


```
-- state register
process(clk,reset)
begin
   if (reset='1') then
      state_reg <= s0;
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
   end if;
end process;
```



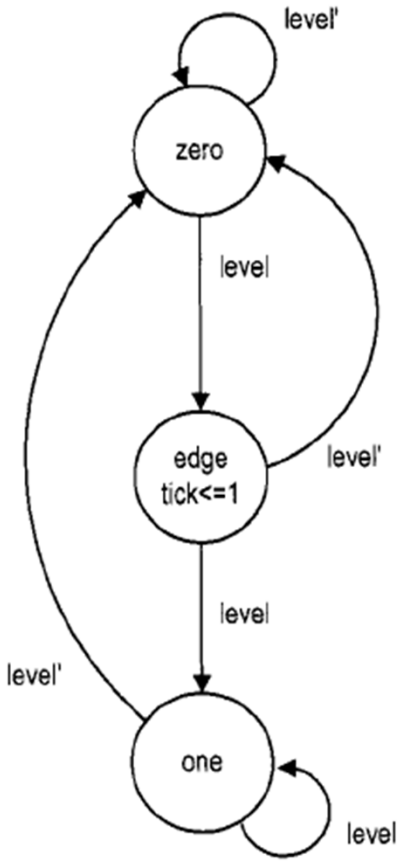

```
-- next-state logic
process(state_reg,a,b)
begin
   case state_reg is
      when s0 =>
         if a='1' then
            if b='1' then
               state_next <= s2;
            else
               state_next <= s1;
            end if;
         else
            state_next <= s0;
         end if;
      when s1 =>
         if (a='1') then
            state_next <= s0;
         else
            state_next <= s1;
         end if;
      when s2 =>
         state_next <= s0;
   end case;
end process;
```

```
-- Moore output logic
process(state_reg)
begin
   case state_reg is
      when s0|s2 =>
         y1 <= '0';
      when s1 =>
         y1 <= '1';
   end case;
end process;
```



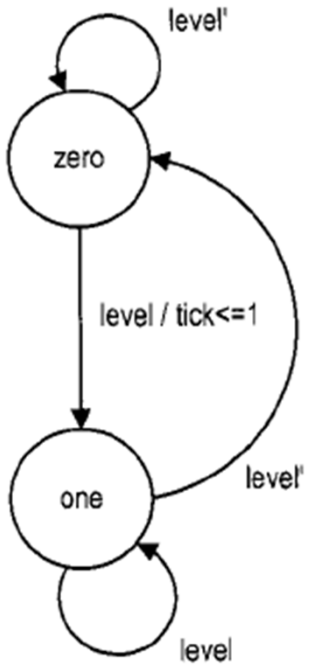
Example- Rising-edge detector

Moore



The zero and one states indicate that the input signal has been ’0’ and ’ 1’ for awhile

Mealy

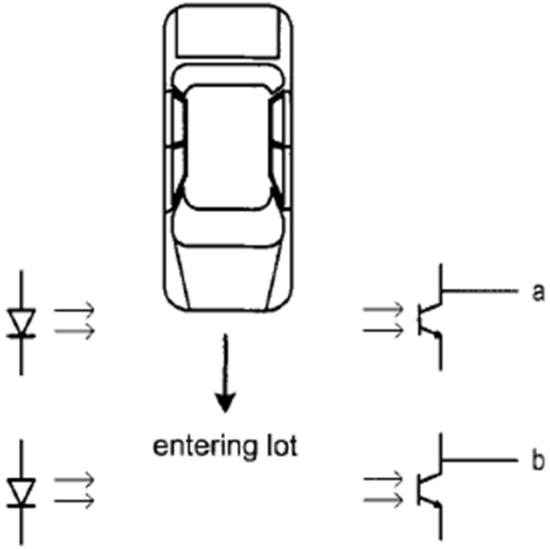


Exercise

Consider a parking lot with a single entry and exit gate. Two pairs of photo sensors are used to monitor the activity of cars, as shown in Figure 5.11. When an object is between the photo transmitter and the photo receiver, the light is blocked and the corresponding output is asserted to '1'. By monitoring the events of two sensors, we can determine whether a car is entering or exiting or a pedestrian is passing through. For example, the following sequence indicates that a car enters the lot:

- Initially, both sensors are unblocked (i.e., the a and b signals are "00").
- Sensor a is blocked (i.e., the a and b signals are "10").
- Both sensors are blocked (i.e., the a and b signals are "11").
- Sensor a is unblocked (i.e., the a and b signals are "01").
- Both sensors becomes unblocked (i.e., the a and b signals are "00").

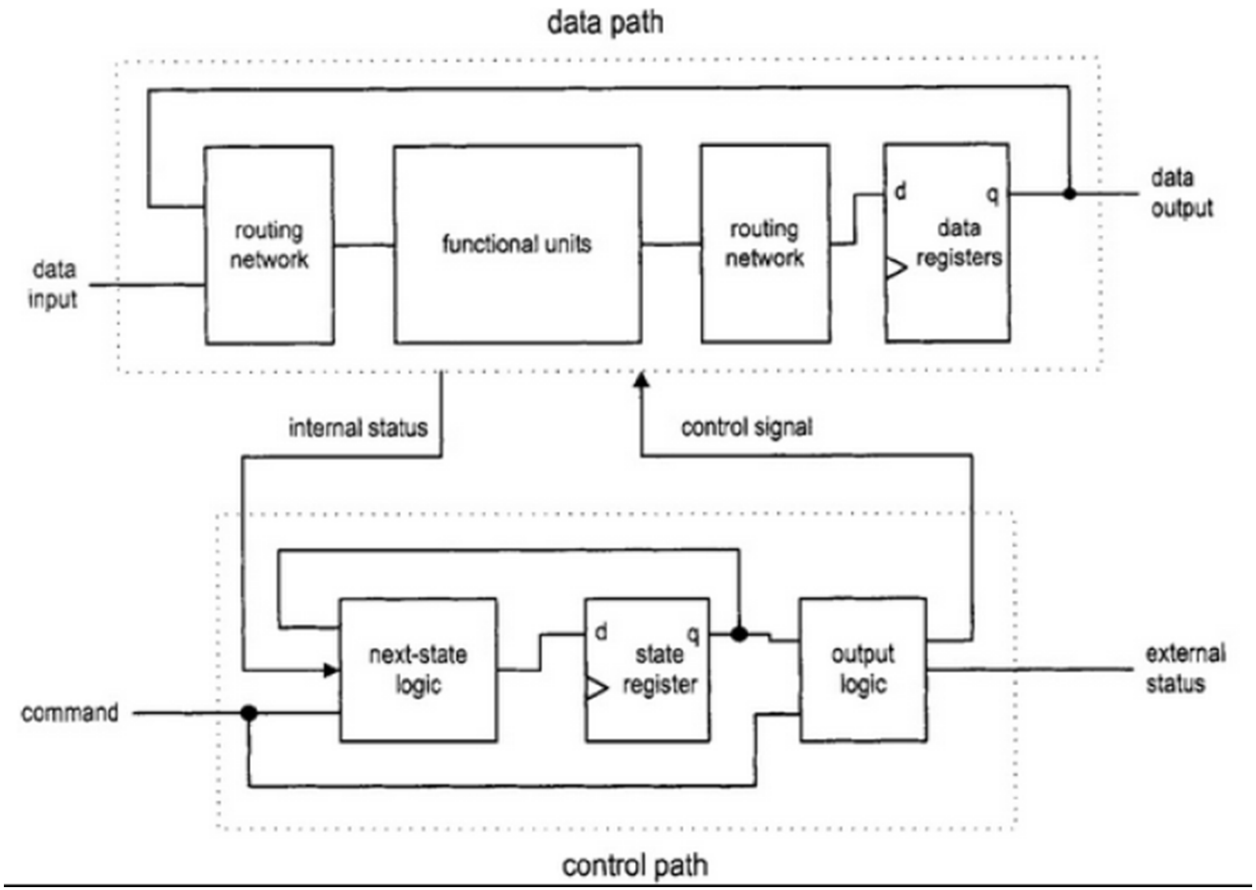
Design a parking lot occupancy counter as follows:



Design an FSM with two input signals a and b, and 4 output signals P_in, P_out, C_in, C_out that assert one clock signal when a pedestrian enters, a pedestrian exits, a car enters, and a car exits, respectively.
Implement the FSM by VHDL

FSMD

- Finite state machine with datapath
- Combines an FSM and regular sequential circuits



RTL

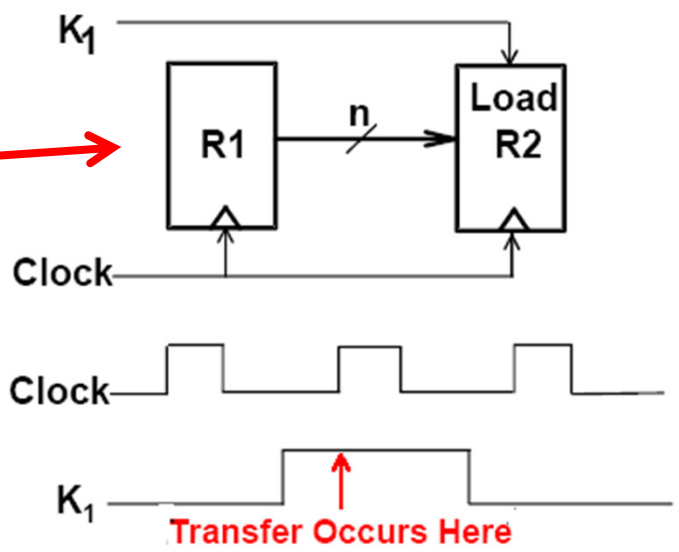
- The FSMD is used to implement systems described at the Register Transfer Level (RTL)
- An RT operation specifies data manipulation and transfer for a single destination register

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

- Elementary operations - called *microoperations*
  - load, count, shift, add, bitwise "OR", etc.
  - Example:
    - R0 $\leftarrow$ R1 $\oplus$ R2
    - R6 $\leftarrow$ R7+R8-1

RT description methodology

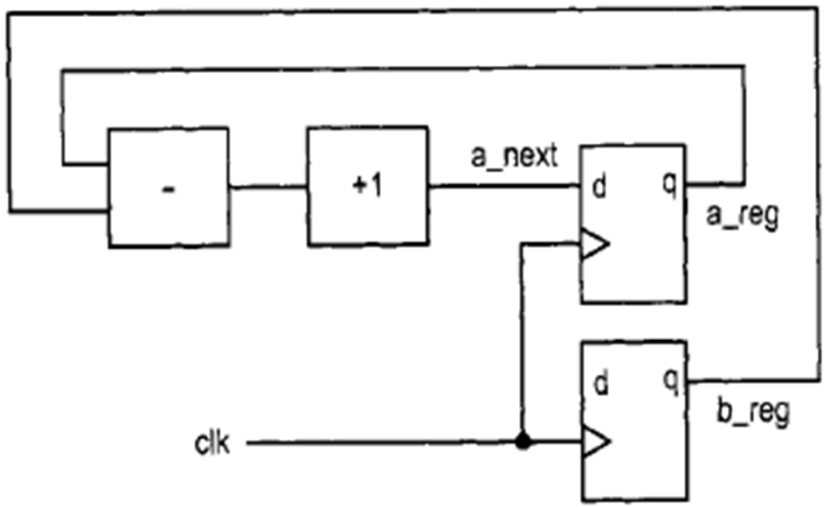
- Arrow (→): data transfer
  - e.g. R1 ← R2
- Brackets [ ]: Specifies a memory address
  - e.g. R0 ← M[AR]
- Comma: separates parallel operations
- Colon: states a condition
  - If (K1 =1) then (R2 ← R1)
  - ⇔ K1: (R2 ← R1)



RTL

- RT operations are synchronized by an embedded clock
- The result from the *f (.)* function is not stored to the destination register until the next rising edge of the clock

a ← a-b+1

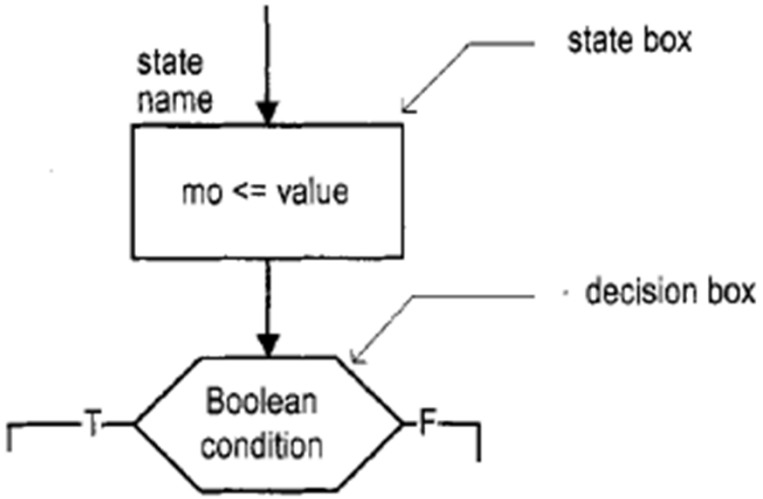


RTL

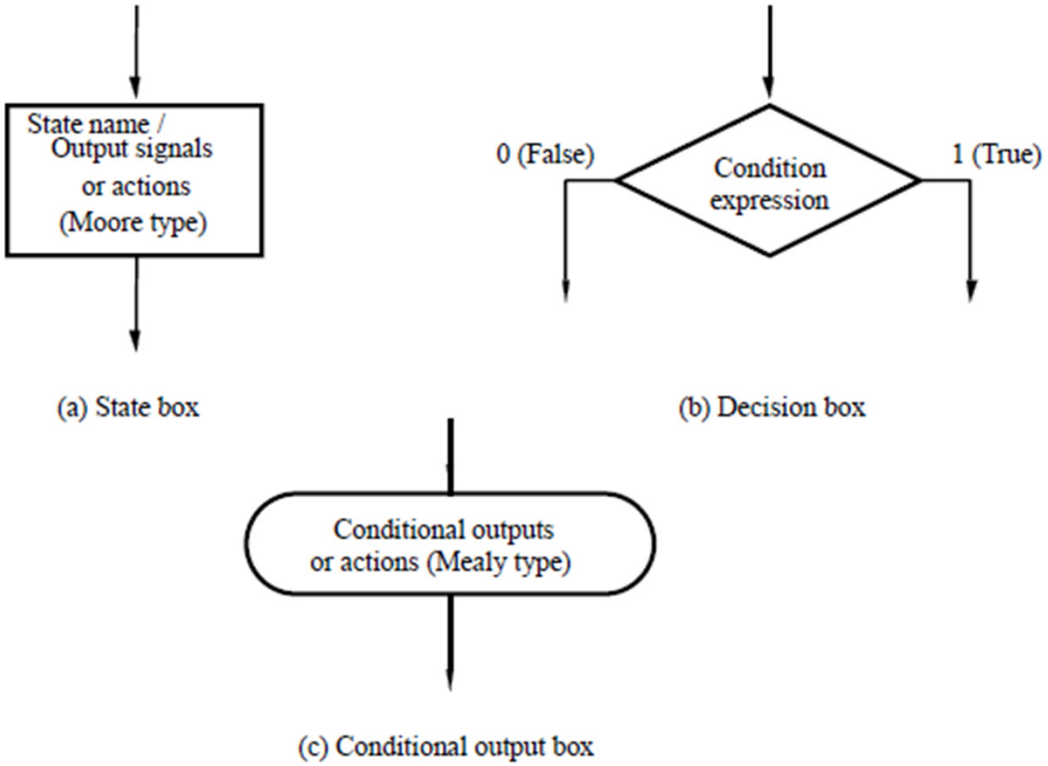
- A circuit based on the RT methodology specifies which RT operations should be executed in each step
- An RT operation is done in a clock-by-clock basis
  - Its timing is similar to a state transition of an FSM

ASM chart

- ASM: algorithmic state machine
- Composed of a network of ASM blocks
  - State blocks: represents a state in an FSM
  - Decision blocks: tests the input condition and determines which exit path to take



Elements of ASM chart



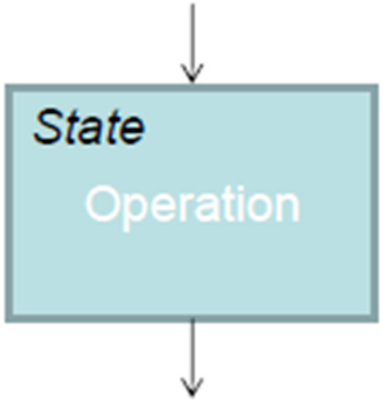
(a) State box

(b) Decision box

(c) Conditional output box

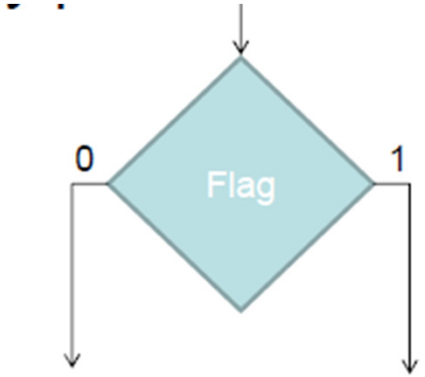
State box

- The symbol is a rectangle, with a label (state name) and output control operations(Moore type) written inside
- It has a single exit path (arrow)



Decision box

- The symbol is a diamond
- A condition variable and multiple exit paths, labeled
- with values of the input condition
- It has a single entry path.

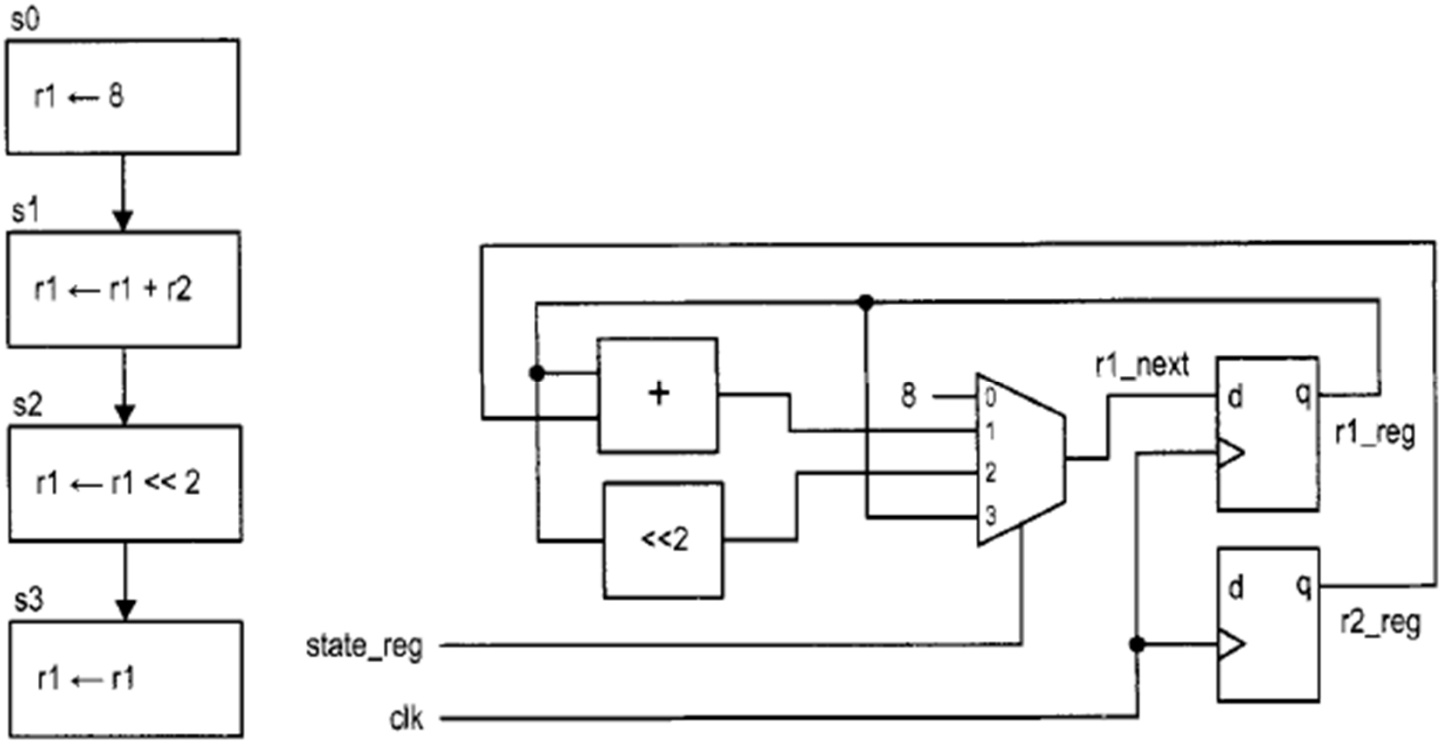


Conditional output box

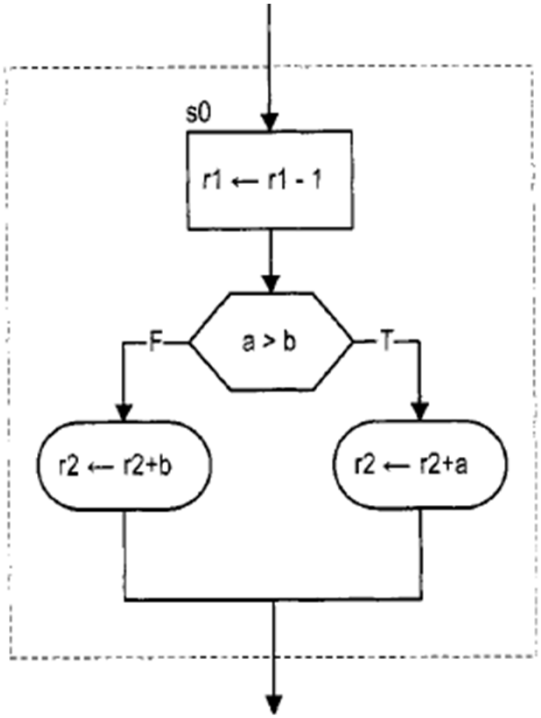
- The symbol is a rounded rectangle
- Denotes output signals that are of the Mealy type.
- Always after a decision box
- Single entry and exit path

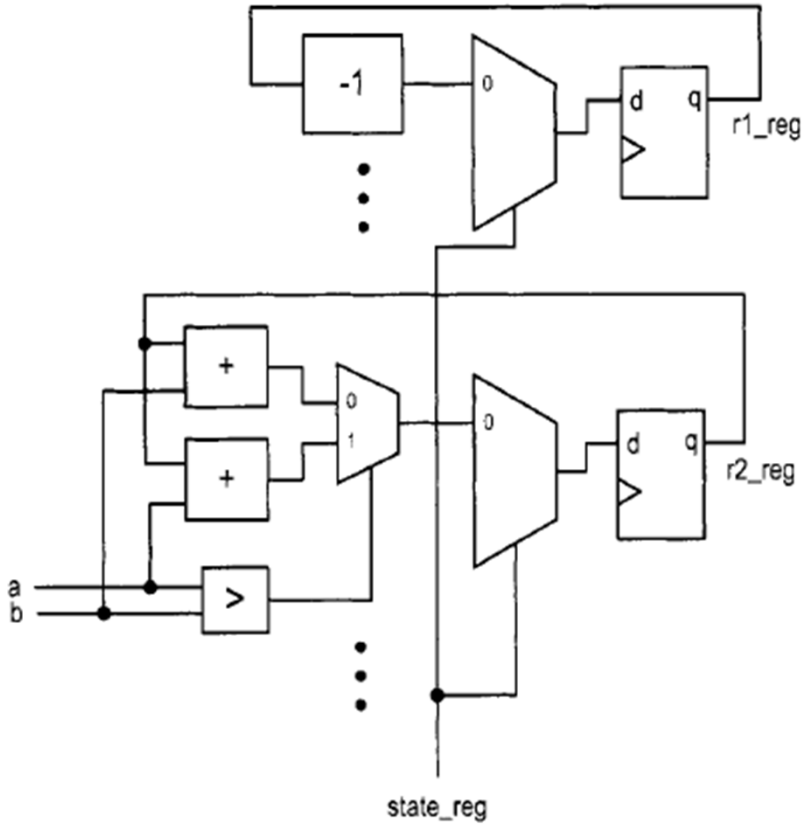
ASMD chart

- ASM with datapath
- A clock is embedded in the chart
  - Registers are updated when exiting the current ASMD block



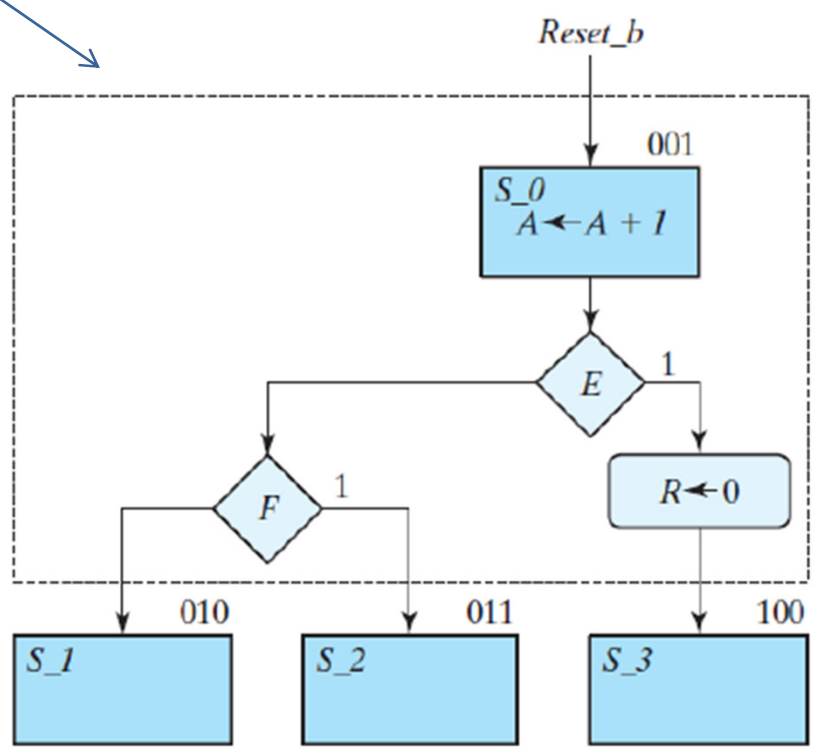
ASMD chart


s0
r1 ← r1 - 1
a > b
F
T
r2 ← r2+b
r2 ← r2+a


-1
0
d
q
r1_reg
+
+
0
1
>
a
b
0
d
q
r2_reg
state_reg

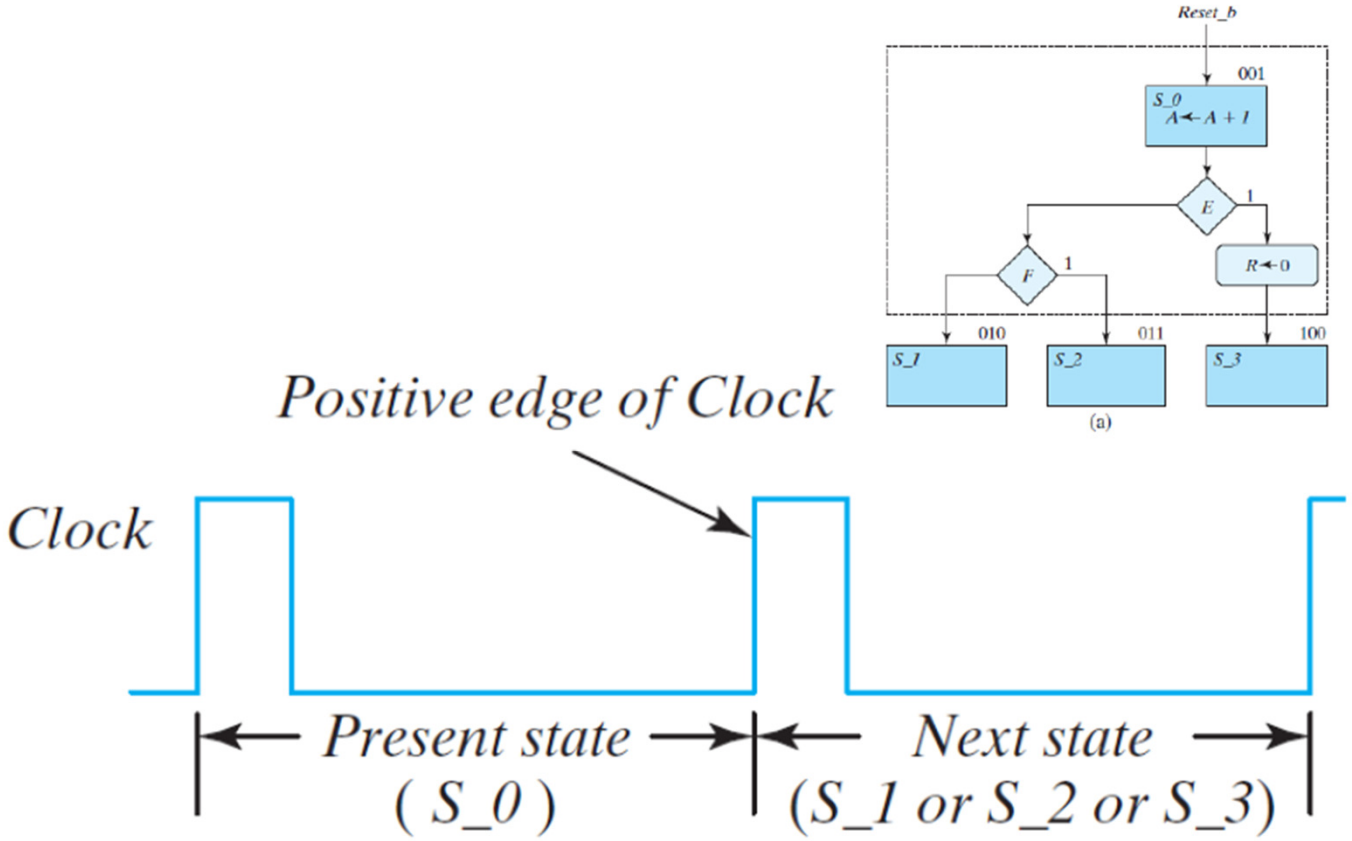
ASM block

- ASM block



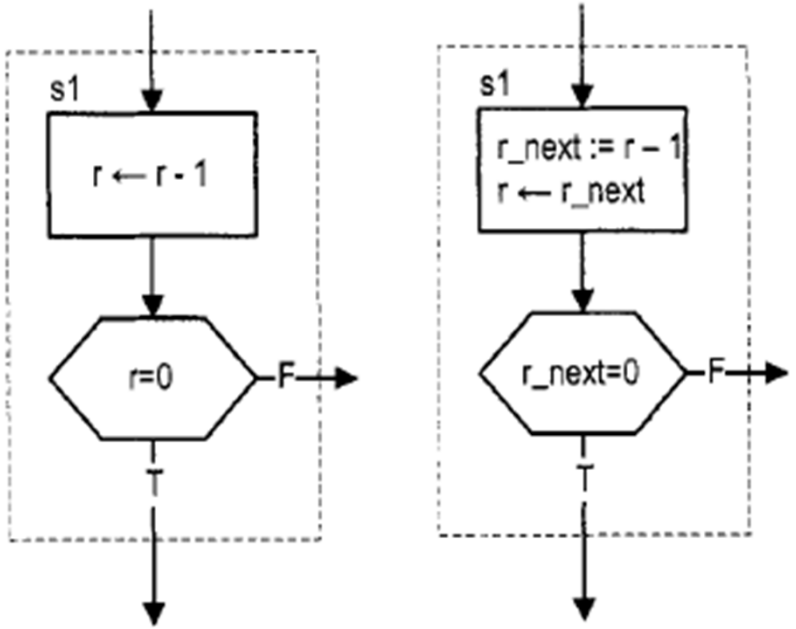
ASMD timing

- Control passes one branch in the ASM block during the clock period and waits for the clock to enter the next state box



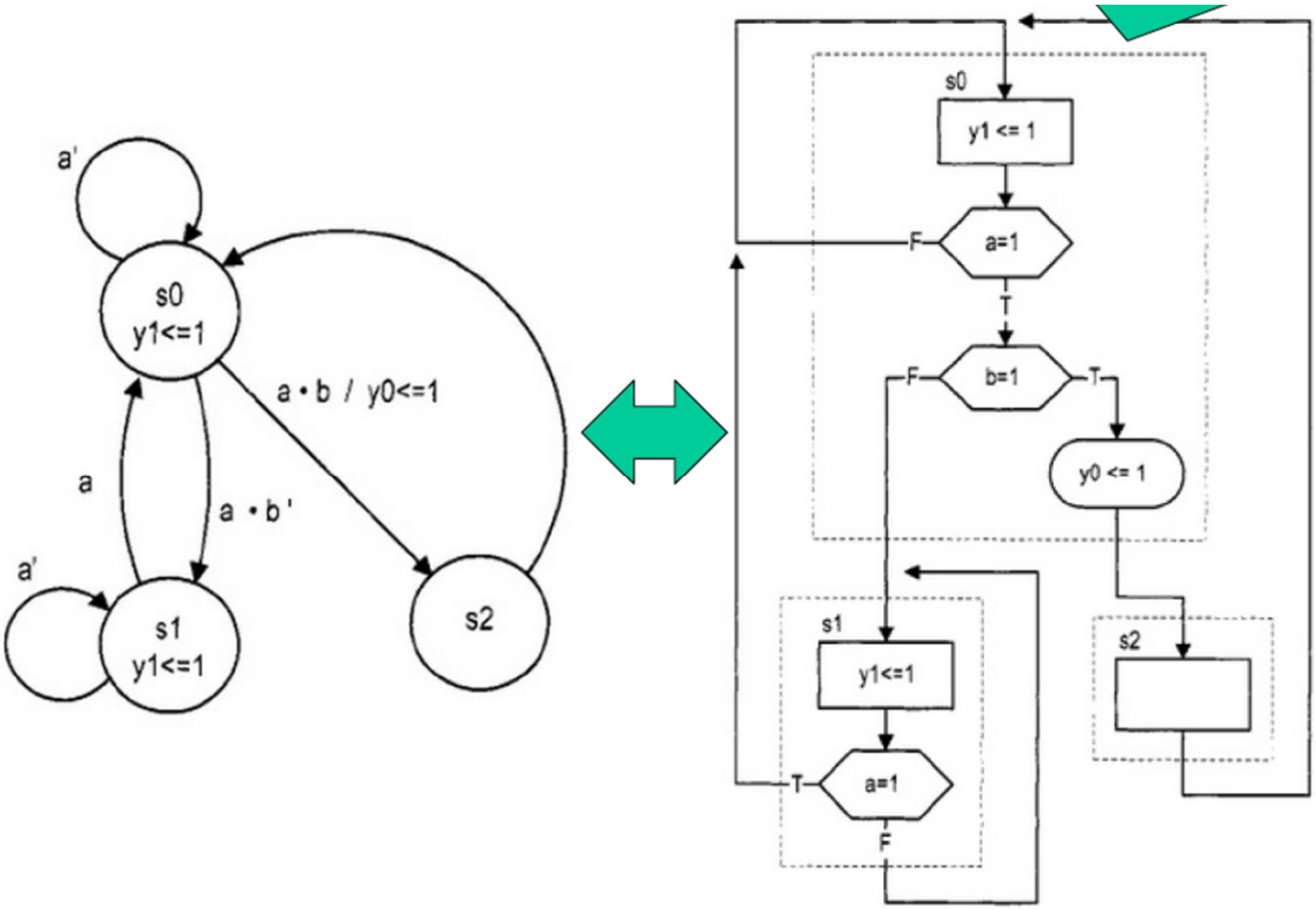
ASMD timing

- The destination registers are updated when exiting the current ASMD block, but not within the block
  - May introduce errors when a register is used in a decision box
- If the new value of *r* is desired, we should use *r_next*, the output of the combinational logic



" := " is used for immediate assignment

State diagram- ASM equivalence



Source: EE 459/500 – HDL Based Digital Design with Programmable Logic

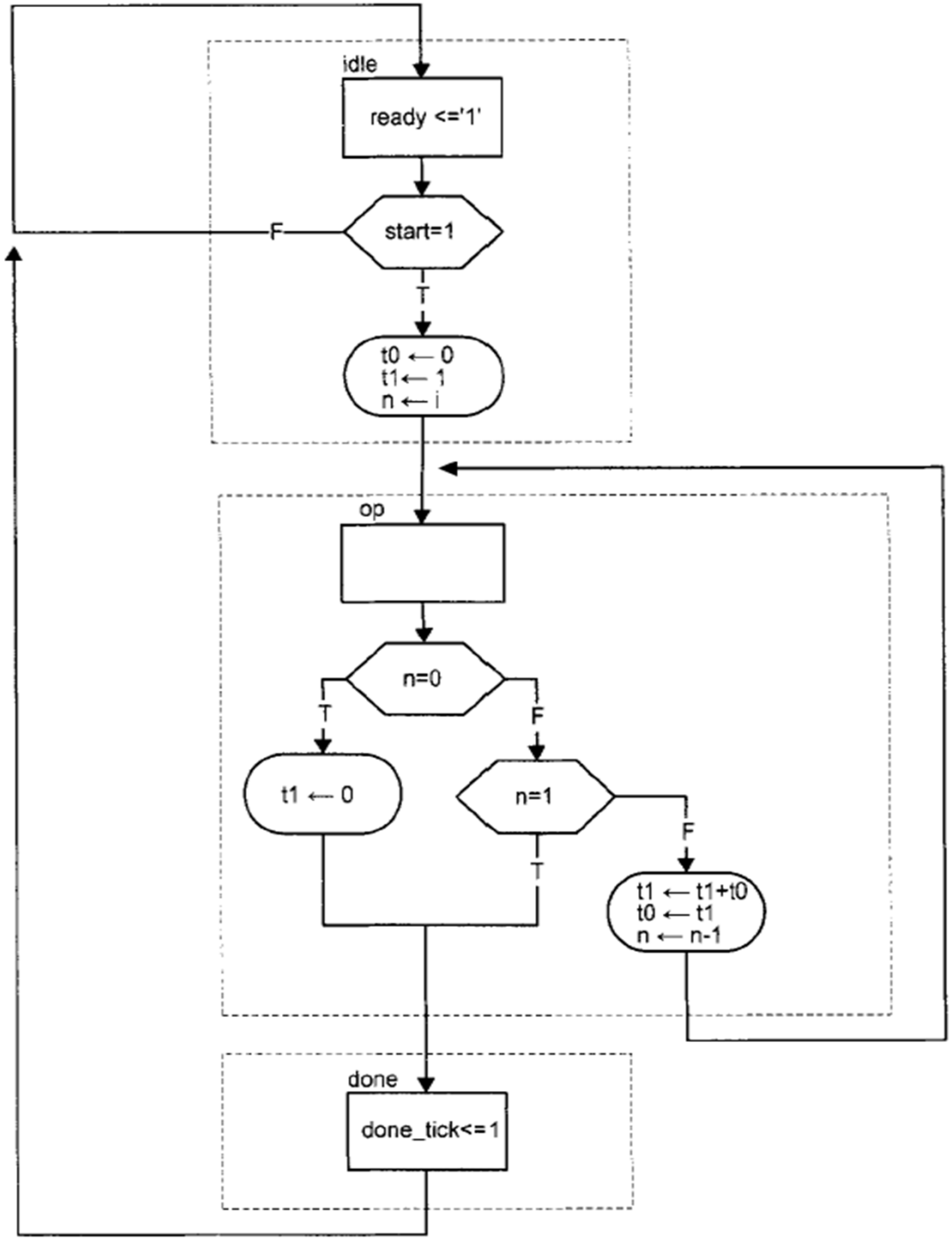
Example: Fibonacci number

- Fibonacci number circuit
- A sequence of integers

$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

ASMD

- tl and t0 are temporary storage registers
- n is the index register
- Input:
  - i: the number of required iterations
  - Start: Commands the beginning of operation
- Output:
  - F: the last generated number
  - Ready: indicates that the circuit is idle and ready to take new input
  - done-tick: asserted for one clock cycle when the operation is completed



VHDL code

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

entity fib is
    port(
        clk, reset: in std_logic;
        start: in std_logic;
        i: in std_logic_vector(4 downto 0);
        ready, done_tick: out std_logic;
        f: out std_logic_vector(19 downto 0)
    );
end fib;

architecture arch of fib is
   type state_type is (idle,op,done);
   signal state_reg, state_next: state_type;
   signal t0_reg, t0_next, t1_reg, t1_next: unsigned(19 downto 0);
   signal n_reg, n_next: unsigned(4 downto 0);

begin
```

VHDL code

- We use a 3-process implementation, much like the FSM implementation in previous lecture
  - but we should also handle the datapath registers in the next state calculation and assignment processes

Process 1: State and datapath register assignment

```
-- fsmd state and data registers
process(clk,reset)
begin
   if reset='1' then
      state_reg <= idle;
      t0_reg <= (others=>'0');
      t1_reg <= (others=>'0');
      n_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      t0_reg <= t0_next;
      t1_reg <= t1_next;
      n_reg <= n_next;
   end if;
end process;
```

```
-- fsmd next-state logic
  process(state_reg,n_reg,t0_reg,t1_reg,start,i,n_next)
  begin
     ready <='0';
     done_tick <= '0';
     state_next <= state_reg;
     t0_next <= t0_reg;
     t1_next <= t1_reg;
     n_next <= n_reg;
     case state_reg is
        when idle =>
           ready <= '1';
           if start='1' then
              t0_next <= (others=>'0');
              t1_next <= (0=>'1', others=>'0');
              n_next <= unsigned(i);
              state_next <= op;
           end if;
        when op =>
           if n_reg=0 then
              t1_next <= (others=>'0');
              state_next <= done;
           elsif n_reg=1 then
              state_next <= done;
           else
              t1_next <= t1_reg + t0_reg;
              t0_next <= t1_reg;
              n_next <= n_reg - 1;
           end if;
        when done =>
           done_tick <= '1';
           state_next <= idle;
     end case;
  end process;
```

Process 2: next state and datapath register

```
-- output
f <= std_logic_vector(t1_reg);
```

Process 3: output generation, we actually use a dataflow assignment, as the output is very simple