

CDA 4253/CIS 6930 FPGA System Design

Finite State Machines

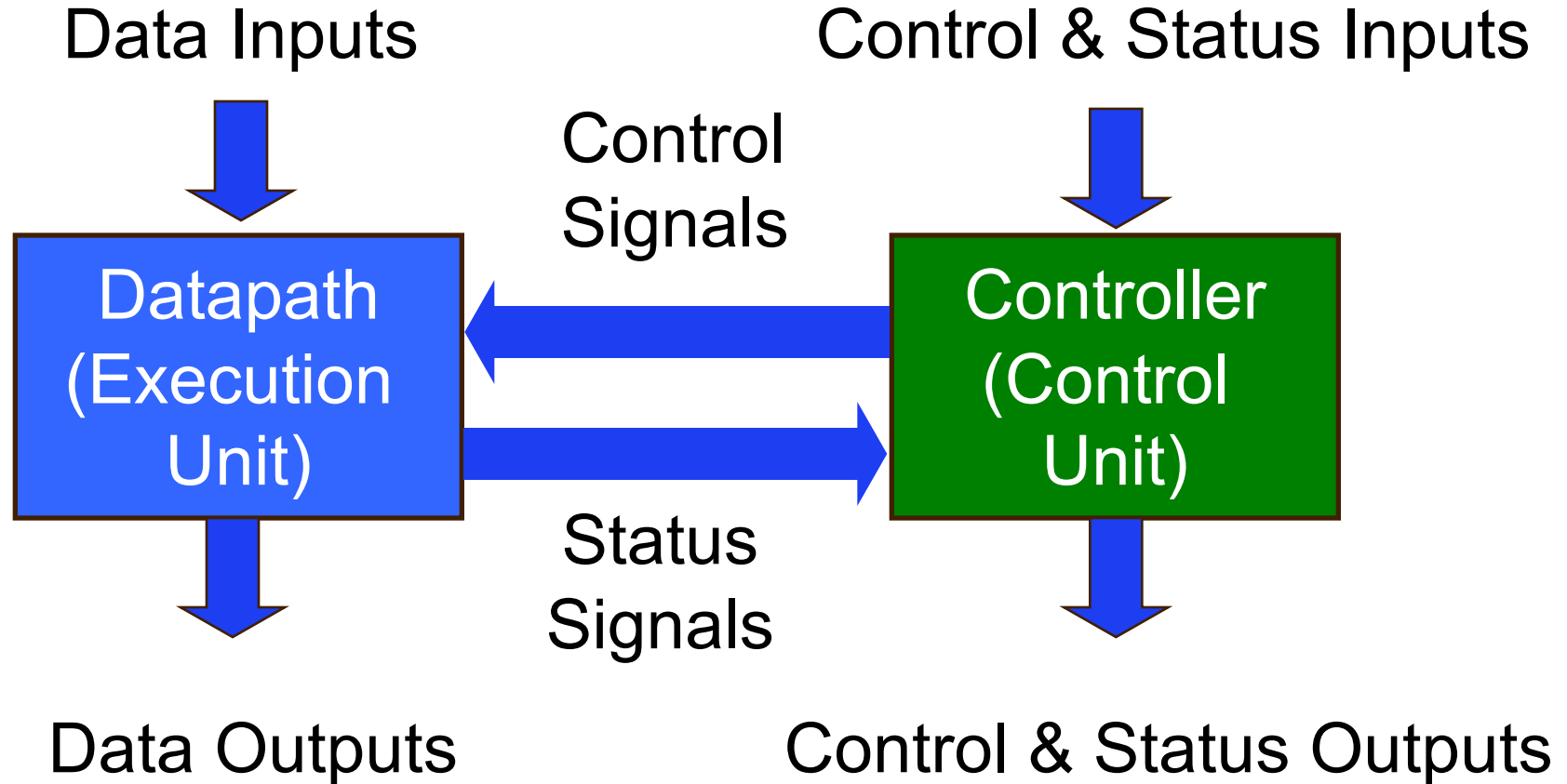
Dr. Hao Zheng
Comp Sci & Eng
U of South Florida

Outline and Reading

- Modeling FSMs in VHDL
 - Mealy and Moore
- Modeling FSMD in VHDL
 - Map computation into FSMD
- **Reading** – P. Chu, *FPGA Prototyping by VHDL Examples*
 - Chapter 5, FSM (skip discussion on ASM)
 - Chapter 6, FSMD

Datapath vs. Controller

Structure of a Typical Digital System



Datapath (Execution Unit)

- Manipulates and processes data.
- Performs arithmetic and logic operations, shifting/rotating, and other data-processing tasks.
- Is composed of registers, multiplexers, adders, decoders, comparators, ALUs, gates, etc.
- Provides all necessary resources and interconnects among them to perform specified task.
- Interprets control signals from the **controller** and generates status signals for the **controller**.

Controller (Control Unit)

- Controls data movement in the **datapath** by switching multiplexers and enabling or disabling resources
 - Example: enable signals for registers
 - Example: select signals for muxes
- Provides signals to activate various processing tasks in the **datapath**, *i.e.* $+$, $-$, or $*$, ...
- Determines the sequence of operations performed by the **datapath**.
- Follows some ‘program’ or schedule.

Programmable vs. Non-Programmable Controller

- Controller can be programmable or non-programmable
- *Programmable*
 - Has a program counter which points to next instruction
 - Instructions are stored in a RAM or ROM
 - Microprocessor is an example of programmable controller
- *Non-Programmable*
 - Once designed, implements the same functionality
 - Another term is a “hardwired state machine,” or “hardwired FSM,” or “hardwired instructions”
 - **In this course we will be focusing on non-programmable controllers.**

Finite State Machines

- Controllers can be described as Finite State Machines (FSMs)
 - Counters and shift registers are simple FSMs
- Finite State Machines can be represented using
 - **State Diagrams and State Tables** - suitable for simple controllers with a relatively few inputs and outputs
 - **Algorithmic State Machine (ASM) Charts**
Will be skipped as it is equivalent to state diagrams.
- All of these descriptions can be easily translated to the corresponding synthesizable VHDL code

Design Process

1. Text description
2. Define interface
3. Describe the functionality using pseudo-code
4. Convert pseudo-code to FSM in state diagram
 1. Define states and state transitions
 2. Define datapath operations in each state.
5. Develop VHDL code to implement FSM
6. Develop testbench for simulation and debugging
7. Implementation and timing simulation
 - Timing simulation can reveal more bugs than pre-synthesis simulation
8. Test the implementation on FPGA boards

Finite State Machines

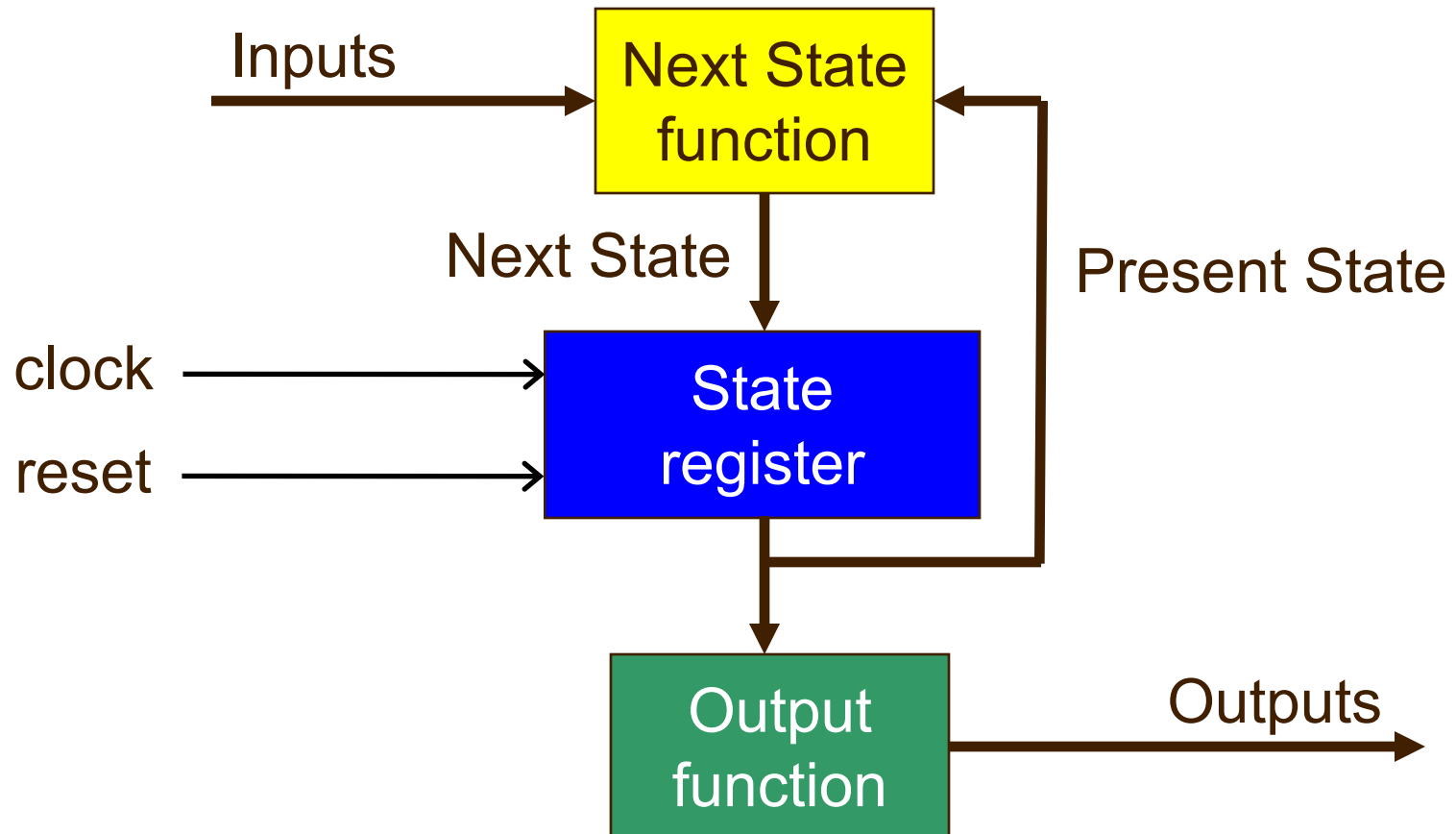
Refresher

Finite State Machines (FSMs)

- An FSM is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input.
- The main application of an FSM is to act as the controller to a large digital system
- Design of FSMs involves
 - Define states
 - Define state transitions
 - Define operations performed in each state
 - Optimize / minimize FSM
- Manual optimization/minimization is practical for small FSMs only.

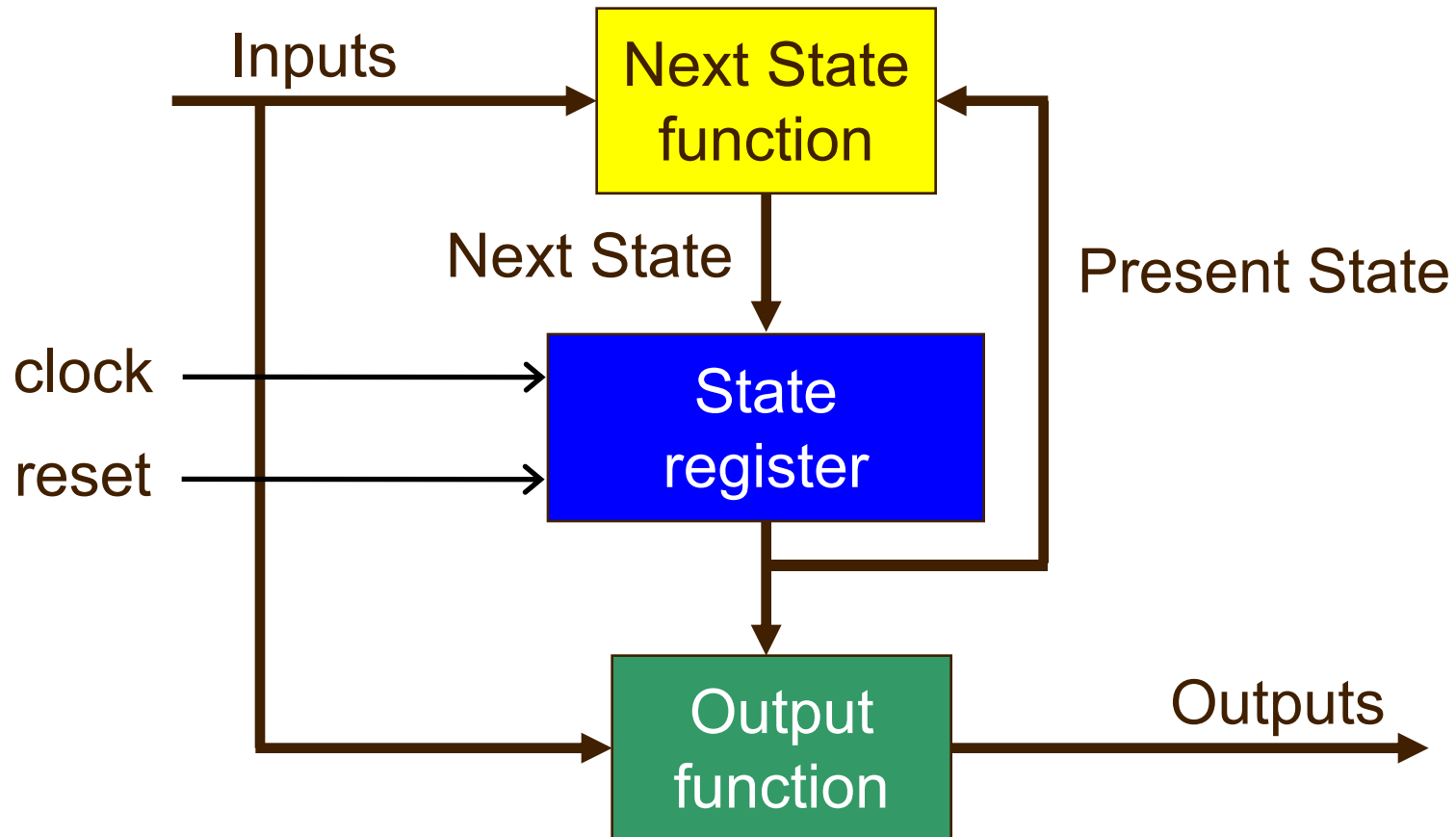
Moore FSM

→ Output is a function of the present state only



Mealy FSM

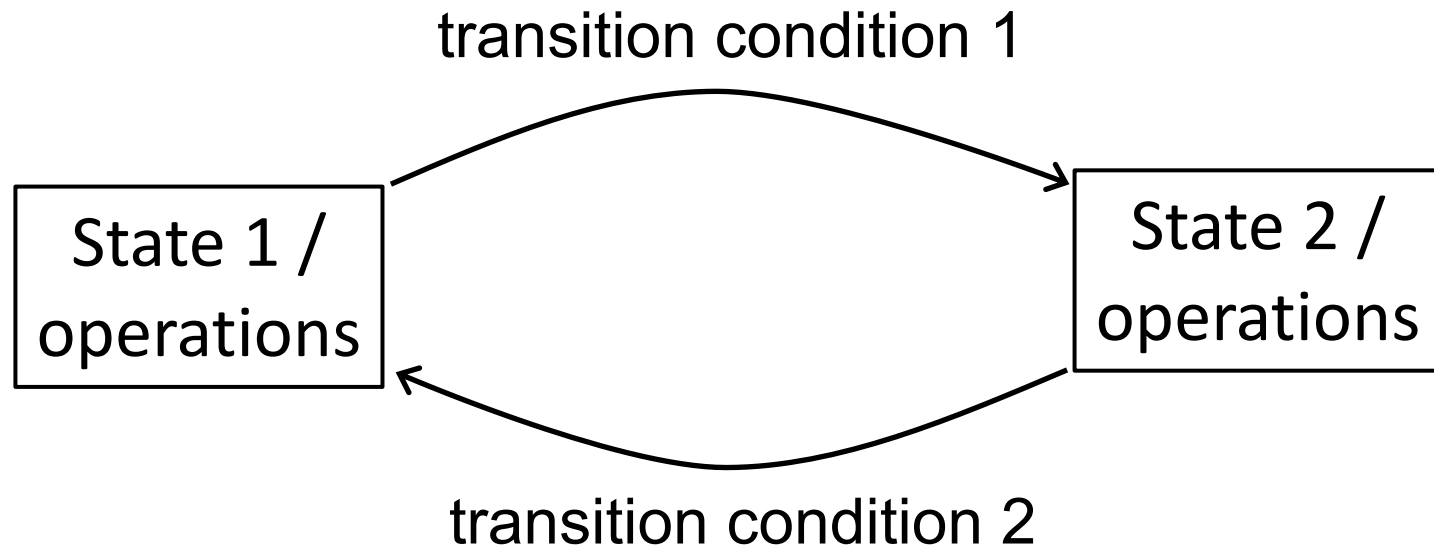
→ Output is a function of the present state and the inputs.



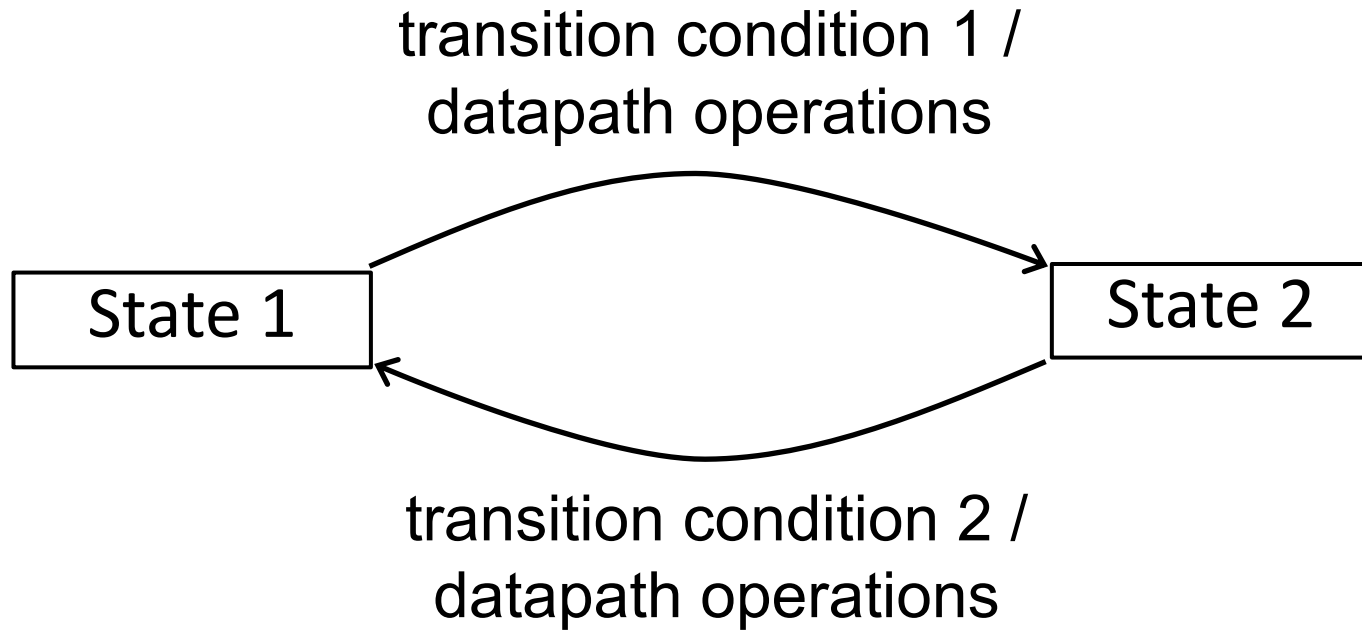
State Diagrams

Moore Machine

- State operations: datapath operations including output assignments.
- Transition conditions: Boolean expressions

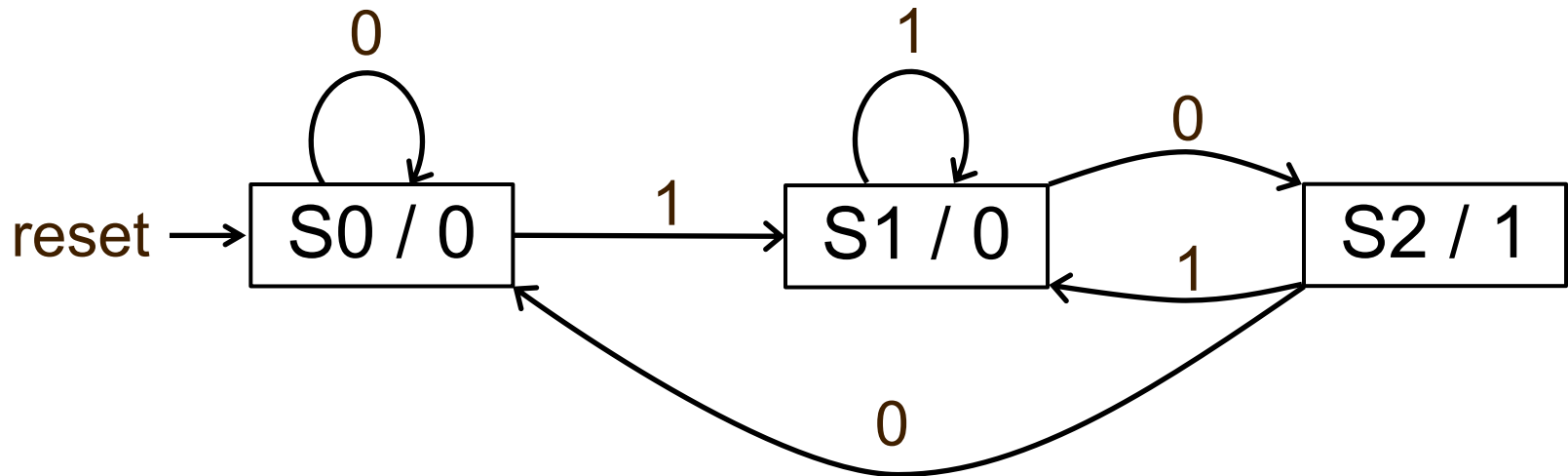


Mealy Machine



Moore FSM – Example 1

→ Moore FSM that recognizes sequence “10”



Meaning
of states:

S0: No
elements
of the
sequence
observed

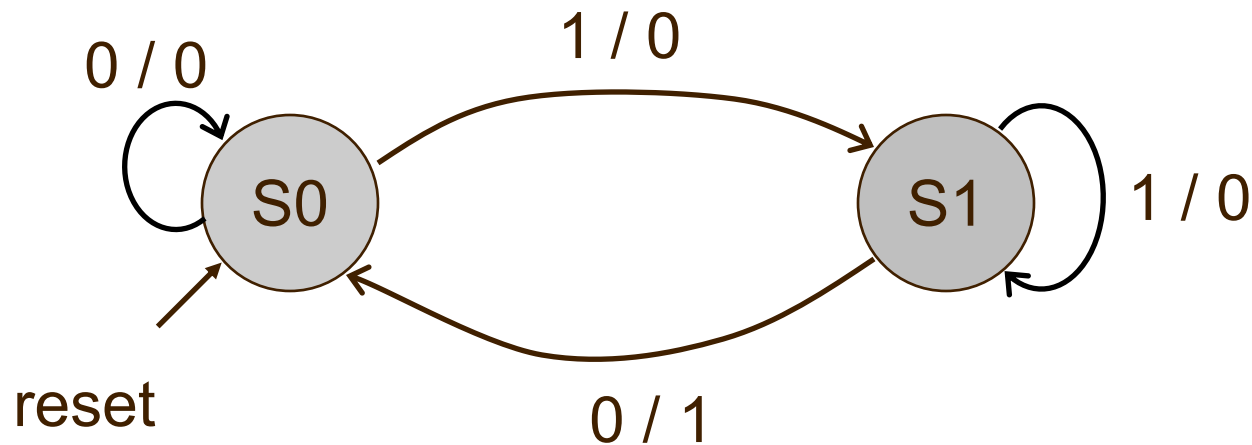
S1: “1”
observed

S2: “10”
observed

Ex: 0100011101010

Mealy FSM – Example 1

→ Mealy FSM that recognizes sequence “10”



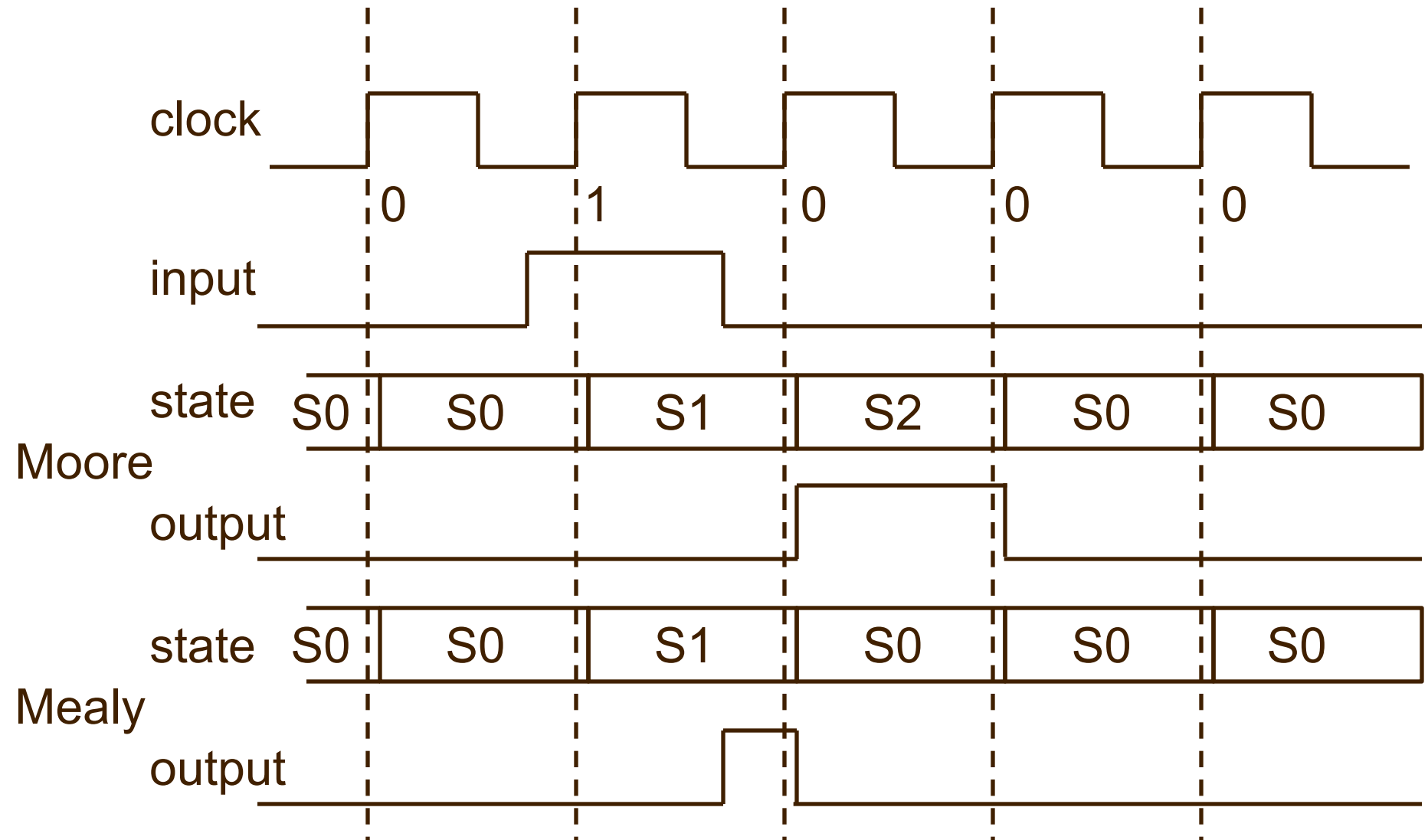
Meaning
of states:

S0: No
elements
of the
sequence
observed

S1: “1”
observed

Ex: 0100011101010

Moore & Mealy FSMs – Example 1



Moore vs. Mealy FSM (1)

- Moore and Mealy FSMs are functionally equivalent.
 - Equivalent Mealy FSM can be derived from Moore FSM and vice versa.
- Mealy FSM usually requires less number of states
 - Smaller circuit area.

Moore vs. Mealy FSM (2)

- Mealy FSM computes outputs as soon as inputs change.
 - Mealy FSM responds to inputs one clock cycle sooner than equivalent Moore FSM.
 - There are direct paths from inputs to outputs – can cause output glitches.
- Moore FSM has no combinational path between inputs and outputs.
 - Less likely to affect the critical path of the entire circuit.

Which Way to Go?

Mealy FSM

Fewer states

Lower Area

Responds one clock
cycle earlier

Moore FSM

Safer.
Less likely to affect
the critical path.

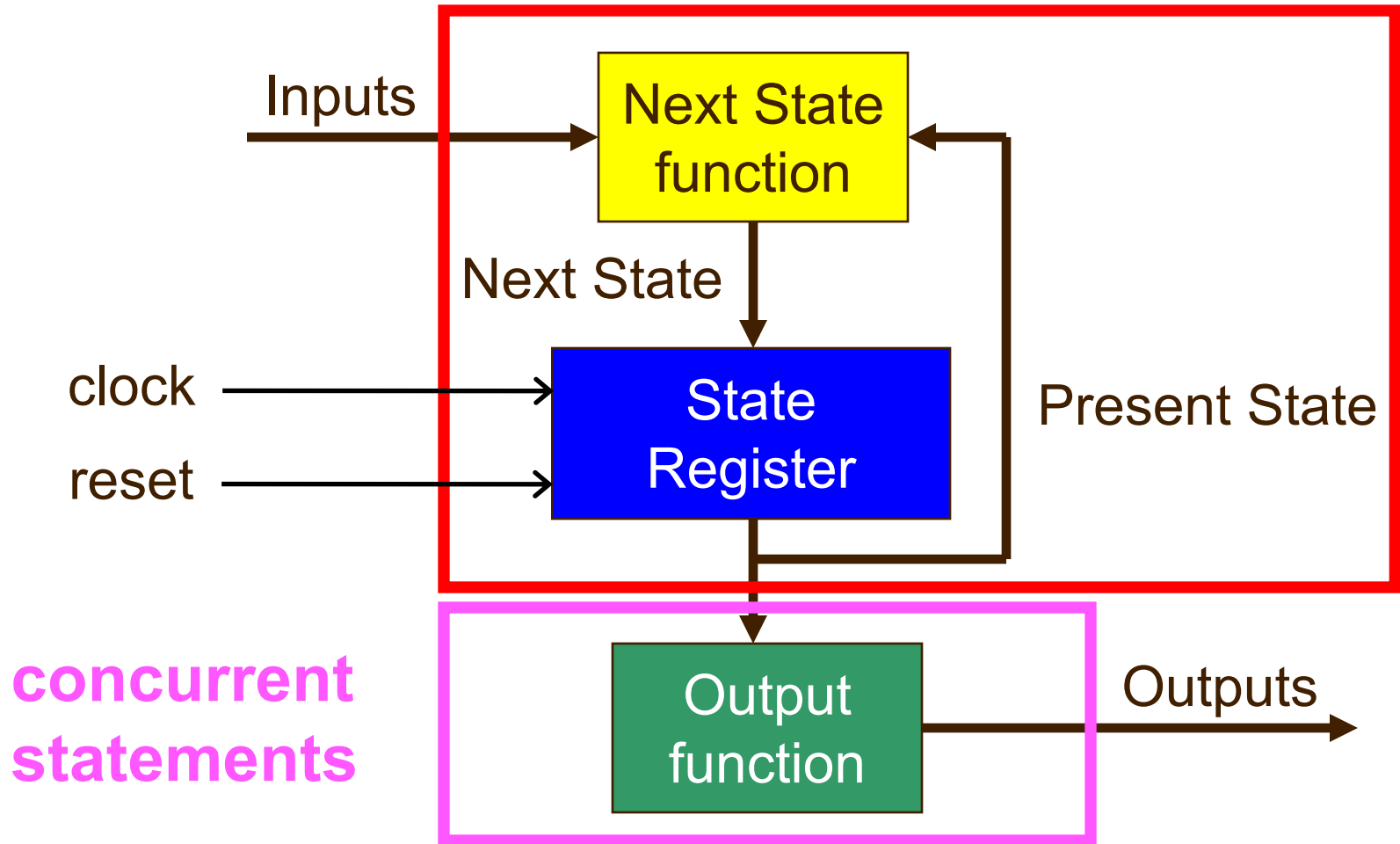
Finite State Machines in VHDL

FSMs in VHDL

- Finite State Machines can be easily described with processes.
- Synthesis tools understand FSM description if certain rules are followed.
 - State transitions should be described in a process sensitive to *clock* and *asynchronous reset* signals only.
 - Output function described using rules for combinational logic, i.e. as concurrent statements or a process with all inputs and state variables in the sensitivity list.

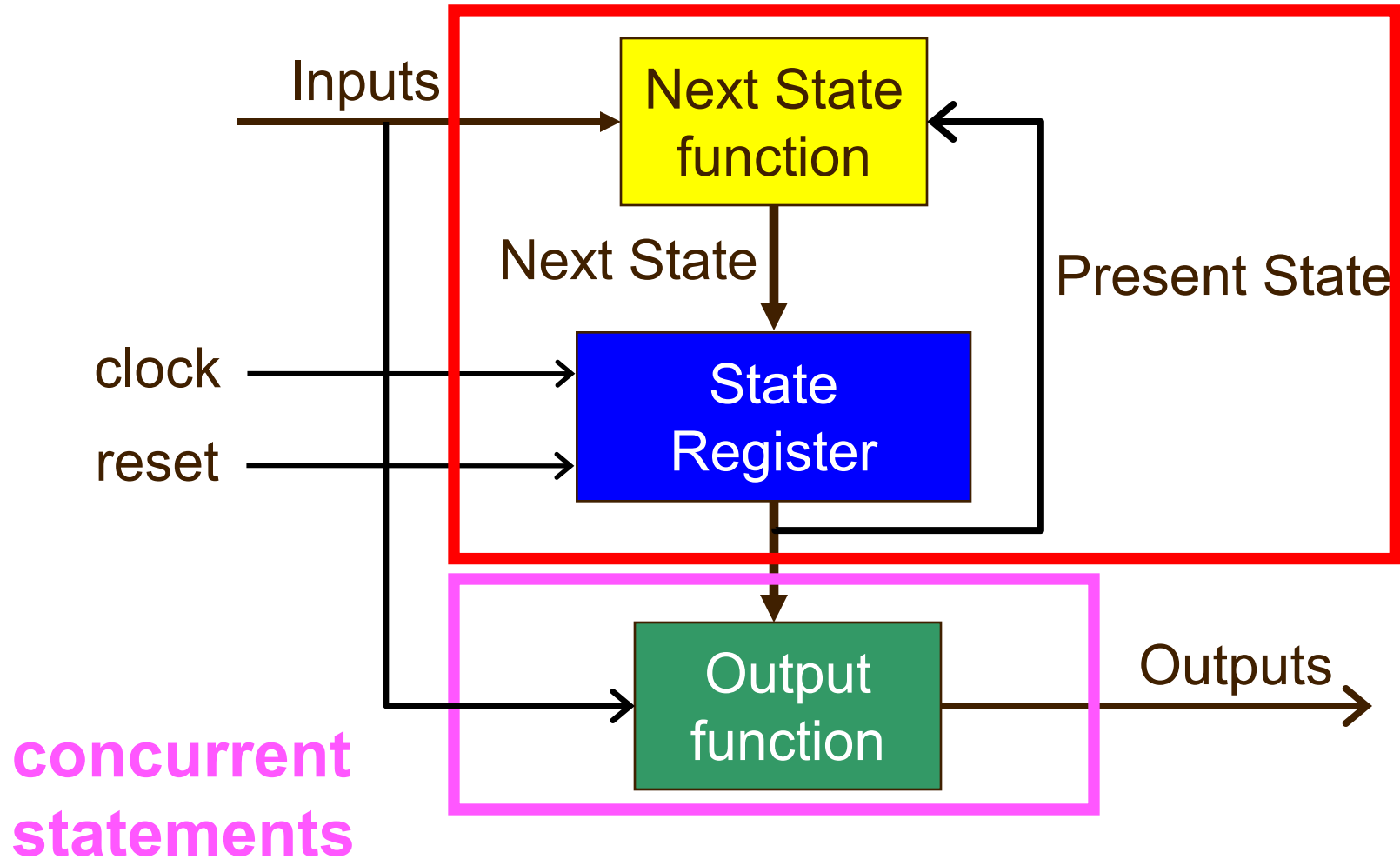
Moore FSM

process(clock, reset)



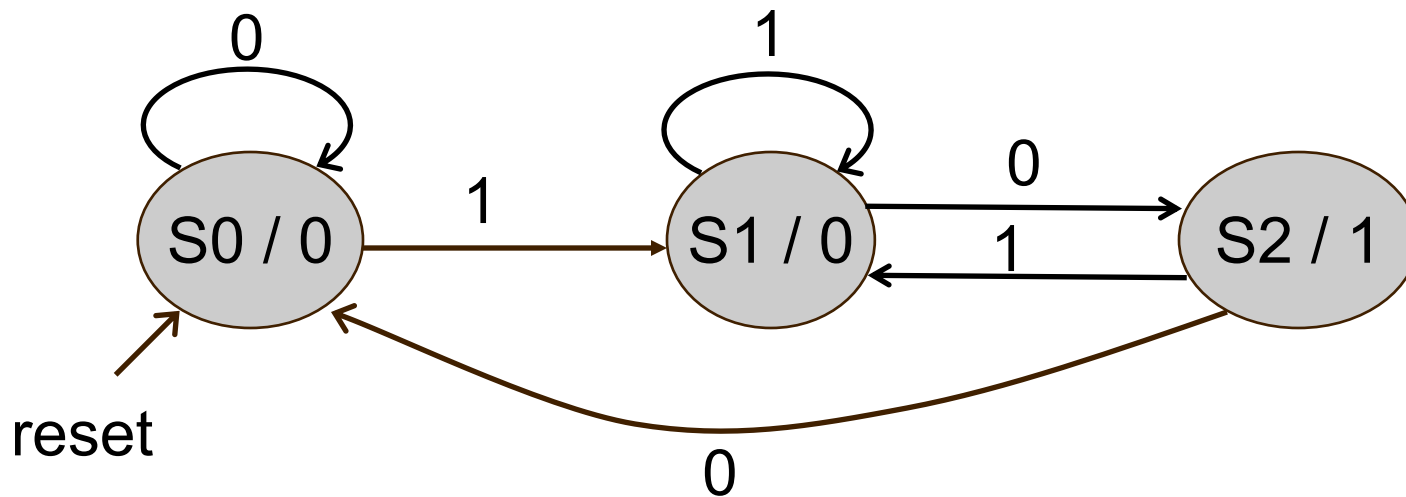
Mealy FSM

process(clock, reset)



Moore FSM - Example 1

→ Moore FSM that Recognizes Sequence “10”



Moore FSM in VHDL (1)

```
architecture ...
```

```
    type state_type is (s0, s1, s2); -- enumeration type
```

```
    signal state: state_type;
```

```
begin
```

```
    U_Moore: process(clock, reset)
```

```
    begin
```

```
        if (reset = '1') then
```

```
            state <= s0;
```

```
        elsif rising_edge(clock) then
```

```
            case state is
```

```
                when s0 =>
```

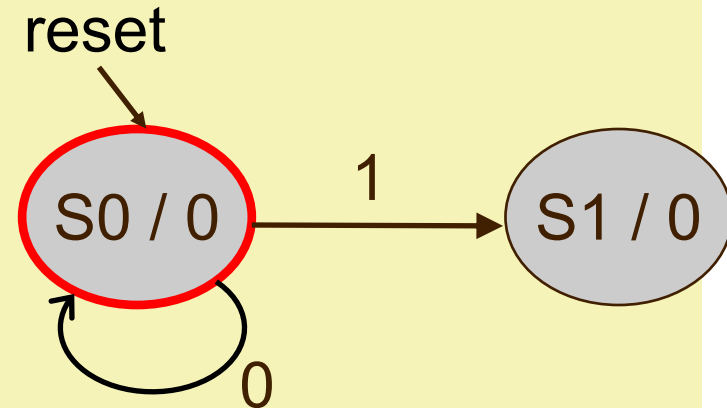
```
                    if input = '1' then
```

```
                        state <= s1;
```

```
                    else
```

```
                        state <= s0;
```

```
                    end if;
```

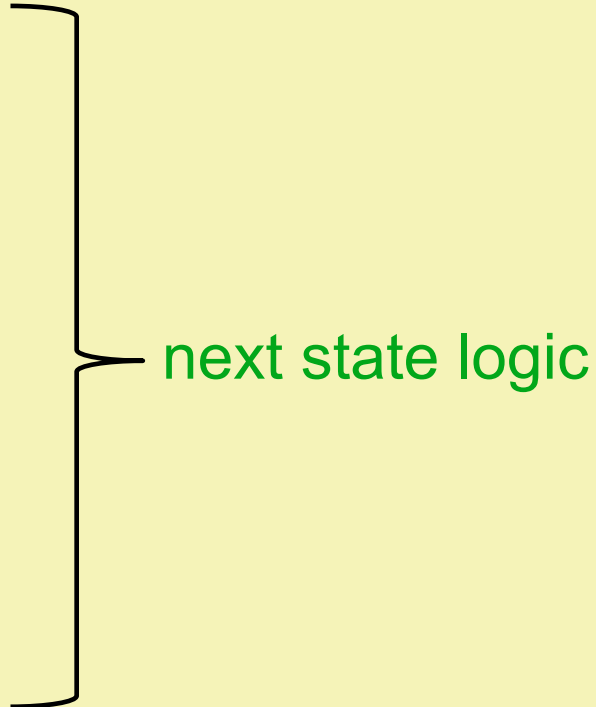


next state logic

Moore FSM in VHDL (2) – cont'd

```
    when S1 =>
        if input = '0' then
            state <= S2;
        else
            state <= S1;
        end if;
    when S2 =>
        if input = '0' then
            state <= S0;
        else
            state <= S1;
        end if;
    end case;
end if;
end process;

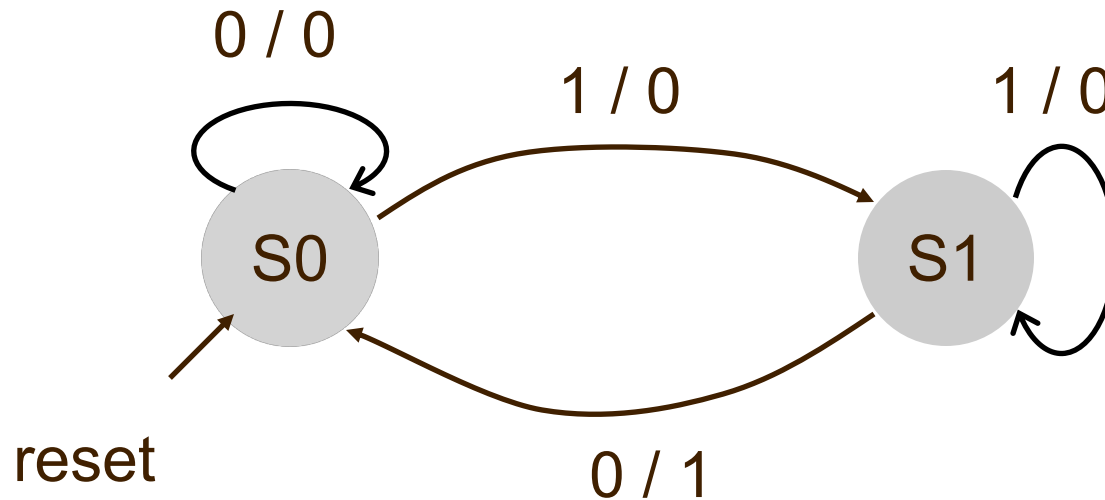
-- output function
Output <= '1' when state = S2 else
          '0';
```



next state logic

Mealy FSM - Example 1

→ Mealy FSM that Recognizes Sequence “10”.



Mealy FSM in VHDL (1)


```
architecture ...  
    type state_type is (S0, S1);  
    signal Mealy_state: state_type;  
begin  
    U_Mealy: process (clock, reset)  
    begin  
        if (reset = '1') then  
            Mealy_state <= S0;  
        elsif rising_edge(clock) then  
            case Mealy_state is  
                when S0 =>  
                    if input = '1' then  
                        Mealy_state <= S1;  
                    else  
                        Mealy_state <= S0;  
                    end if;  
            end case;  
        end if;  
    end process;  
end architecture;
```

next state logic

Mealy FSM in VHDL (2)

```
        when S1 =>
            if input = '0' then
                Mealy_state <= S0;
            else
                Mealy_state <= S1;
            end if;
        end case;
    end if;
end process;

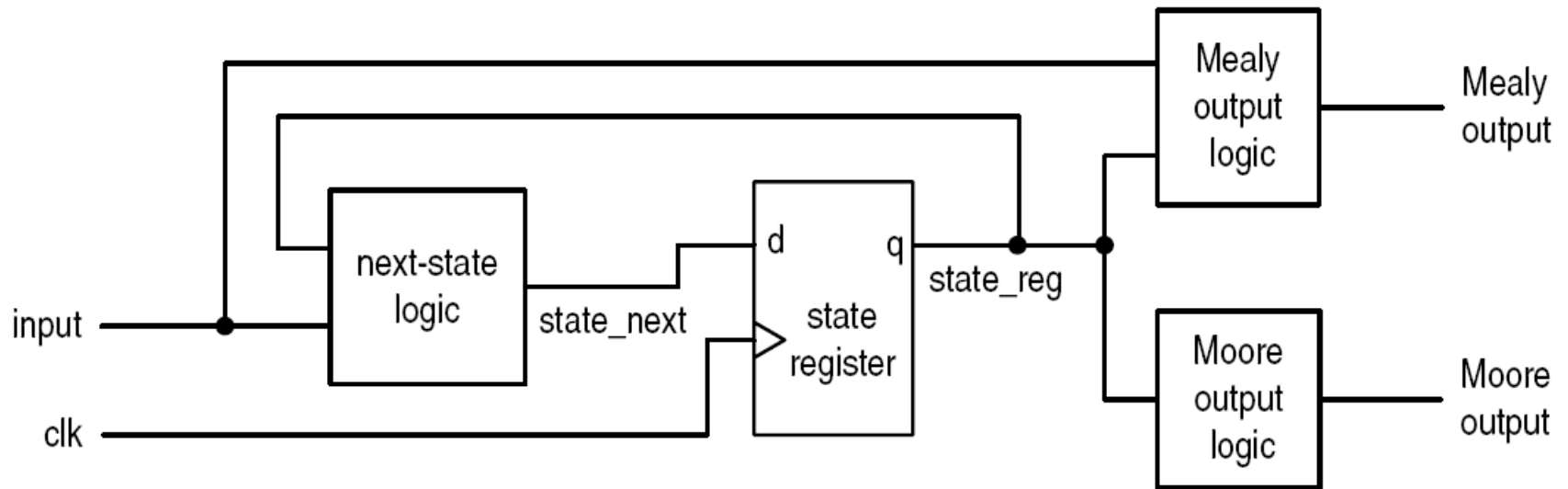
-- output function
Output <= '1' when (Mealy_state=S1 and input = '0') else
    '0';
end architecture;
```



next state logic

What would happen if output logic is merged with next state logic?

Generalized FSM

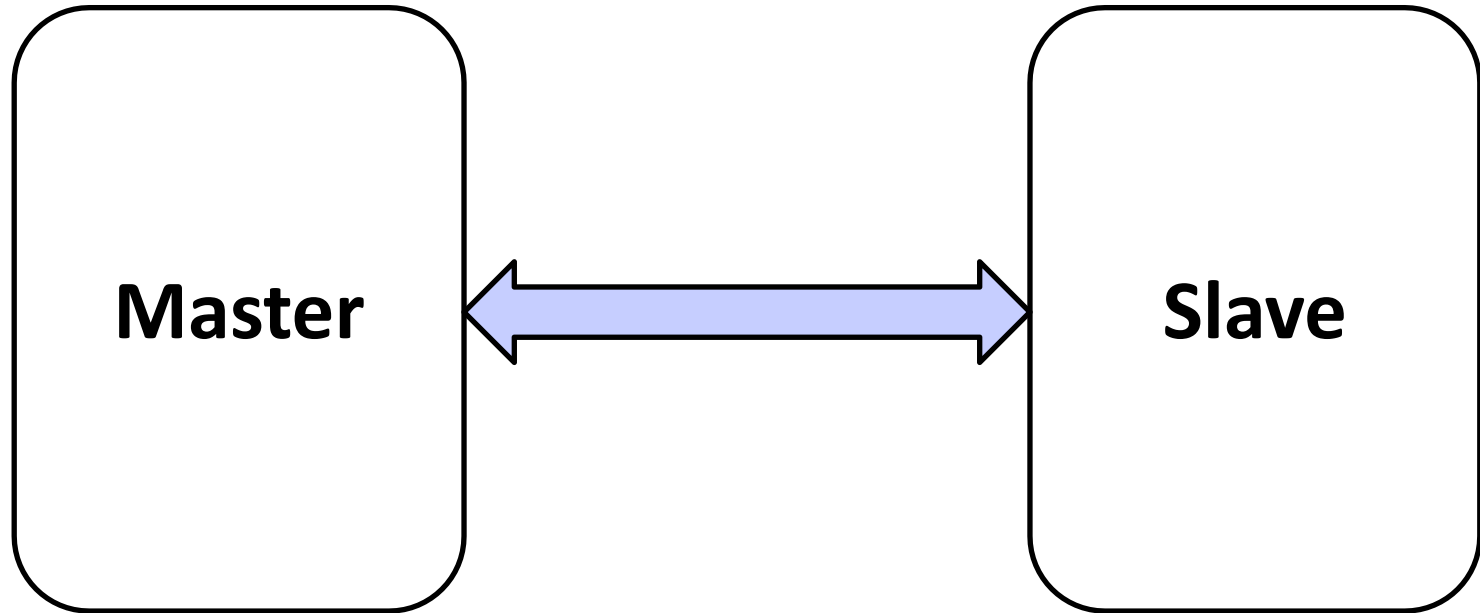


Based on RTL Hardware Design by P. Chu

Case Study 1

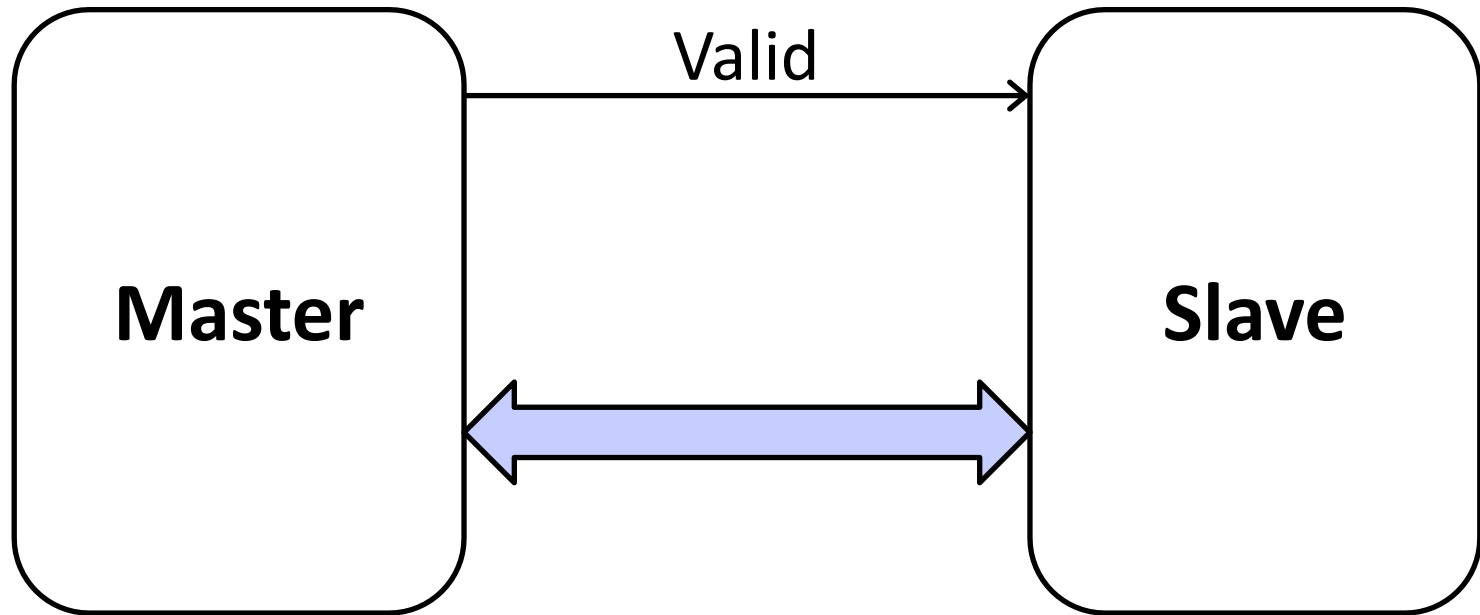
A Simple Communication Protocol

Inter-Component Communication



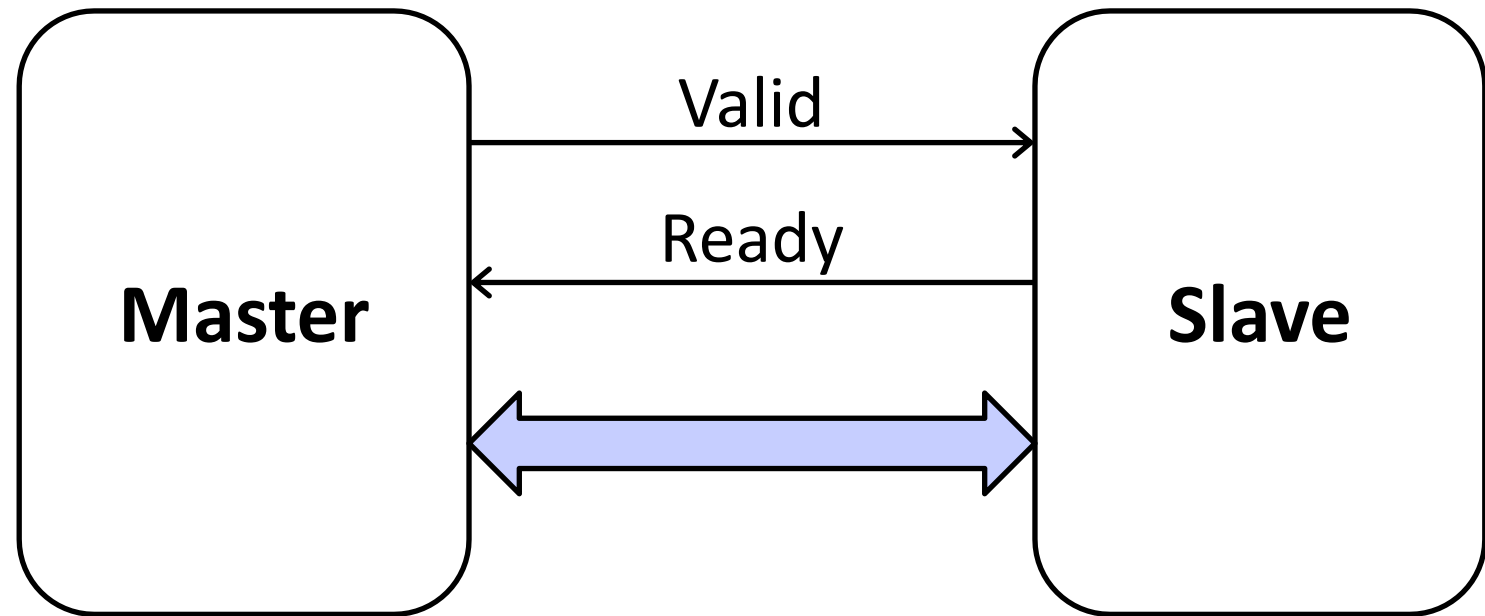
How does master transfer information to slave if they operate at different speeds?

Inter-Component Communication



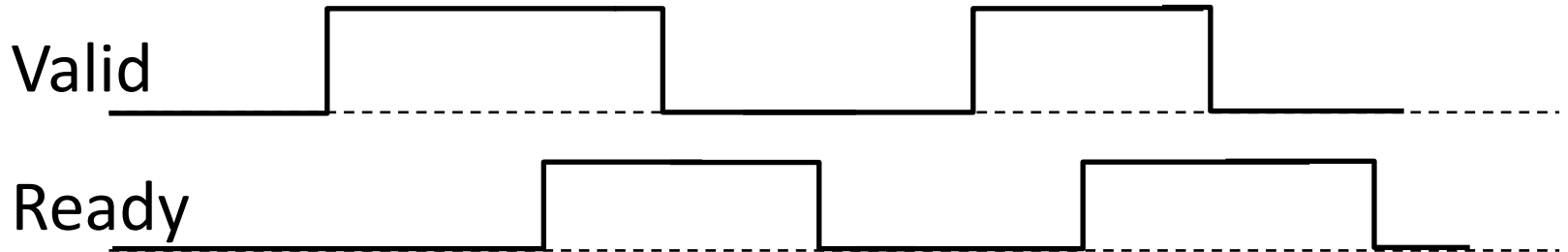
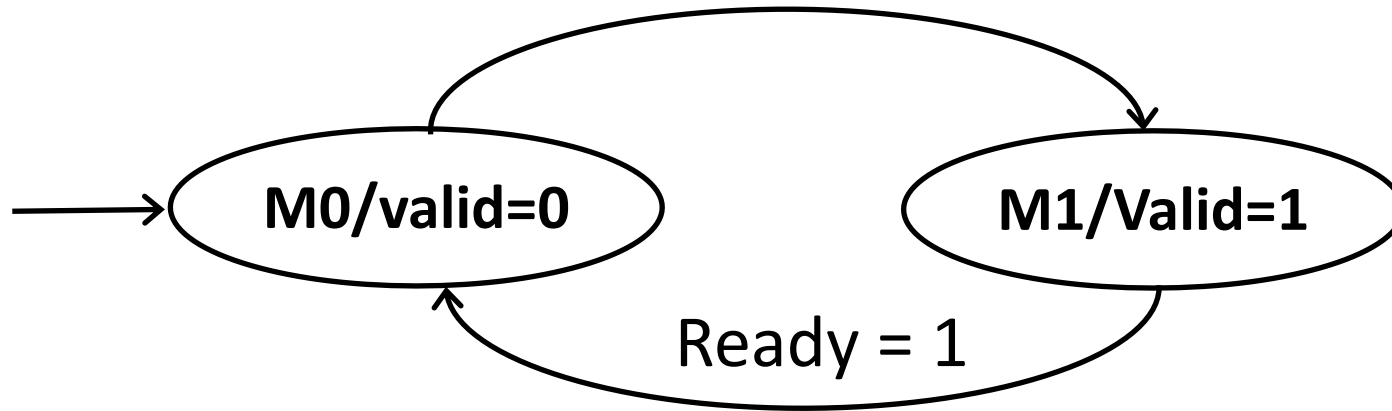
How does master transfer information to slave if they operate at different speeds?

Communication Protocol



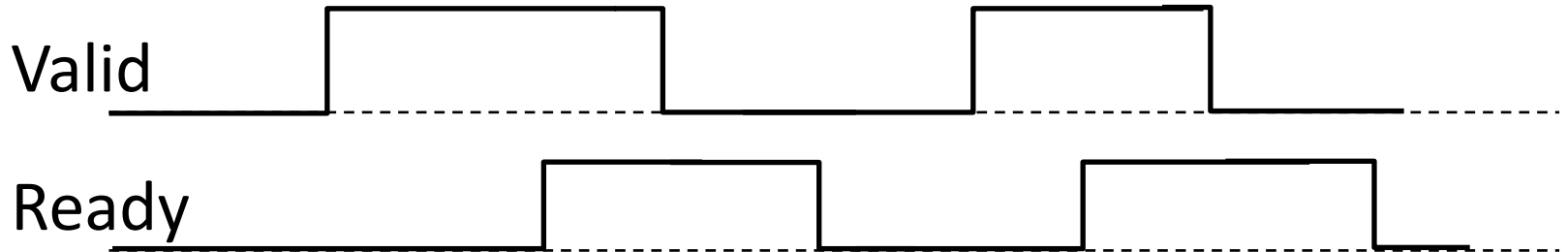
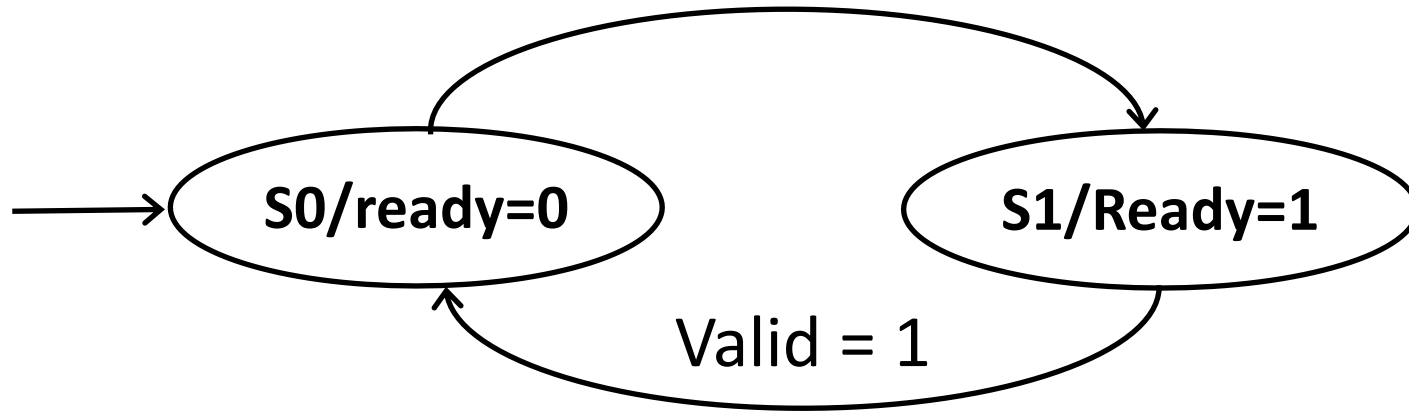
Communication Protocol – Master

Whenever data output is valid



Communication Protocol – Slave

When ready to accept data input



Case Study 2
Fibonacci Number
(section 6.3.1, Chu's book)

Fibonacci Number

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

ex. 0, 1, 1, 2, 3, 5, 8, 13, ...

Fibonacci Number – cont'd

start: start the operation

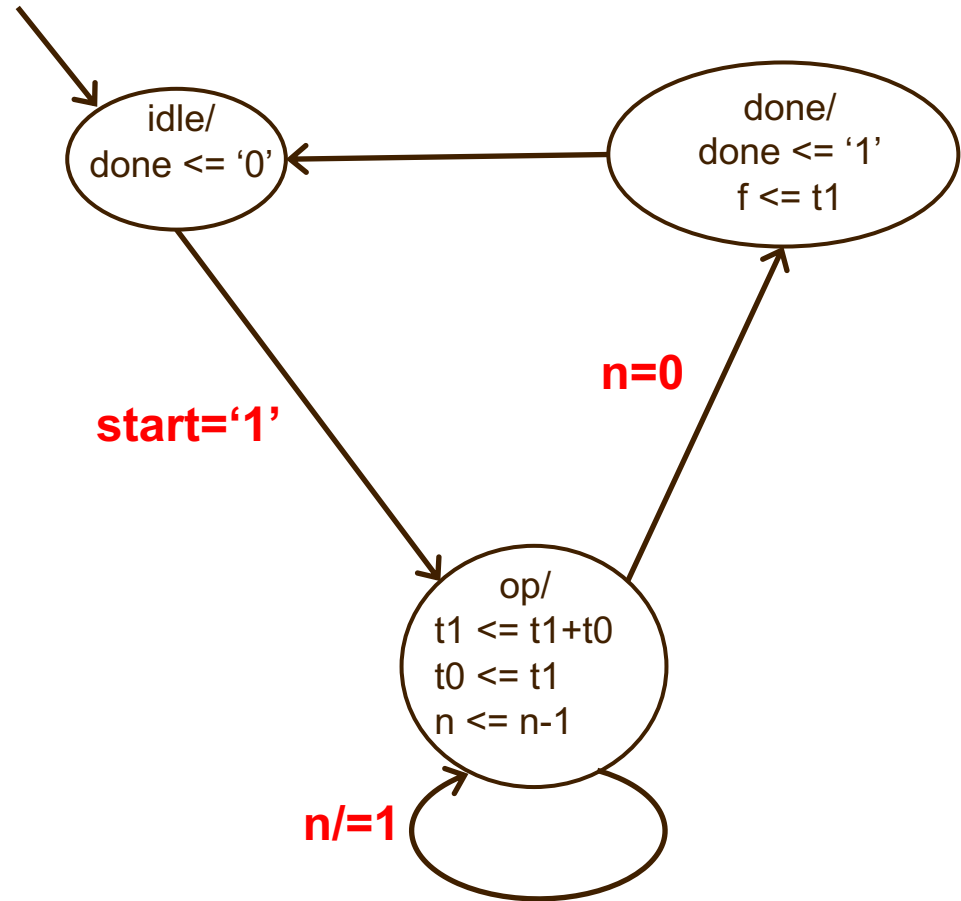
done: result is available

n: number of iterations

f: output

t0: register holding fib(i-2)

t1: register holding fib(i-1)



Case Study 3
Binary Division
(Section 6.3.2, Chu's Book)

Binary Division

$$\begin{array}{r} \text{divisor} \\ 0010 \overline{) 00001101} \quad \begin{array}{l} 00110 \text{ --- quotient} \\ 00001101 \text{ --- dividend} \end{array} \\ \underline{0000} \\ 0001 \\ \underline{0000} \\ 0011 \\ \underline{0010} \\ 0010 \\ \underline{0010} \\ 0001 \text{ --- remainder} \end{array}$$

Binary Division Algorithm

1. Double the dividend width by appending '0' to its left.
2. Align the divisor – double its width by appending '0' to its right.
3. If dividend \geq divisor, subtract divisor from dividend, and left shift '1' into quotient. Otherwise, left shift '0' into quotient.
4. Right shift divisor one position.
5. Repeat 3 and 4 until the remaining dividend is less than divisor.

Binary Division Algorithm

$$1101 / 0010 = ?$$

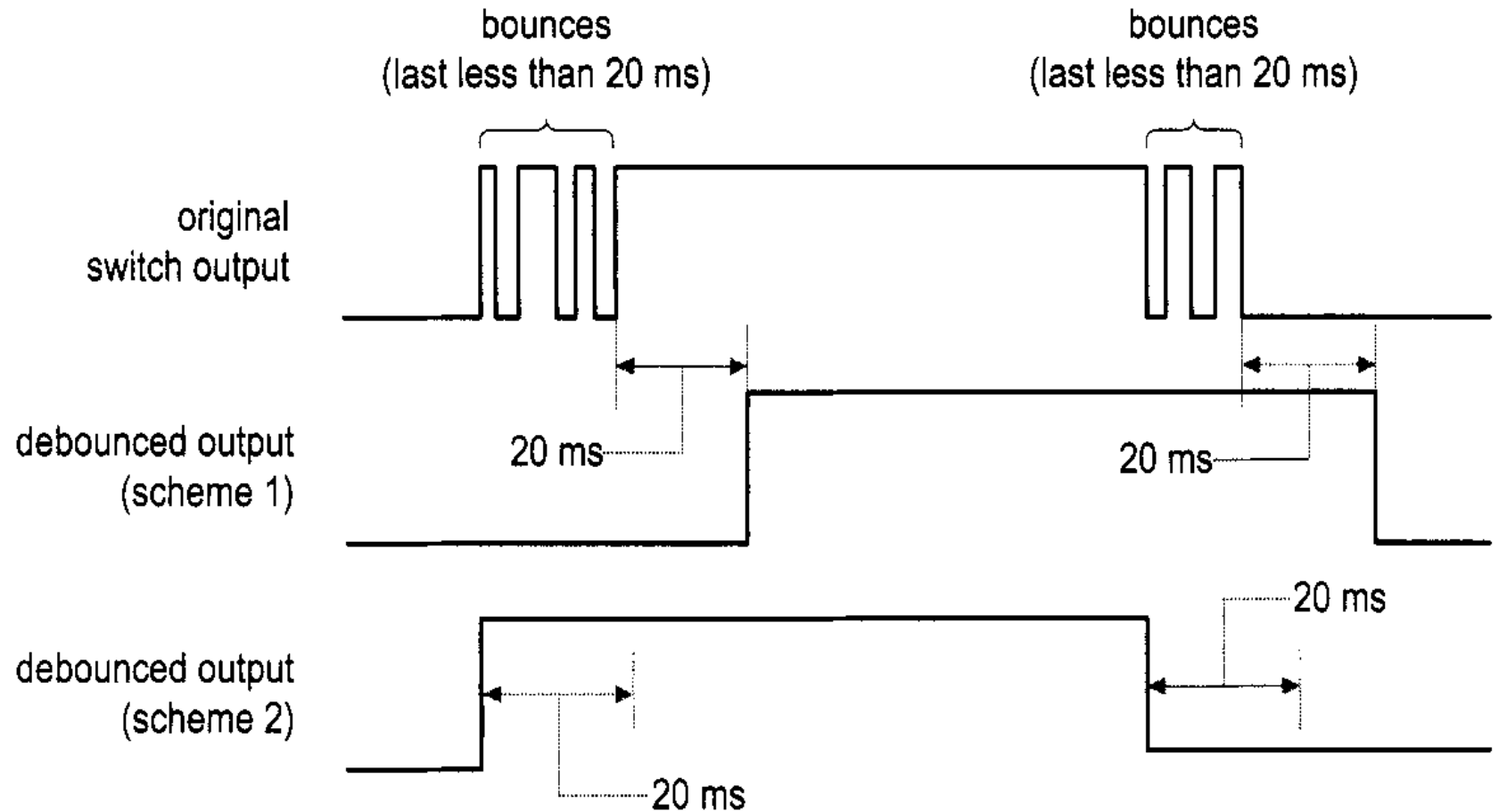
Dividend	Divisor	Quotient	
00001101	00100000		align divisor
00001101	00010000	0	right shift divisor
00001101	00001000	00	right shift divisor
00000101	00000100	001	dividend – divisor right shift divisor
00000001	00000010	0011	dividend – divisor right shift divisor
00000001	00000010	00110	dividend < divisor terminate

Binary Division Algorithm – FSM

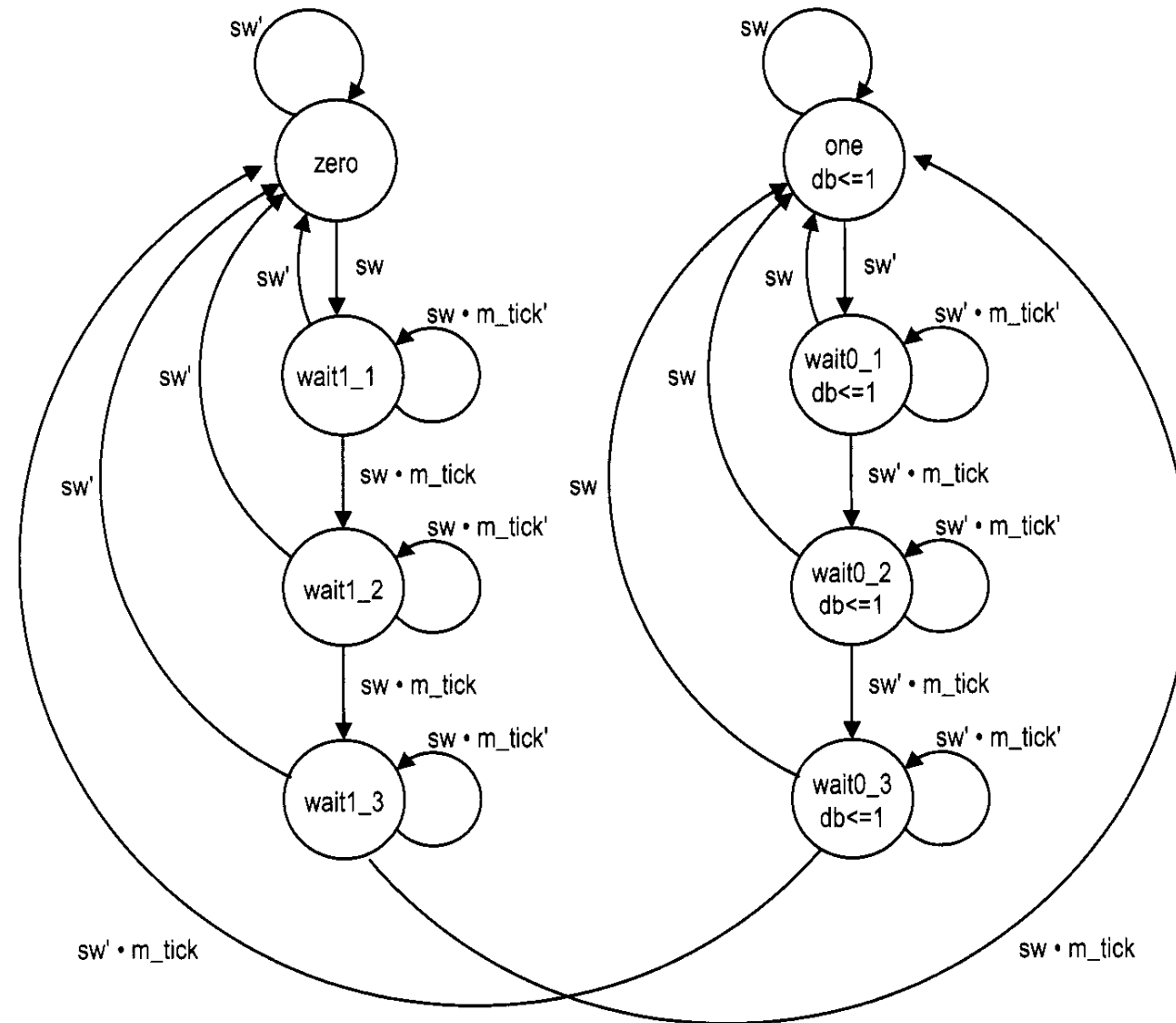
Case Study 4

Debouncing Circuit

Original & Debounced Inputs



Debouncing Circuit – Scheme 1

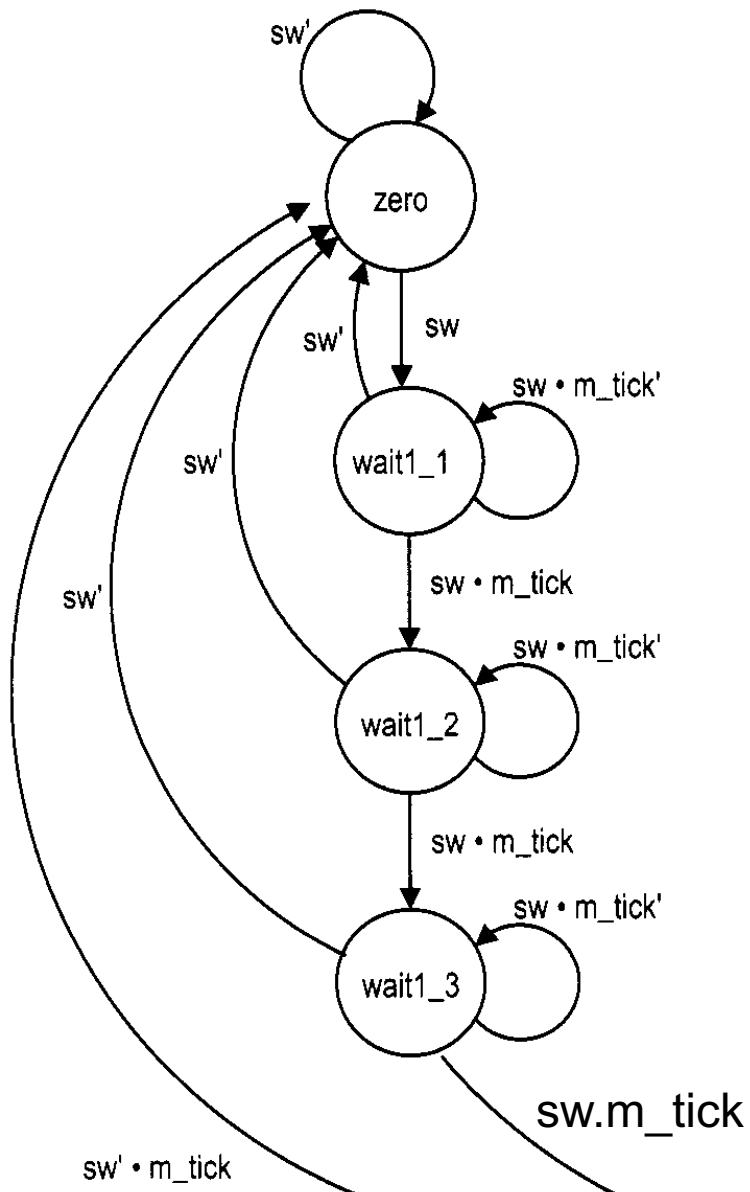


sw: input from slide switches or push buttons.

m_tick: input from a timer with 10ms period.

See listing 5.6 for VHDL code that implements this FSM

Debouncing Circuit – Scheme 1

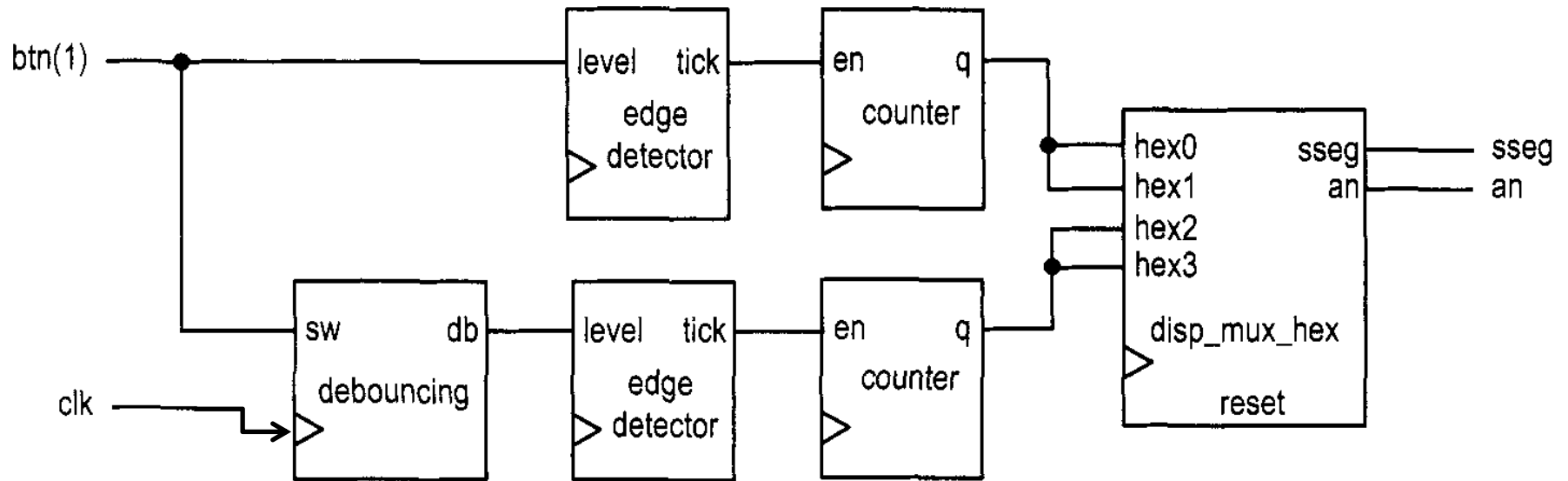


sw: input from slide switches or push buttons.

m_tick: input from a timer with 10ms period.

See listing 5.6 for VHDL code that implements this FSM

Debouncing Testing Circuit



Debouncing Circuit – Exercise

Re-design the debouncer using a 20ms timer

sw: input from slide switches or push buttons.

tick_20ms: input from a timer with 20ms period.

timer can be controller by **sw** input.

Loop Statements

For-Loop Statements

– – Count number of '0' in the input

```
library ieee;
```

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

```
use ieee.std_logic_unsigned.all;
```

```
entity countzeros is
```

```
    port(a : in std_logic_vector(7 downto 0);
```

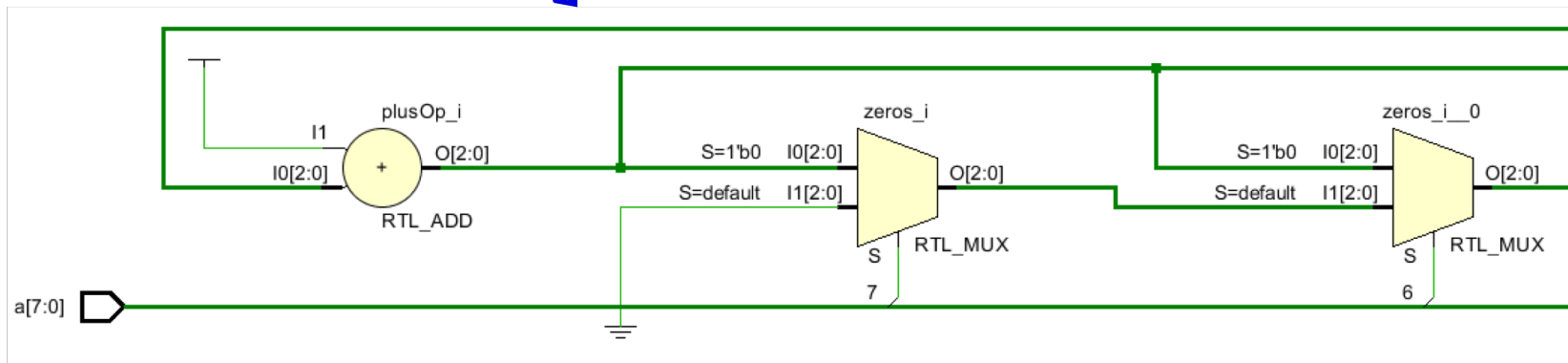
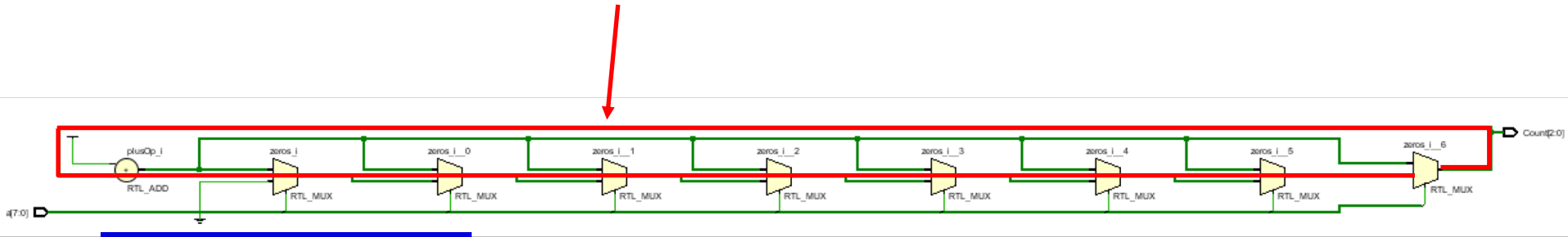
```
          Count : out std_logic_vector(2 downto 0)  
    );
```

```
end countzeros;
```

architecture behavior of countzeros is

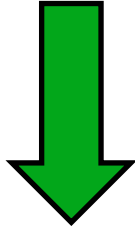
```
    signal zeros: std_logic_vector(2 downto 0);
begin
    process (a, zeros)
    begin
        zeros <= "000";
        for i in 7 downto 0 loop    -- bounds must
            if (a(i) = '0') then    -- be constants
                zeros <= zeros + 1;
            end if;
        end loop;
        Count <= zeros;
    end process;
end behavior;
```


Combinational loop



Another Example

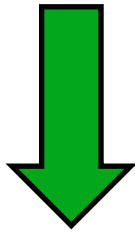
```
shreg <= shreg (6 downto 0) & SI;
```



```
for i in 0 to 6 loop  
    shreg(i+1) <= shreg(i);  
end loop;  
shreg(0) <= SI;
```

Another Example

```
for i in 0 to 6 loop  
    shreg(i+1) <= shreg(i);  
end loop;
```



```
for i in 0 to 6 loop  
    shreg(1) <= shreg(1);  
    ...  
    shreg(7) <= shreg(7);  
end loop;
```

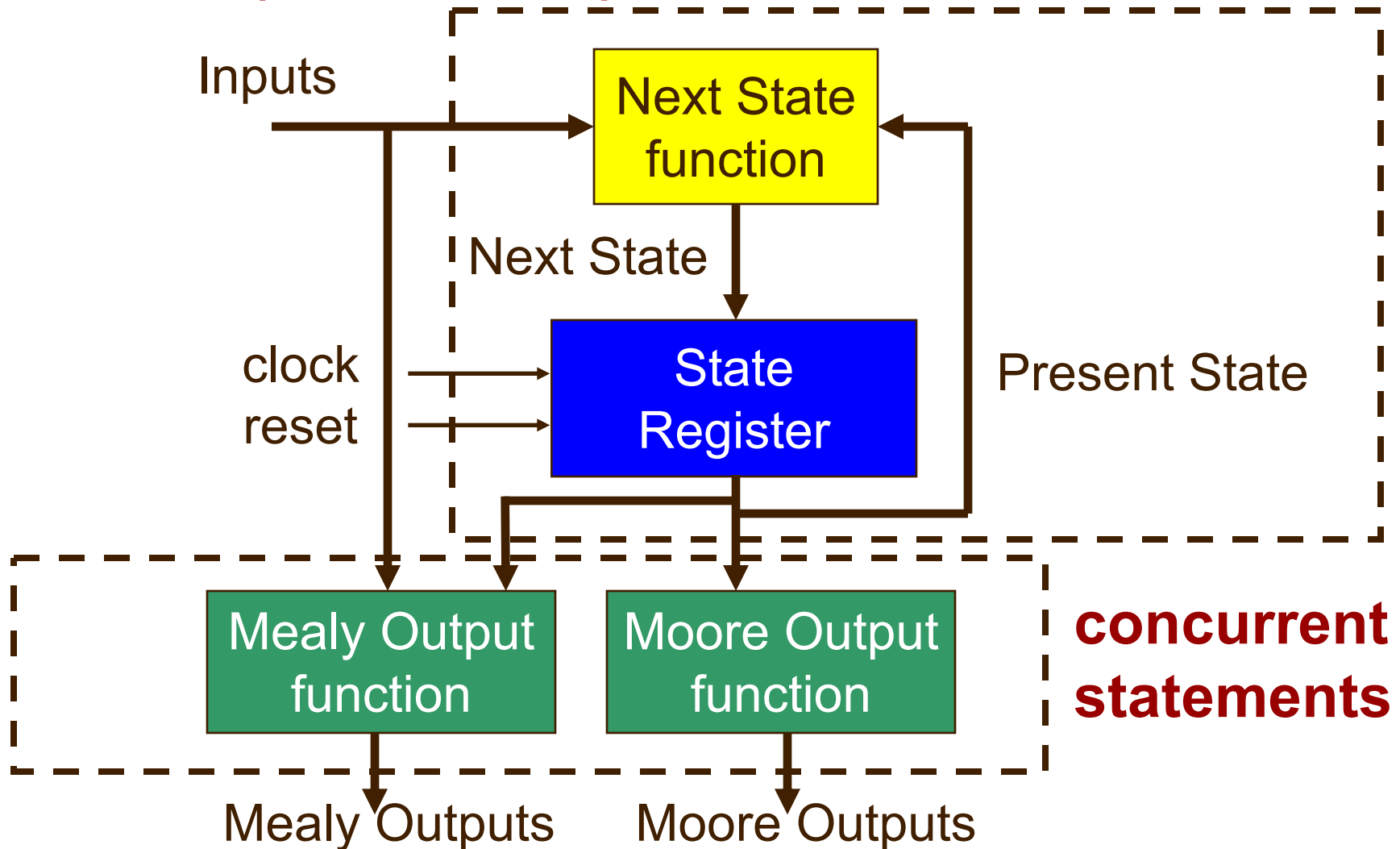
While Loop Statement

```
process (A)
  variable I : integer range 0 to 4;
begin
  Z <= "0000";
  I := 0;
  while (I <= 3) loop
    if (A = I) then
      Z(I) <= '1';
    end if;
    I := I + 1;
  end loop;
end process;
```

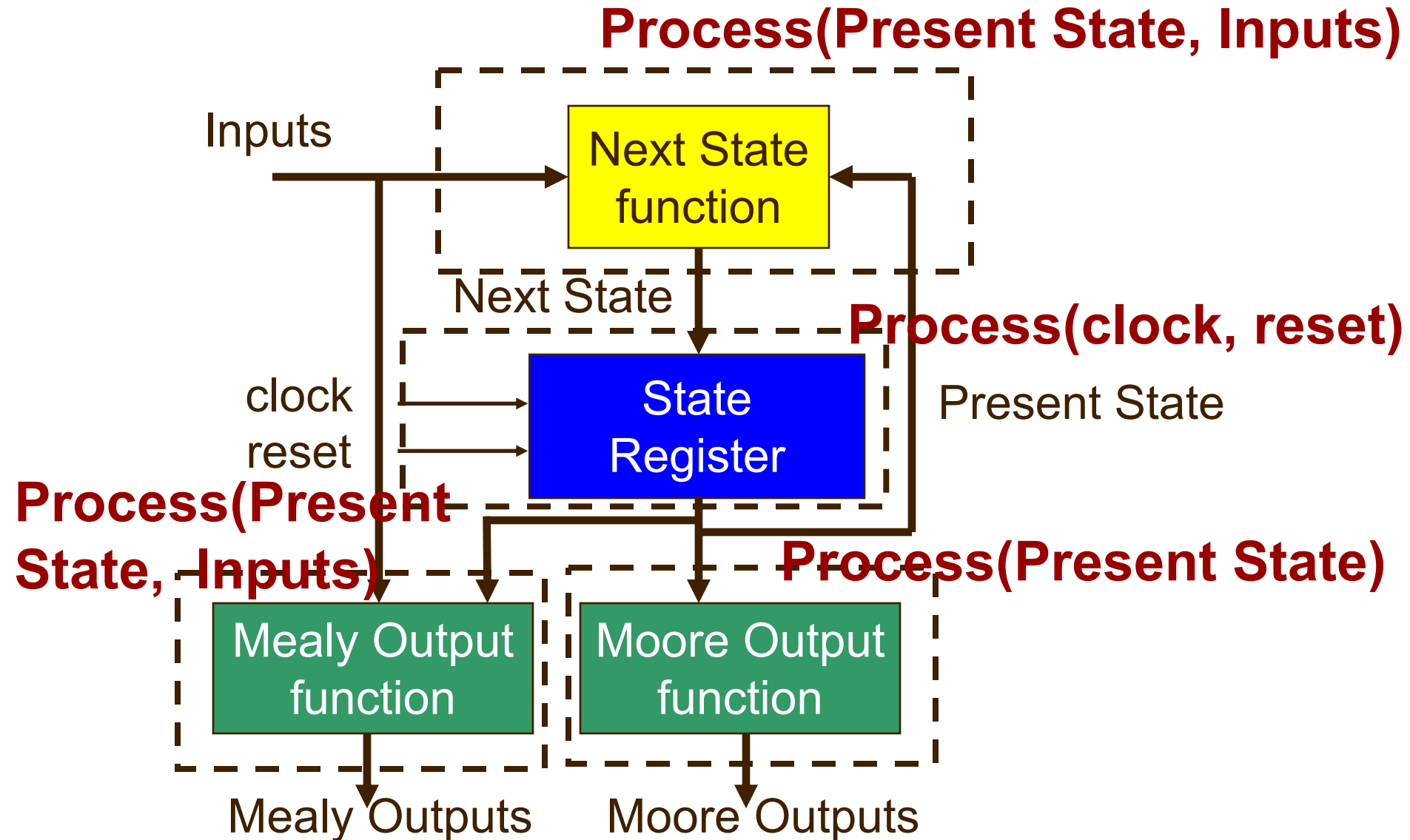
Alternative Coding Styles
by Dr. Chu
(to be used with caution)

Traditional Coding Style

process(clock, reset)

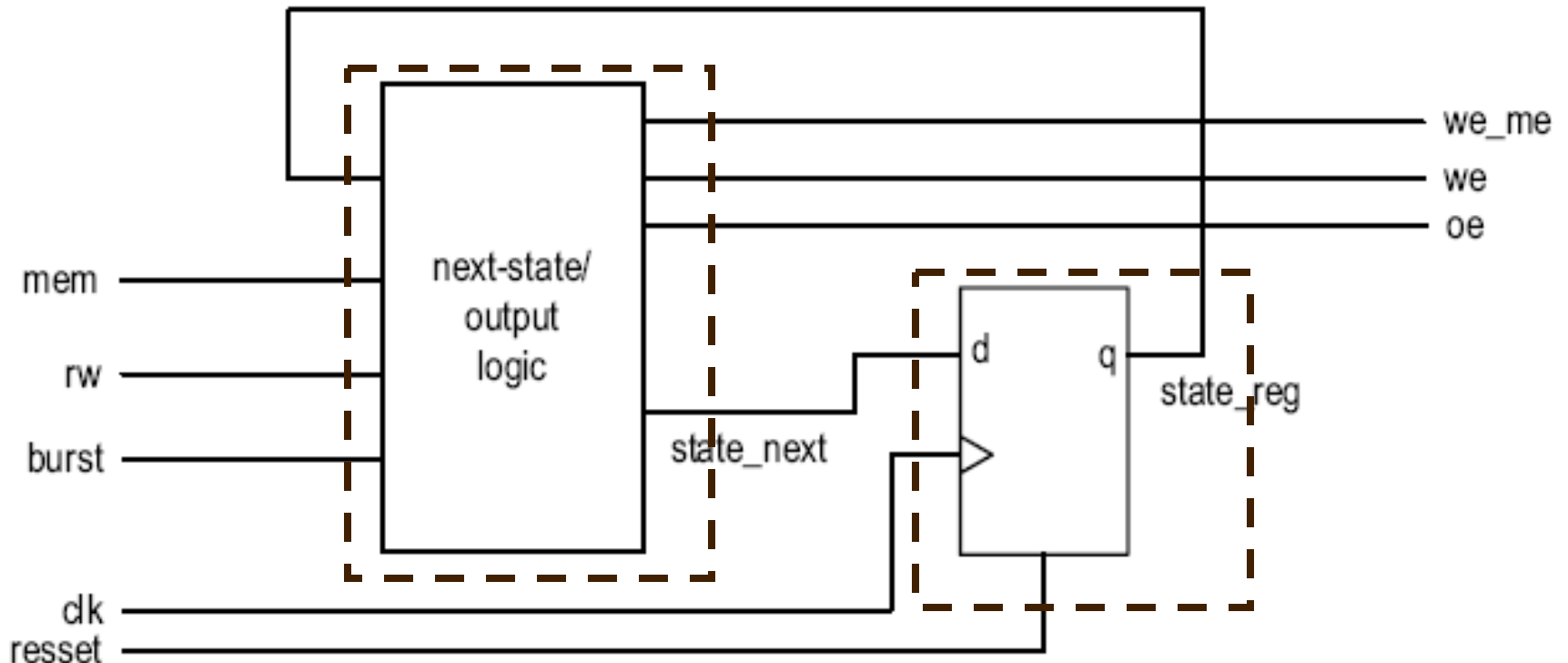


Alternative Coding Style 1



Alternative Coding Style 2

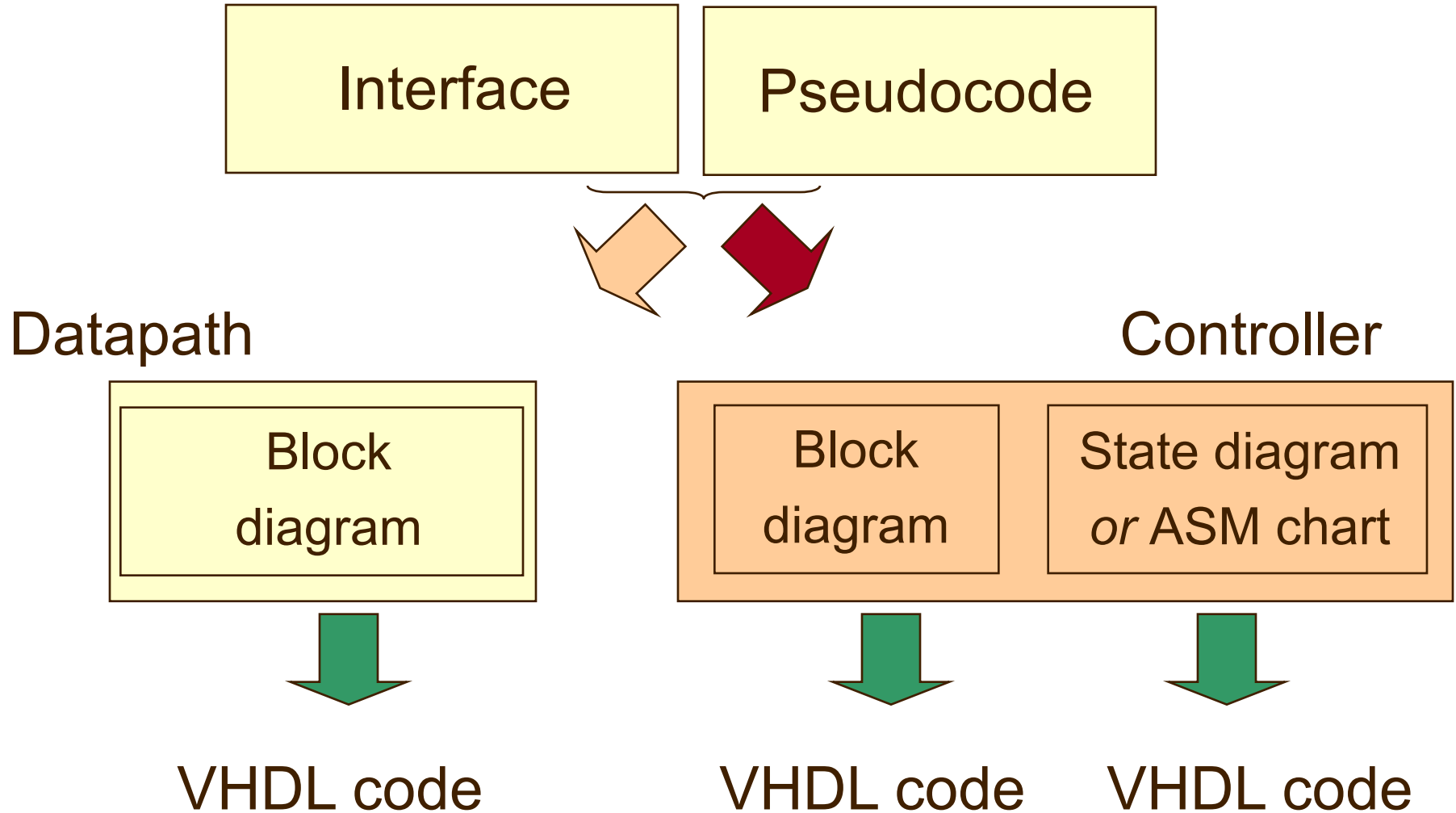
Process(Present State,Inputs)



Process(clk, reset)

Backup

Hardware Design with RTL VHDL

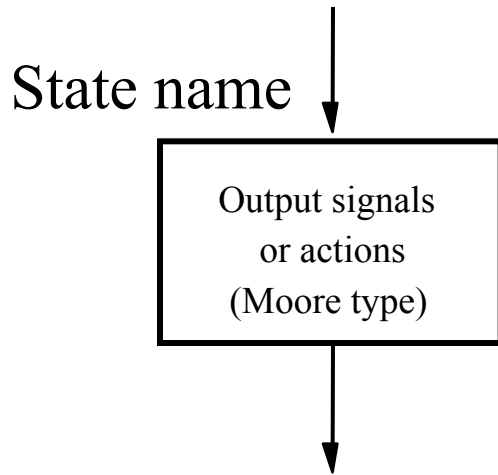


Algorithmic State Machine (ASM) Charts

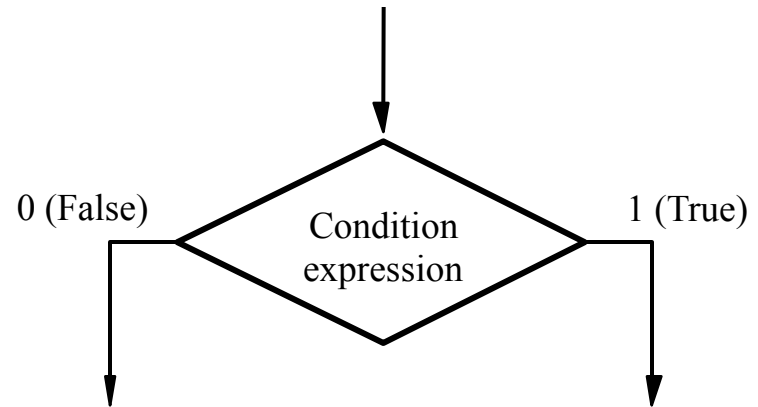
Algorithmic State Machine

Algorithmic State Machine –
representation of a Finite State Machine
suitable for FSMs with a larger number of
inputs and outputs compared to FSMs
expressed using state diagrams and state
tables.

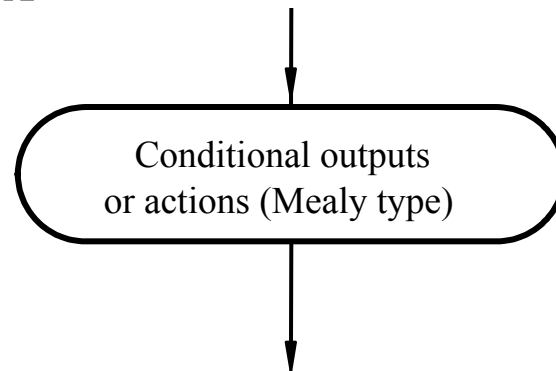
Elements used in ASM charts (1)



(a) State box



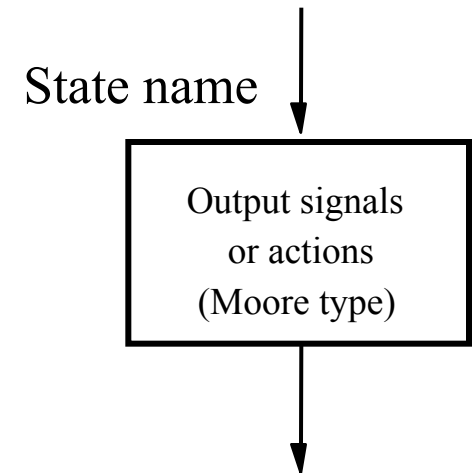
(b) Decision box



(c) Conditional output box

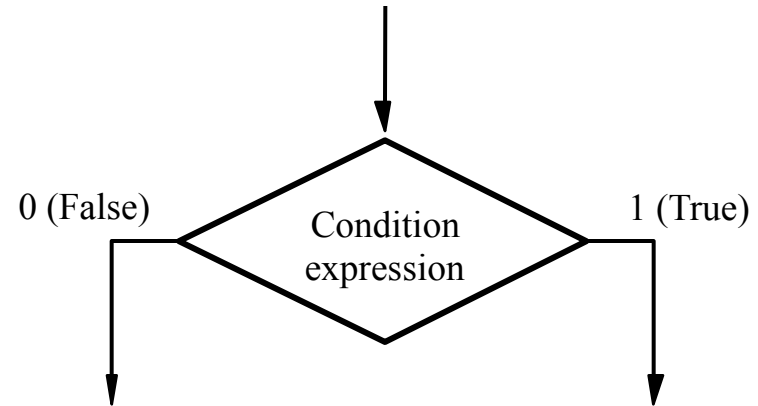
State Box

- A **state box** represents a state.
- Equivalent to a node in a state diagram or a row in a state table.
- Contains register transfer actions or output signals
- **Moore-type outputs are listed inside of the box.**
- It is customary to write only the name of the signal that has to be asserted in the given state, e.g., z instead of $z \leq 1$.
- Also, it might be useful to write an action to be taken, e.g., $\text{count} \leq \text{count} + 1$, and only later translate it to asserting a control signal that causes a given action to take place (e.g., enable signal of a counter).



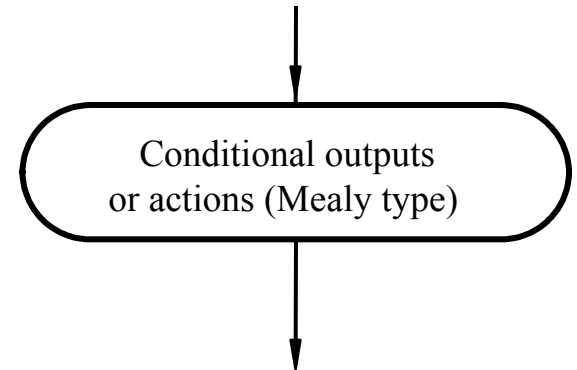
Decision Box

- A **decision box** indicates that a given condition is to be tested and the exit path is to be chosen accordingly.
- The condition expression may include one or more inputs to the FSM.



Conditional Output Box

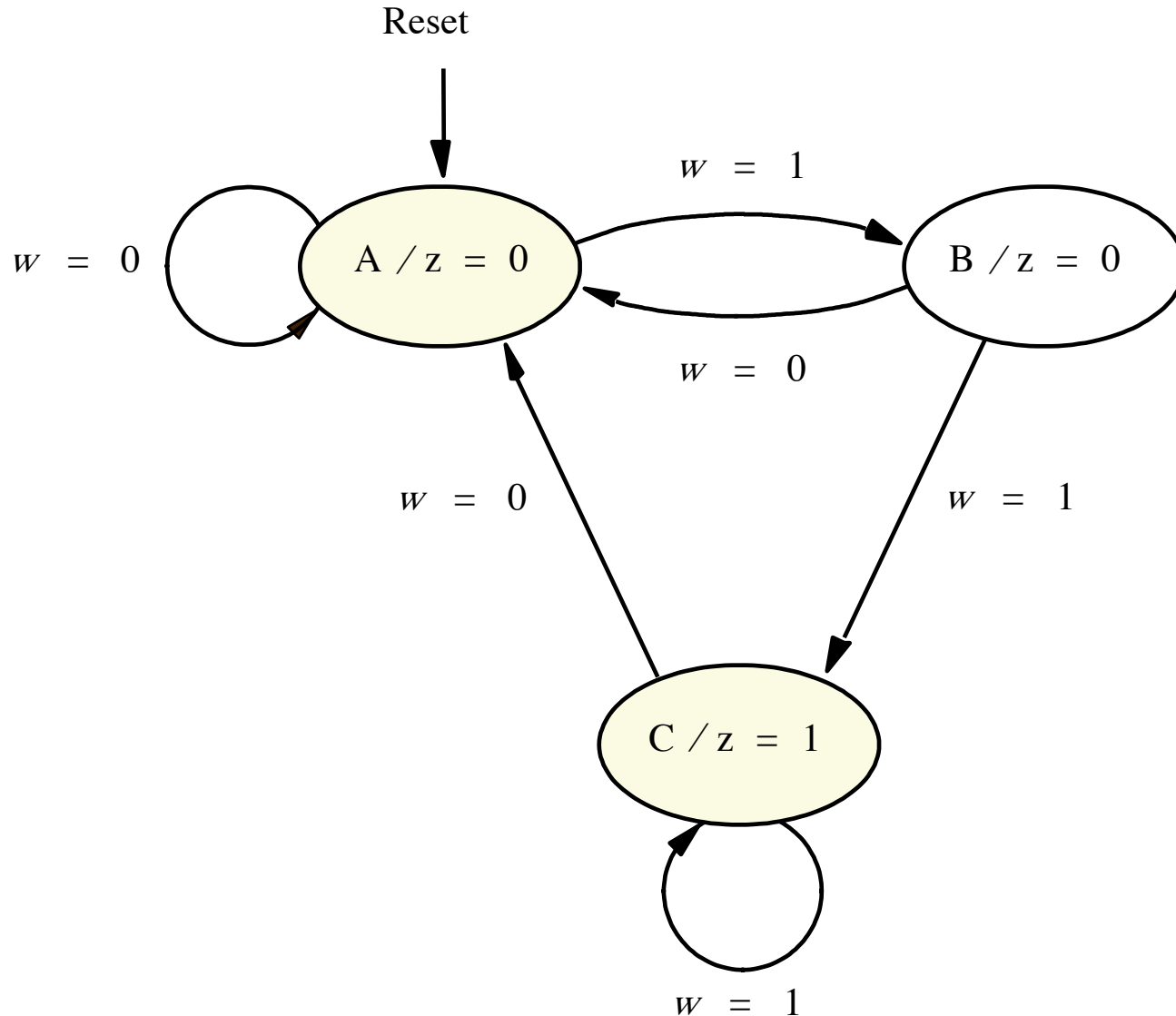
- A **conditional output box** denotes output signals that are of the **Mealy** type.
- The condition that determines whether such outputs are generated is specified in the decision box.



ASMs Representing Simple FSMs

- Algorithmic state machines can model both Mealy and Moore Finite State Machines
- They can also model machines that are of the mixed type.

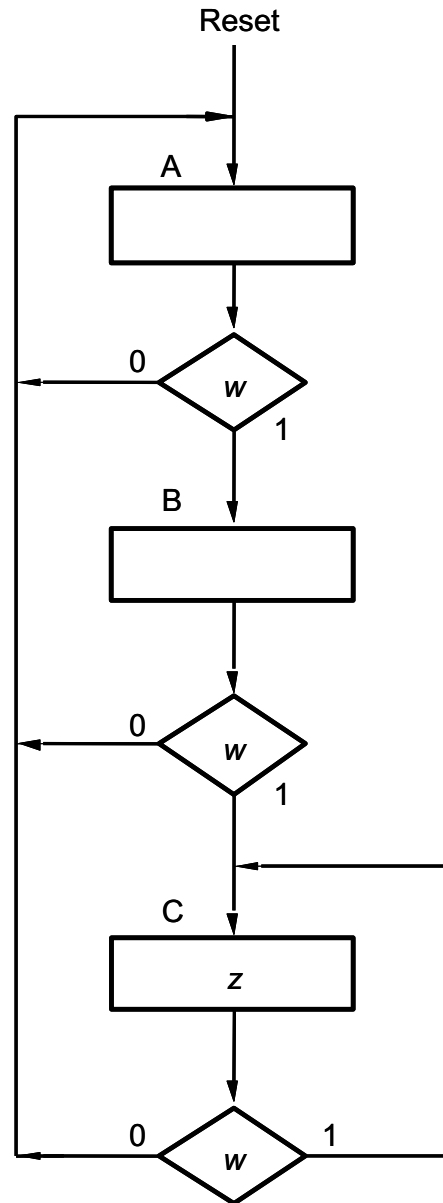
Moore FSM – Example 2: State diagram



Moore FSM – Example 2: State Table

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

ASM Chart for Moore FSM – Example 2

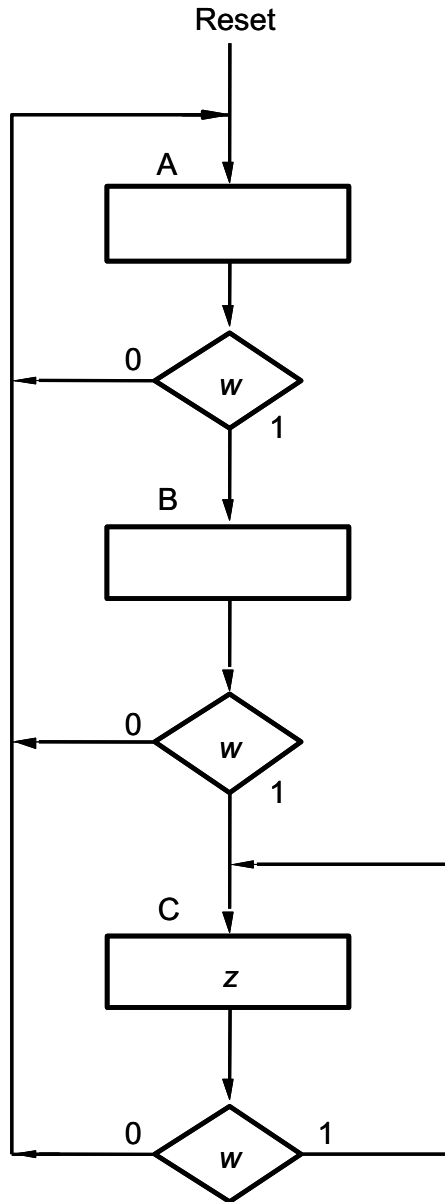


Example 2: VHDL Code (1)

```
entity simple is
    port( clock      : in STD_LOGIC;
          resetn     : in STD_LOGIC;
          w           : in STD_LOGIC;
          z           : out STD_LOGIC);
end simple ;

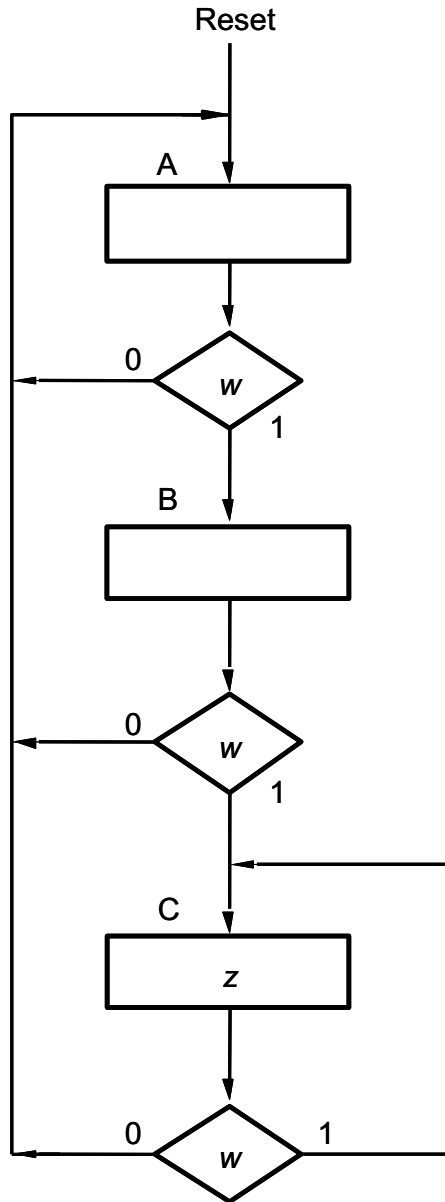
architecture Behavior of simple is
    type State_type IS (A, B, C) ;
    signal state : State_type ;
begin
    process( resetn, clock )
    begin
        if resetn = '0' then
            state <= A ;
        elsif rising_edge(Clock) then
```

Example 2: VHDL Code (2)



```
case state is
  when A =>
    if w = '0' then
      state <= A ;
    else
      state <= B ;
    end if;
  when B =>
    if w = '0' then
      state <= A ;
    else
      state <= C ;
    end if;
  when C =>
    if w = '0' then
      state <= A ;
    else
      state <= C ;
    end if;
end case;
```

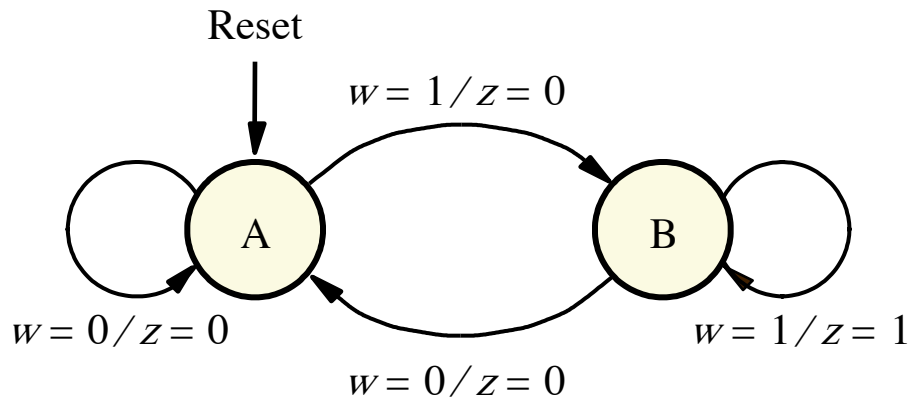
Example 2: VHDL Code (3)



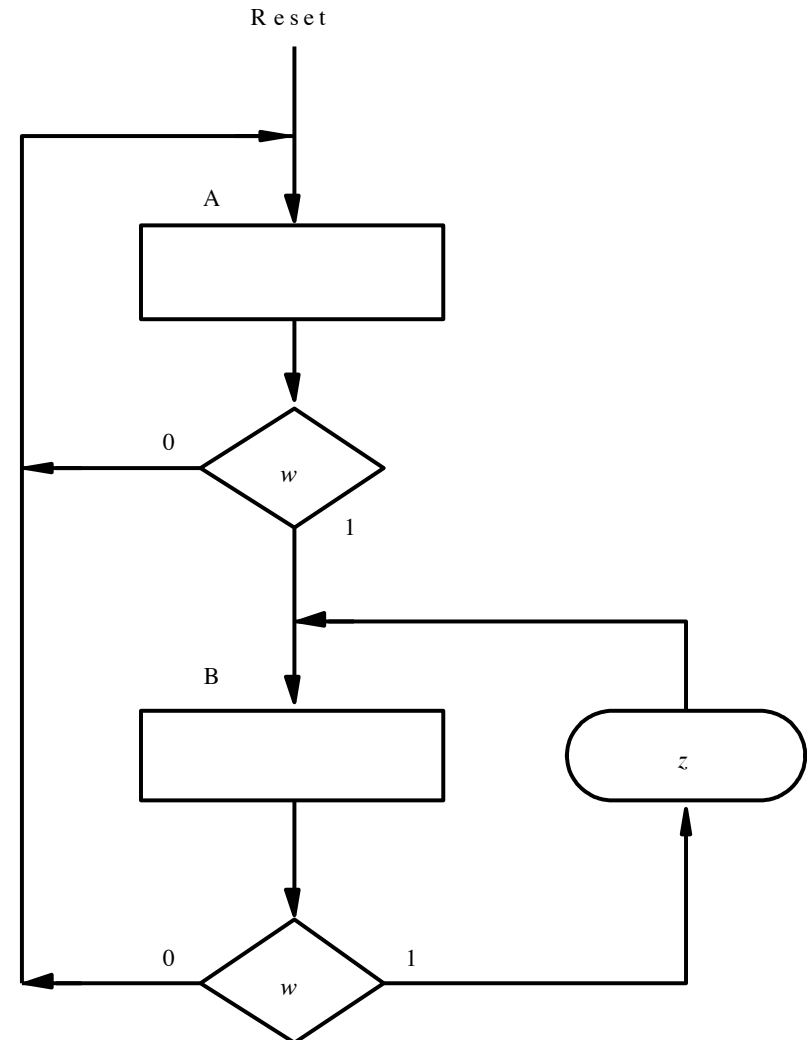
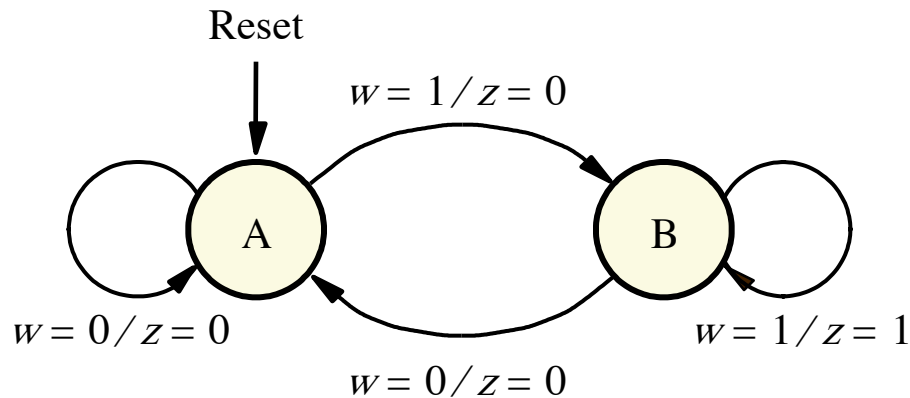
```
END IF;  
END PROCESS;
```

```
z <= '1' when state = C else  
      '0';  
END Behavior;
```

Mealy FSM – Example 3: State Diagram



ASM Chart for Mealy FSM – Example 3



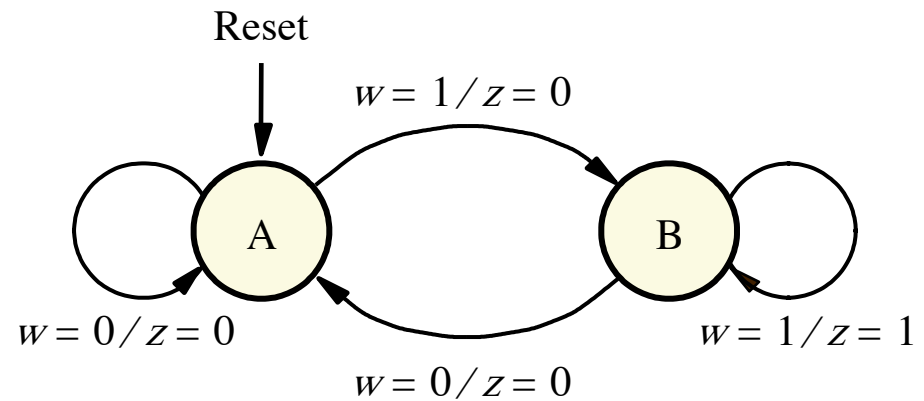
Example 3: VHDL Code (1)

```
entity Mealy is
    PORT ( clock : IN  STD_LOGIC;
          resetn  : IN  STD_LOGIC;
          w       : IN  STD_LOGIC;
          z       : OUT STD_LOGIC);
end Mealy;

architecture Behavior of Mealy is
    type State_type is (A, B) ;
    signal state: State_type ;
begin
    process (resetn, clock)
    begin
        if resetn = '0' then
            state<= A ;
        elsif rising_edge(clock) then
```

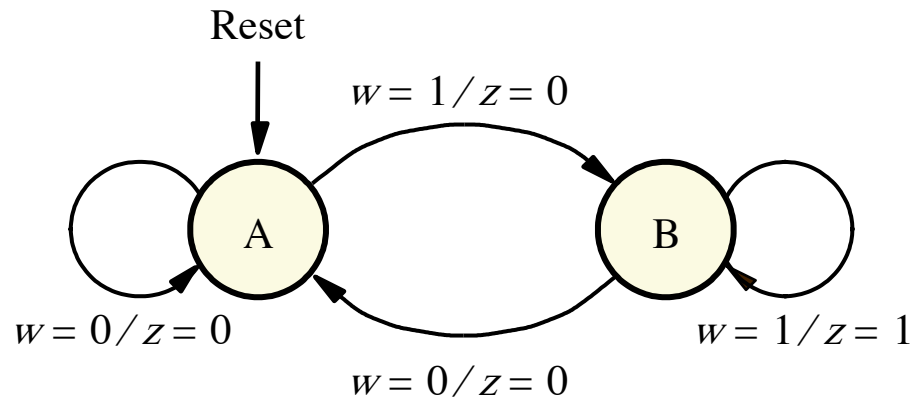
Example 3: VHDL Code (2)

```
case state is
  when A =>
    if w = '0' then
      state<= A ;
    else
      state<= B ;
    end if;
  when B =>
    if w = '0' then
      state<= A ;
    else
      state<= B ;
    end if;
end case;
end if;
end process;
```



Example 3: VHDL Code (3)

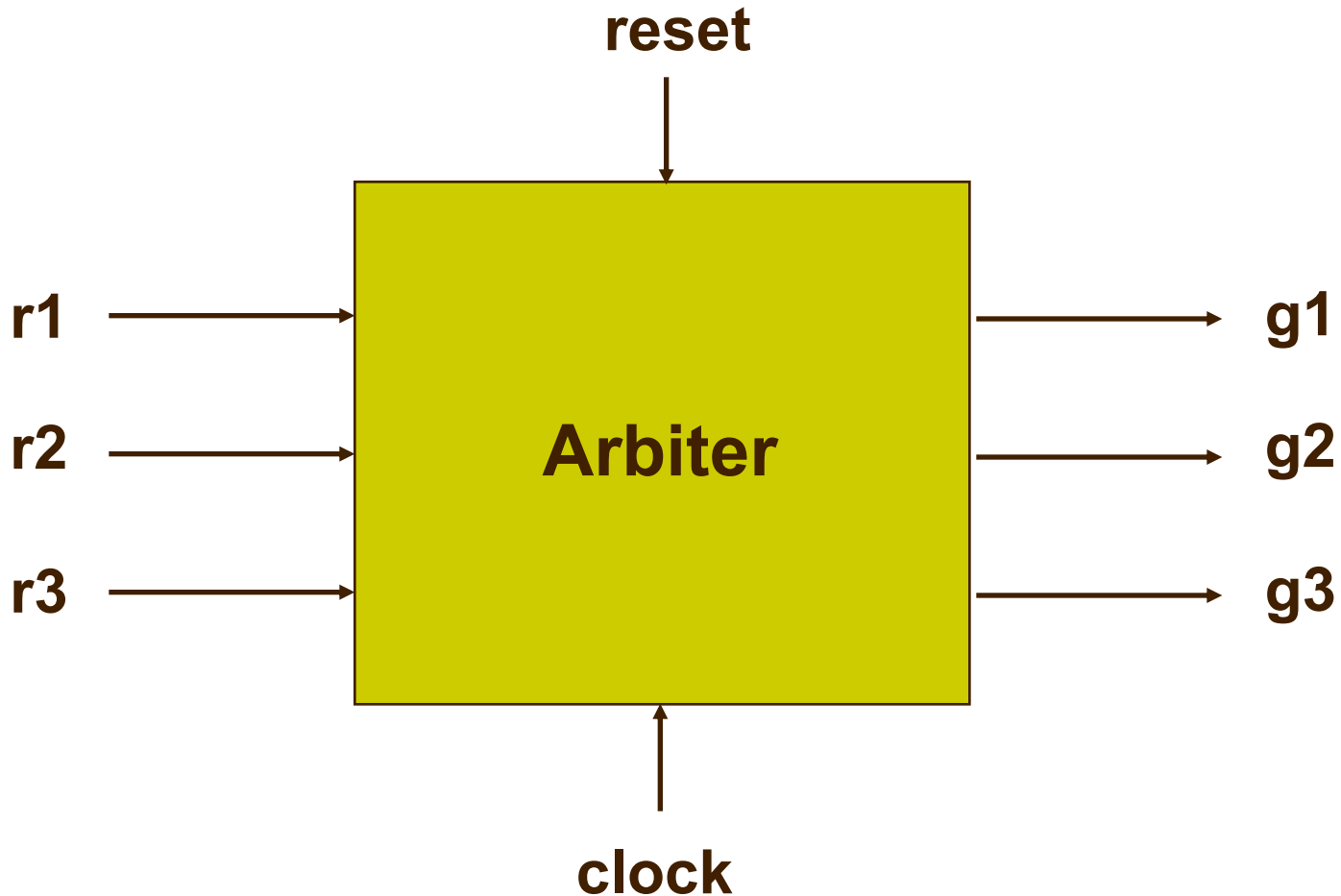
```
z <= '1' when (y = B) and (w='1') else  
      '0';  
end architecture Behavior ;
```



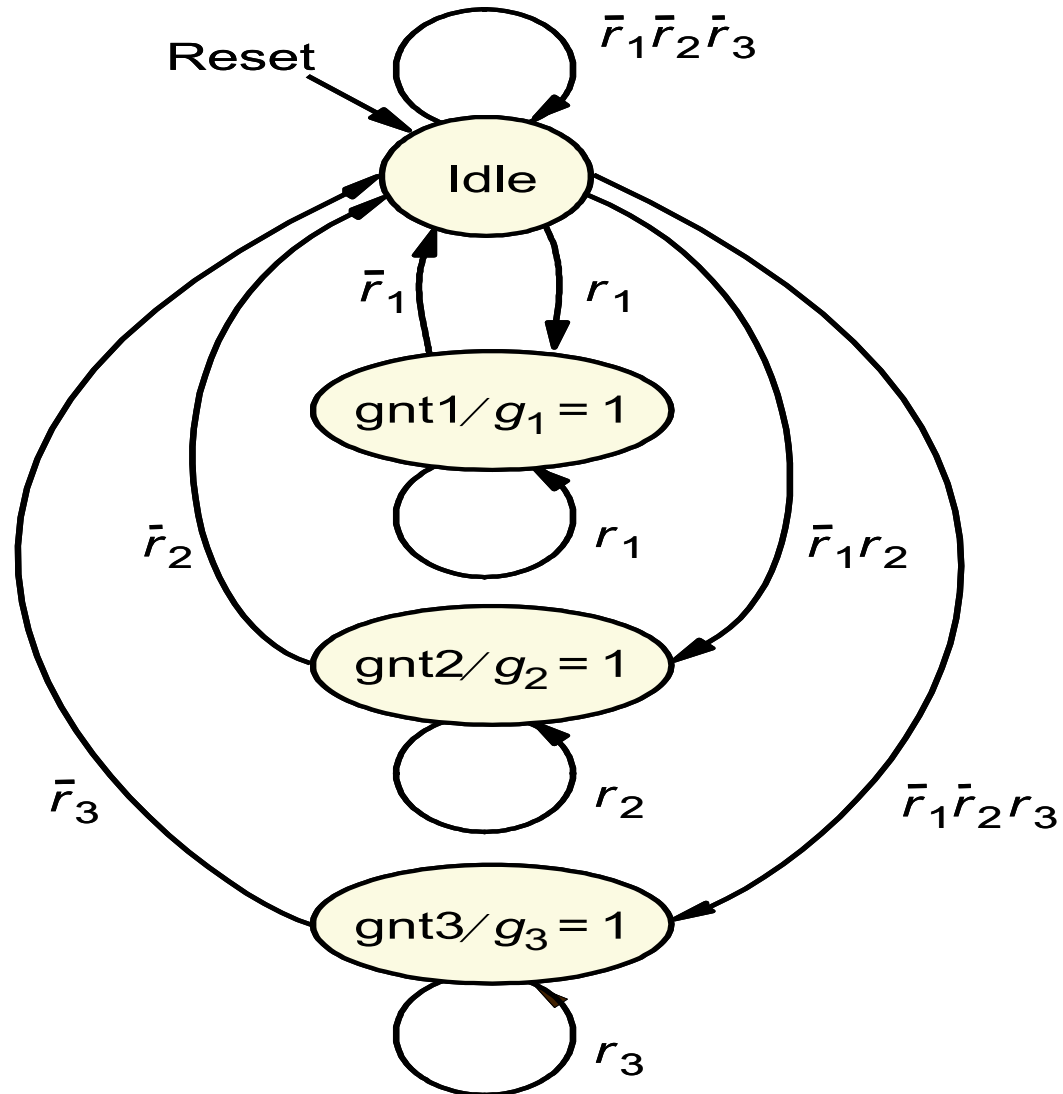
Case Study 1

Arbiter

Arbiter – Interface



Arbiter – FSM



Arbiter – VHDL Code

```
ENTITY arbiter IS
    PORT(Clock, Resetn    : IN  STD_LOGIC ;
          r               : IN  STD_LOGIC_VECTOR(1 TO 3);
          g               : OUT STD_LOGIC_VECTOR(1 TO 3));
END arbiter;
```

```
ARCHITECTURE Behavior OF arbiter IS
    TYPE State_type IS (Idle, gnt1, gnt2, gnt3);
    SIGNAL state: State_type;
begin
```


Arbiter – VHDL Code (cont'd)

```
PROCESS(Resetn, Clock)
BEGIN
    IF Resetn = '0' THEN
        state <= Idle ;
    ELSIF (Clock'EVENT AND Clock = '1') THEN
        CASE state IS
            WHEN Idle =>
                IF r(1) = '1' THEN          state <= gnt1 ;
                ELSIF r(2) = '1' THEN        state <= gnt2 ;
                ELSIF r(3) = '1' THEN        state <= gnt3 ;
                ELSE                          state <= Idle ;
                END IF ;
            WHEN gnt1 =>
                IF r(1) = '1' THEN          state <= gnt1 ;
                ELSE                          state <= Idle ;
                END IF ;
```

-- continue on the next slide

Arbiter – VHDL Code (cont'd)

```
    WHEN gnt2 =>
        IF r(2) = '1' THEN          state <= gnt2 ;
        ELSE                        state <= Idle ;
        END IF ;
```

```
    WHEN gnt3 =>
        IF r(3) = '1' THEN          state <= gnt3 ;
        ELSE                        state <= Idle ;
        END IF ;
```

```
    END CASE ;
```

```
END IF ;
```

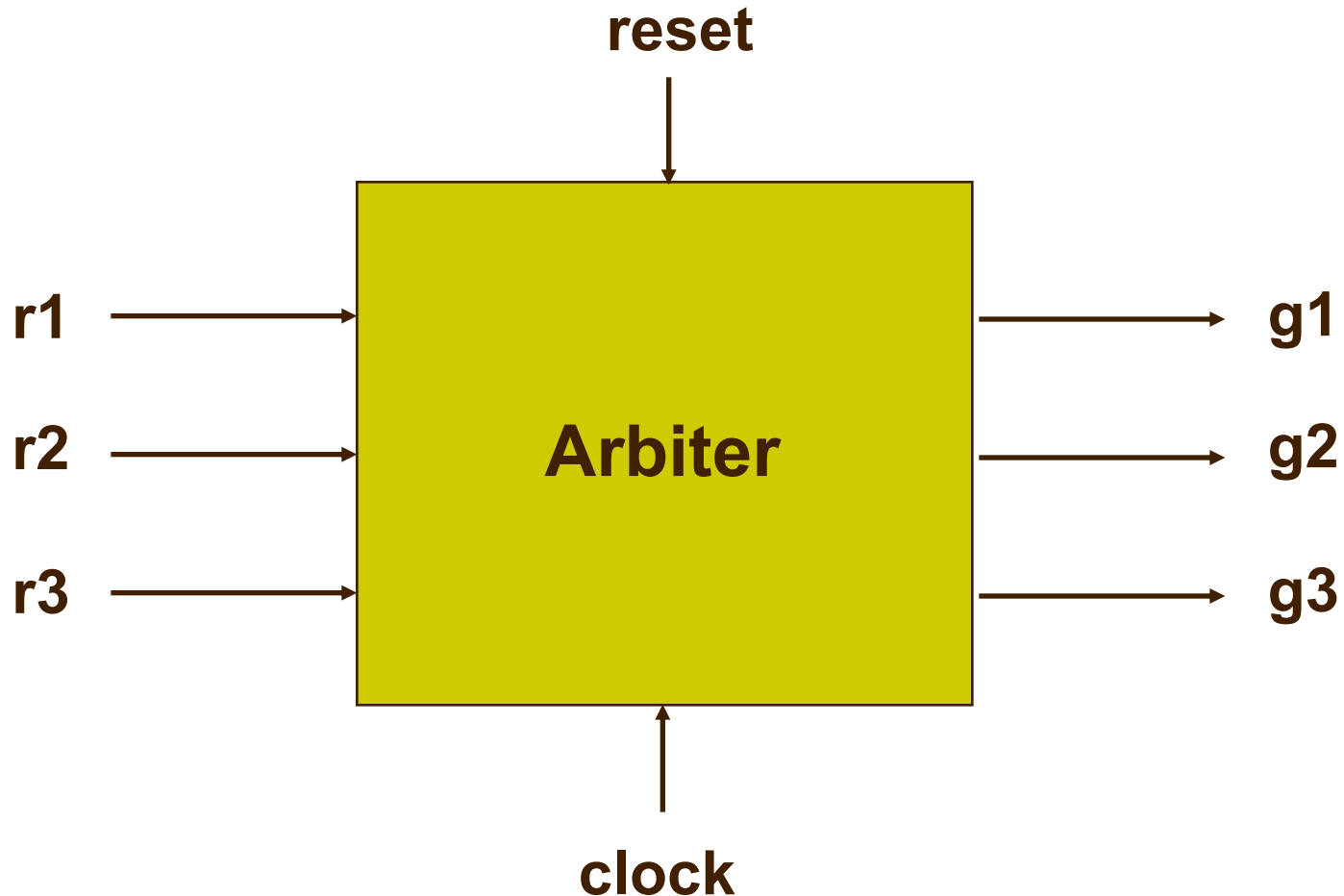
```
END PROCESS ;
```

```
-- continue on the next slide
```

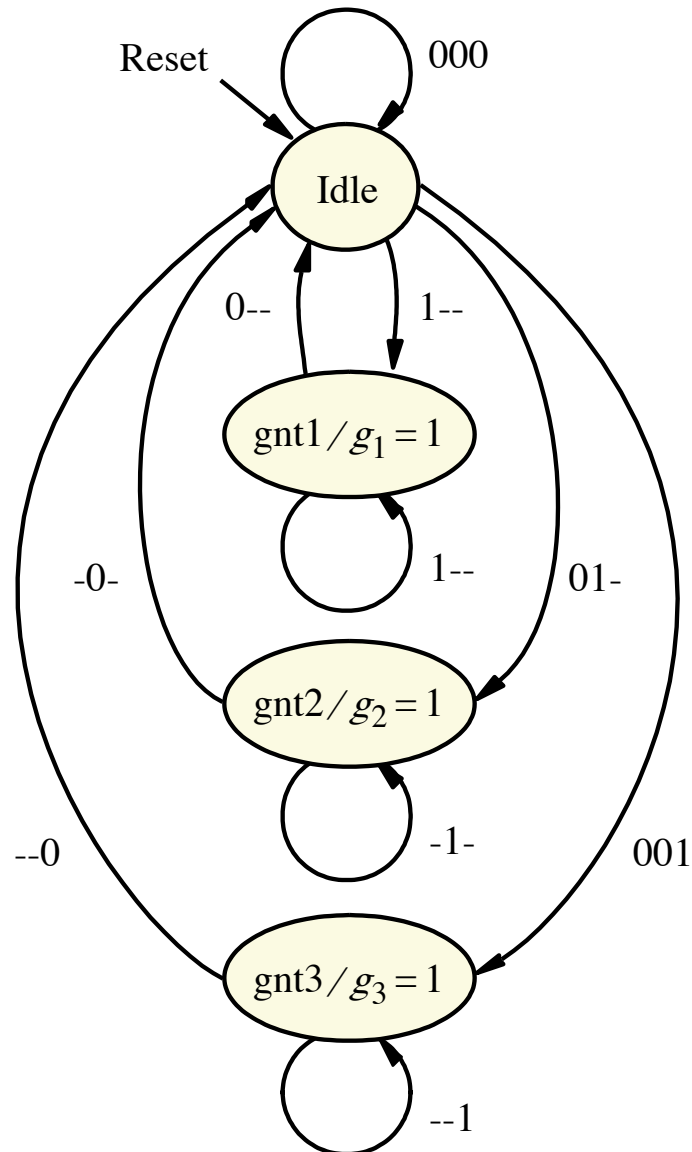
Arbiter – VHDL Code (cont'd)

```
-- output function
g(1) <= '1' WHEN state = gnt1 ELSE
        '0';
g(2) <= '1' WHEN state = gnt2 ELSE
        '0';
g(3) <= '1' WHEN state = gnt3 ELSE
        '0';
END architecture Behavior ;
```

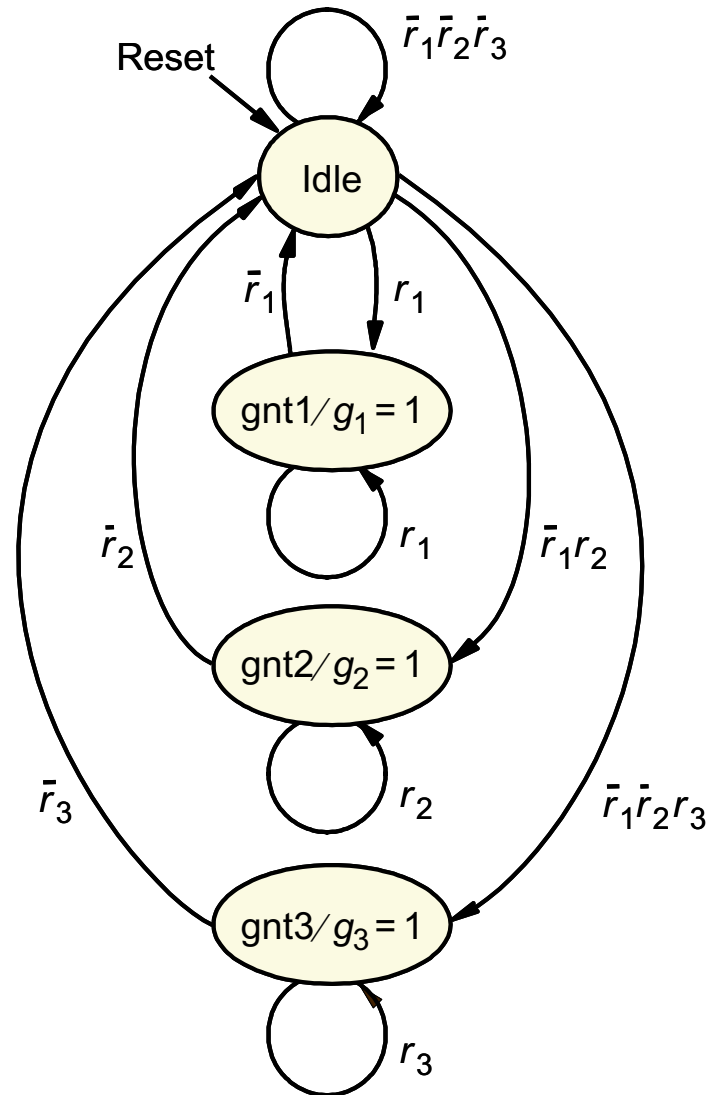
Control Unit Example: Arbiter (1)



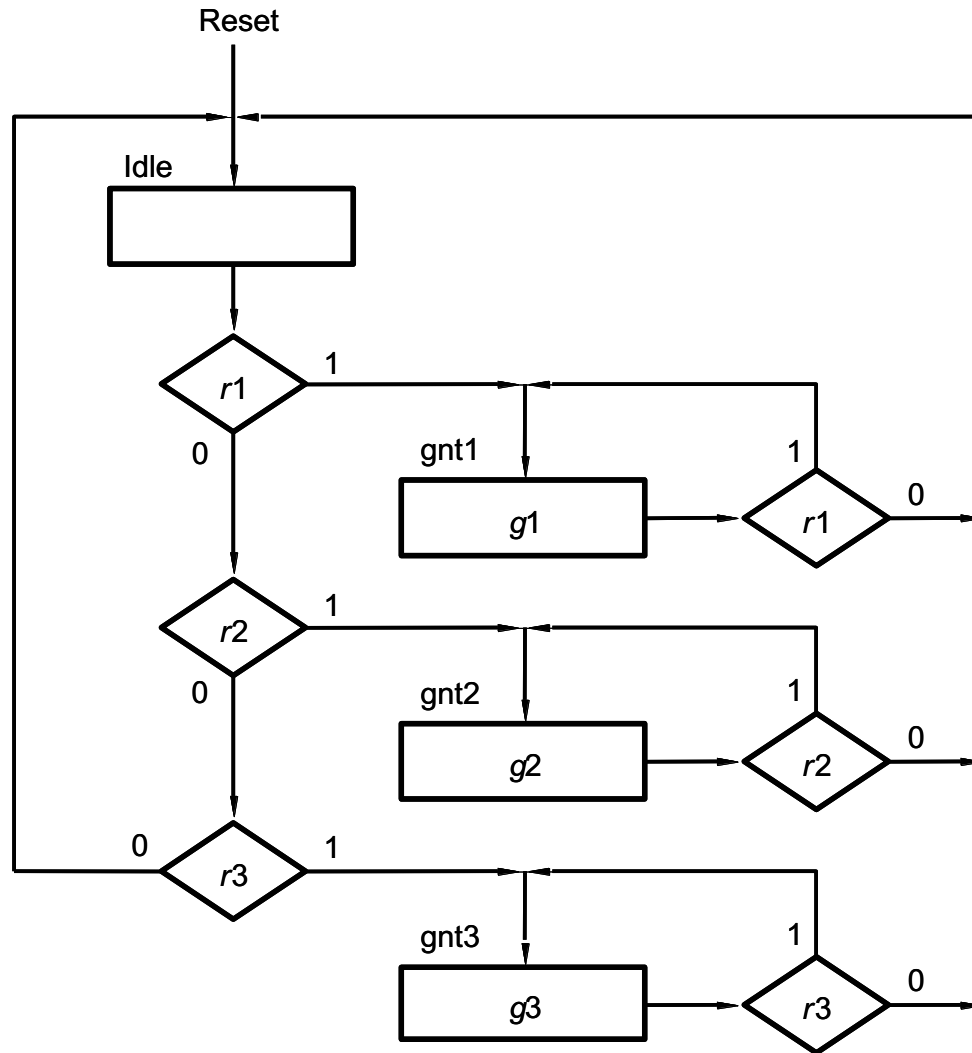
Control Unit Example: Arbiter (2)



Control Unit Example: Arbiter (3)



ASM Chart for Control Unit - Example 4



Example 4: VHDL Code (1)

```
ENTITY arbiter IS
    PORT(Clock, Resetn      : IN  STD_LOGIC ;
          r                  : IN  STD_LOGIC_VECTOR(1 TO 3);
          g                  : OUT STD_LOGIC_VECTOR(1 TO 3));
END arbiter;
```

```
ARCHITECTURE Behavior OF arbiter IS
    TYPE State_type IS (Idle, gnt1, gnt2, gnt3);
    SIGNAL state: State_type;
begin
```


Example 4: VHDL code (2)

```
PROCESS(Resetn, Clock)
BEGIN
    IF Resetn = '0' THEN
        state <= Idle ;
    ELSIF (Clock'EVENT AND Clock = '1') THEN
        CASE state IS
            WHEN Idle =>
                IF r(1) = '1' THEN          state <= gnt1 ;
                ELSIF r(2) = '1' THEN        state <= gnt2 ;
                ELSIF r(3) = '1' THEN        state <= gnt3 ;
                ELSE                          state <= Idle ;
                END IF ;
            WHEN gnt1 =>
                IF r(1) = '1' THEN          state <= gnt1 ;
                ELSE                          state <= Idle ;
                END IF ;
```

-- continue on the next slide

Example 4: VHDL code (3)

```
    WHEN gnt2 =>
        IF r(2) = '1' THEN    state <= gnt2 ;
        ELSE                  state <= Idle ;
        END IF ;
```

```
    WHEN gnt3 =>
        IF r(3) = '1' THEN    state <= gnt3 ;
        ELSE                  state <= Idle ;
        END IF ;
```

```
    END CASE ;
```

```
END IF ;
```

```
END PROCESS ;
```

```
-- continue on the next slide
```

Example 4: VHDL code (3)

```
g(1) <= '1' WHEN y = gnt1 ELSE  
        '0';  
g(2) <= '1' WHEN y = gnt2 ELSE  
        '0';  
g(3) <= '1' WHEN y = gnt3 ELSE  
        '0';  
END architecture Behavior ;
```

ASM Summary

- ASM (algorithmic state machine) chart
 - Flowchart-like diagram
 - Provides the same info as a state diagram
 - More descriptive, better for complex description
 - ASM block
 - One state box
 - One or more optional decision boxes:
with T (1) or F (0) exit path
 - One or more conditional output boxes:
for Mealy output

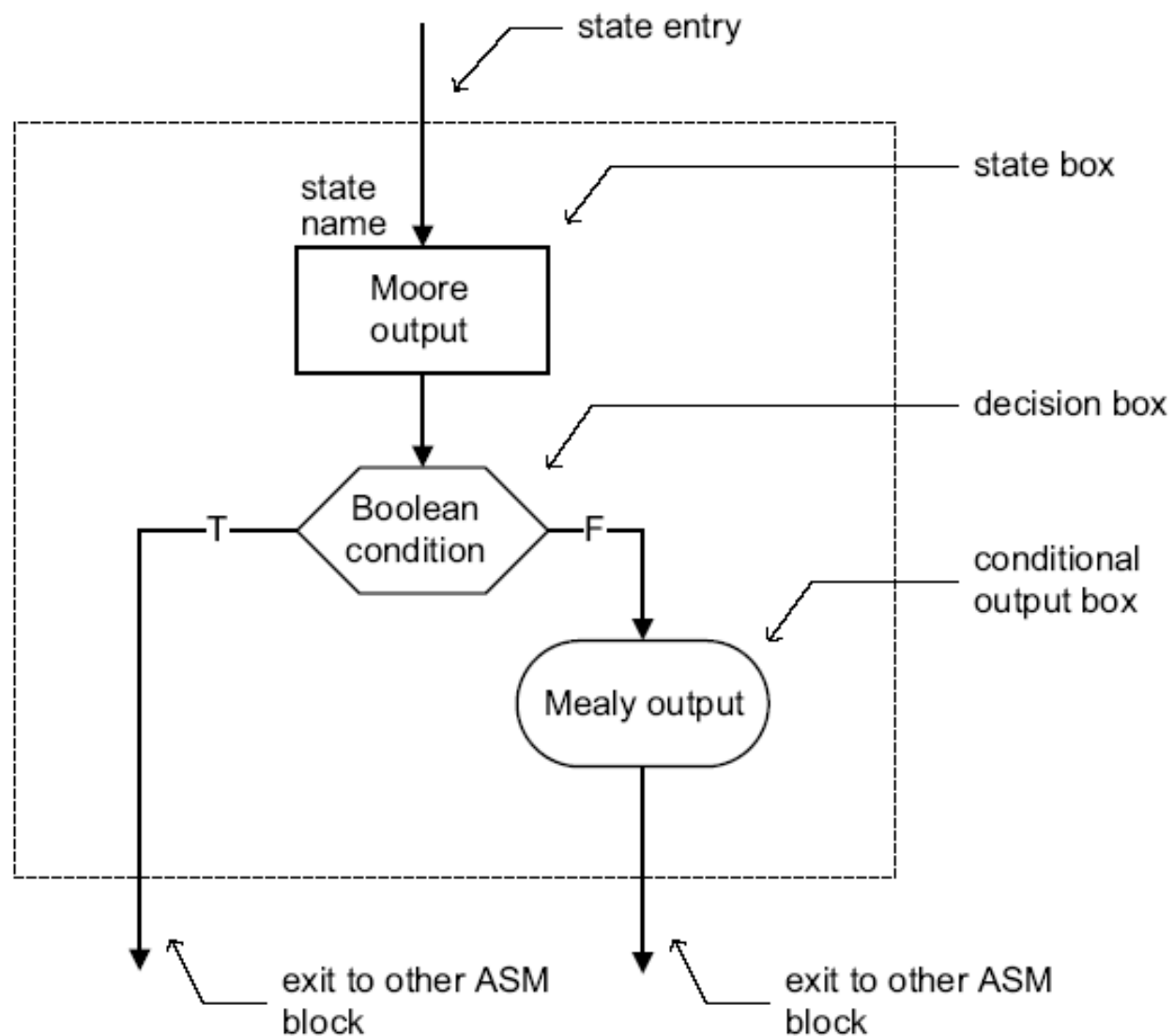
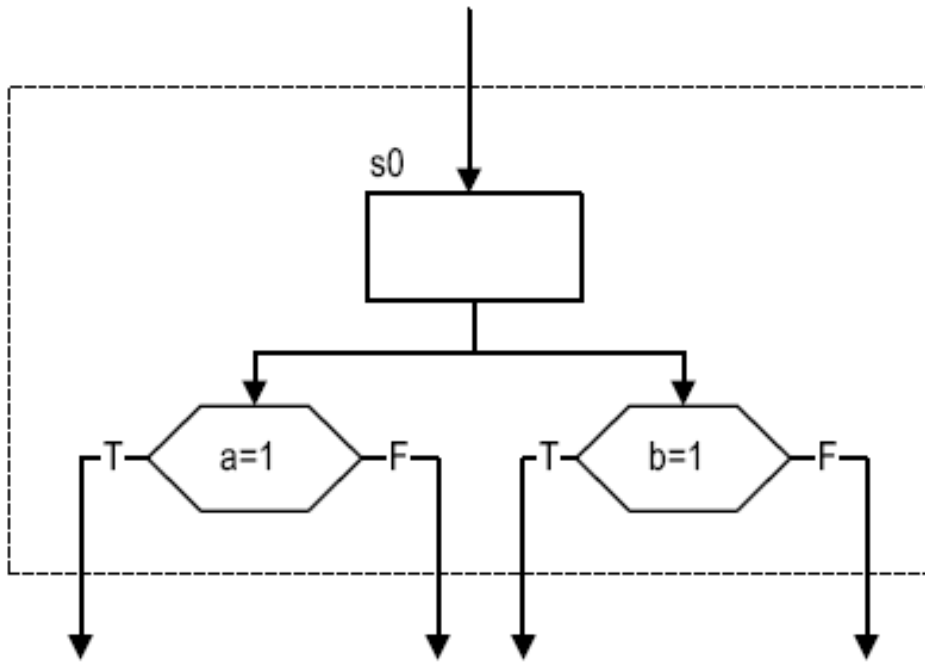


Figure 10.4 ASM block.

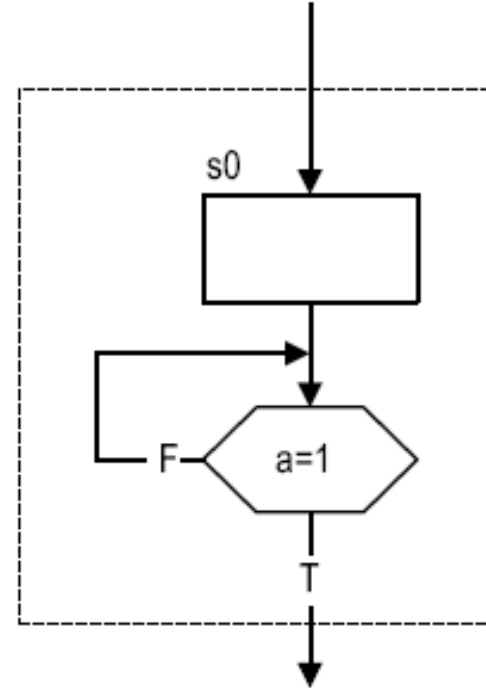
ASM Chart Rules

- Difference between a regular flowchart and an ASM chart:
 - Transition governed by clock
 - Transition occurs between ASM blocks
- Basic rules:
 - For a given input combination, there is **one unique exit path** from the current ASM block
 - Any **closed loop** in an ASM chart **must include a state box**

Incorrect ASM Charts



(a)

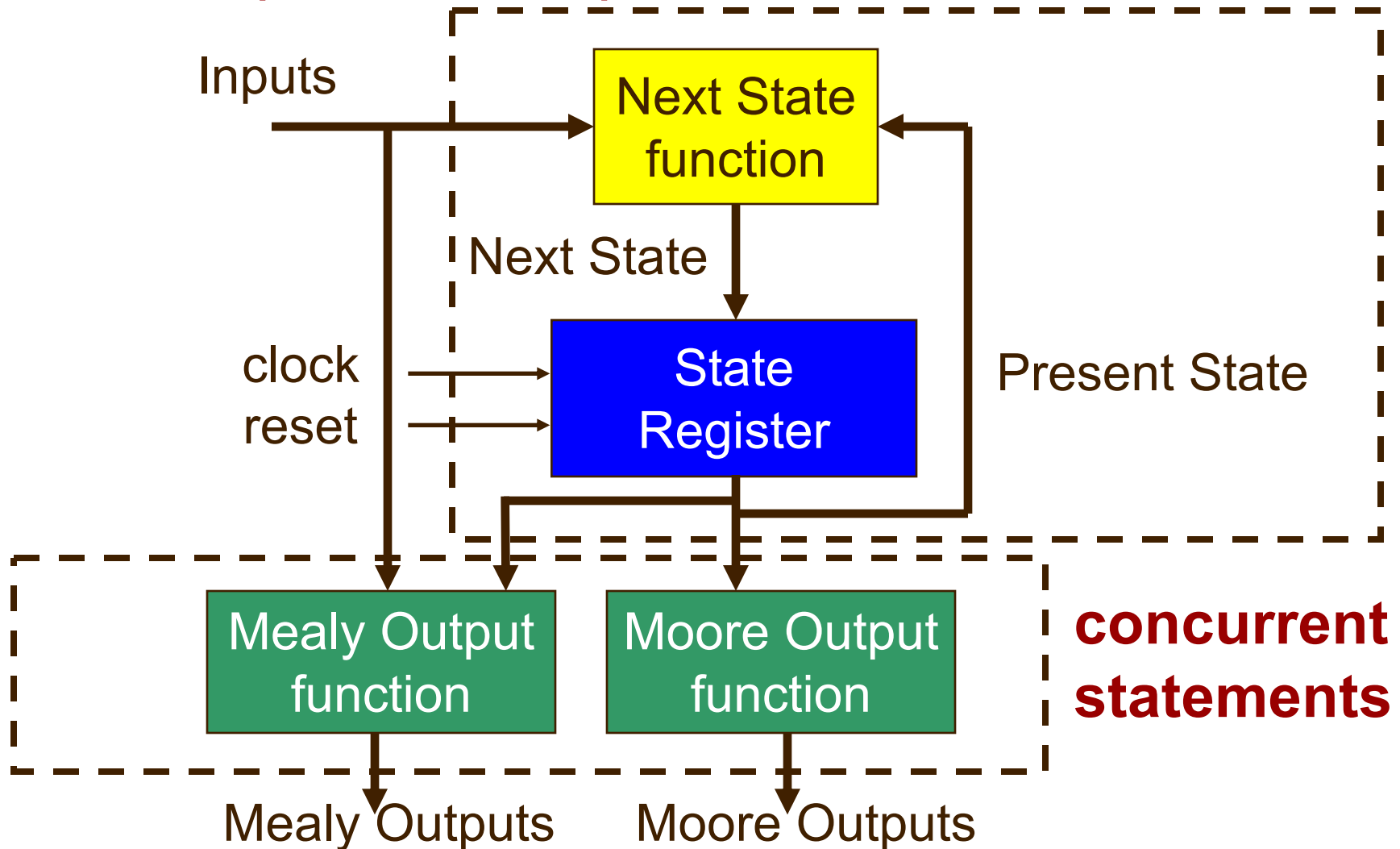


(b)

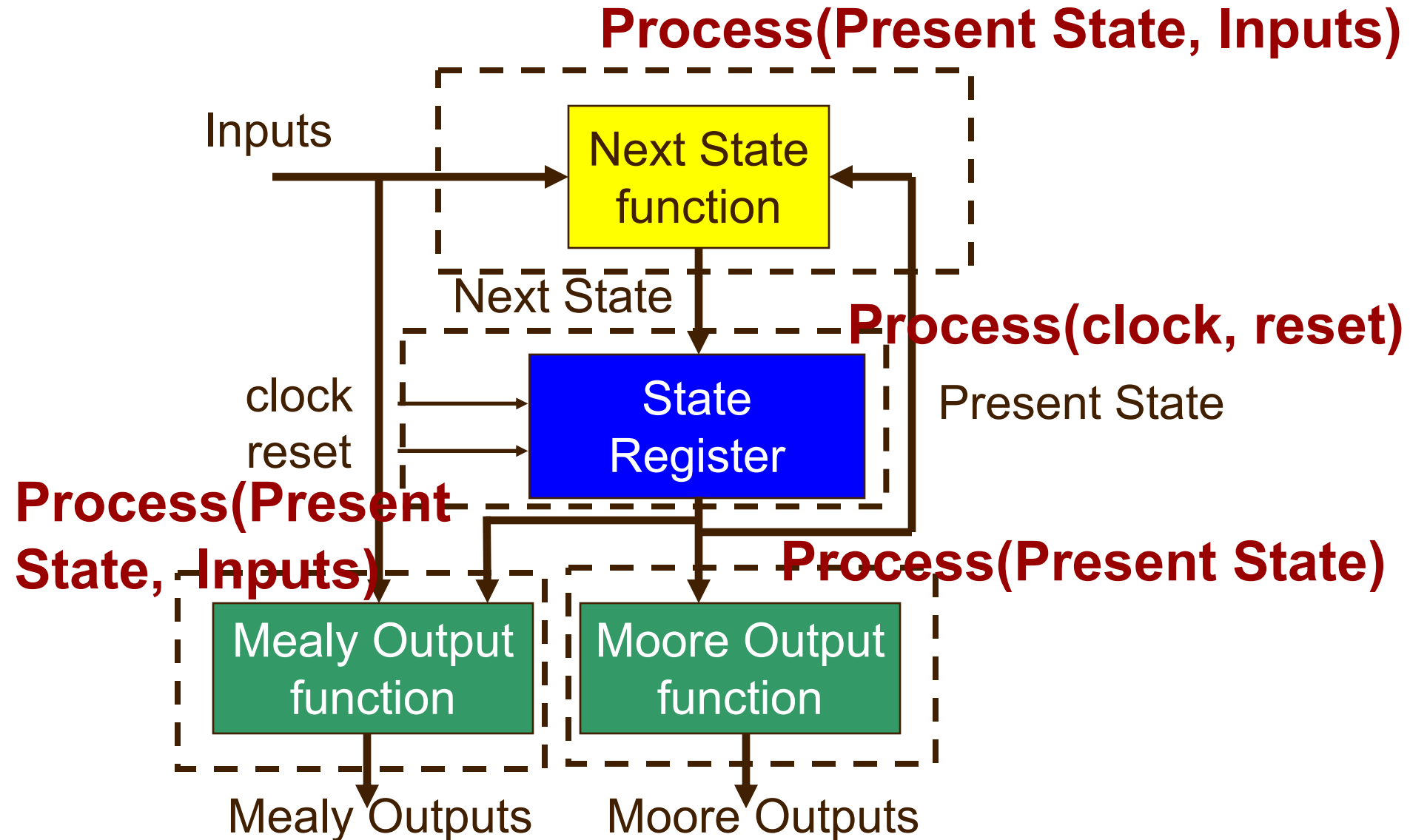
Alternative Coding Styles
by Dr. Chu
(to be used with caution)

Traditional Coding Style

process(clock, reset)



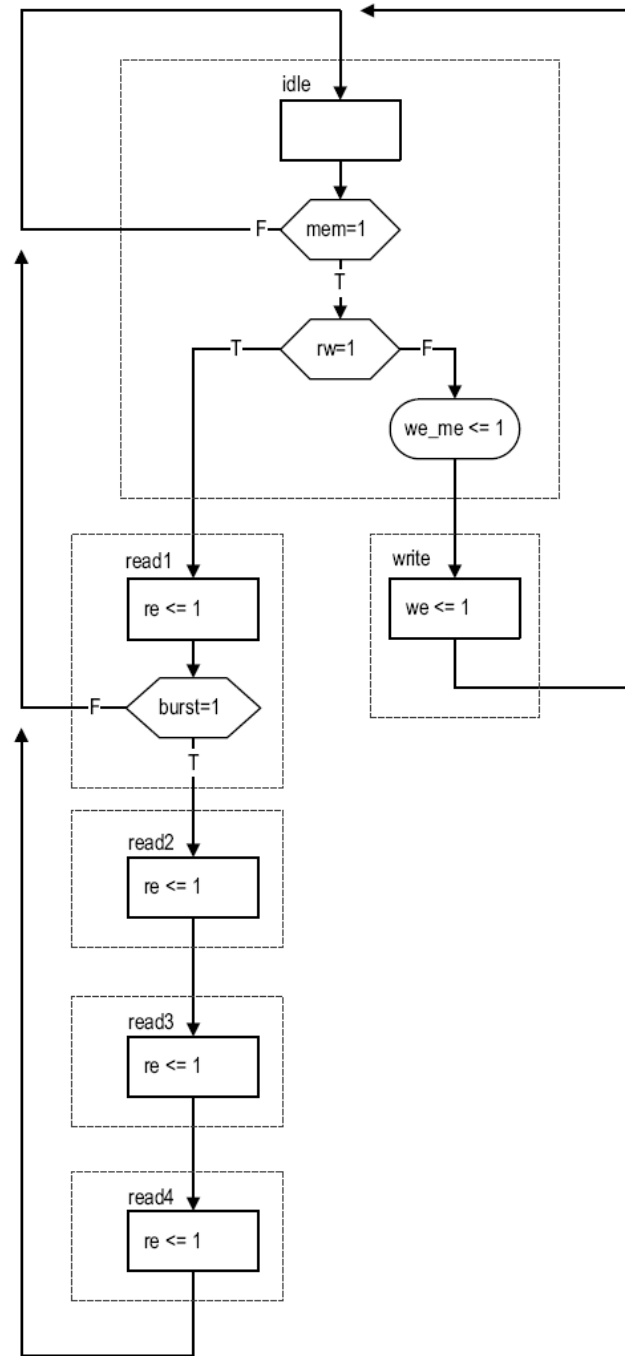
Alternative Coding Style 1



Next state logic depends on **mem**, **rw**, and **burst**.

Moore output: **re** and **we**.

Mealy output: **we_me** that depends on **mem** and **rw**.



```

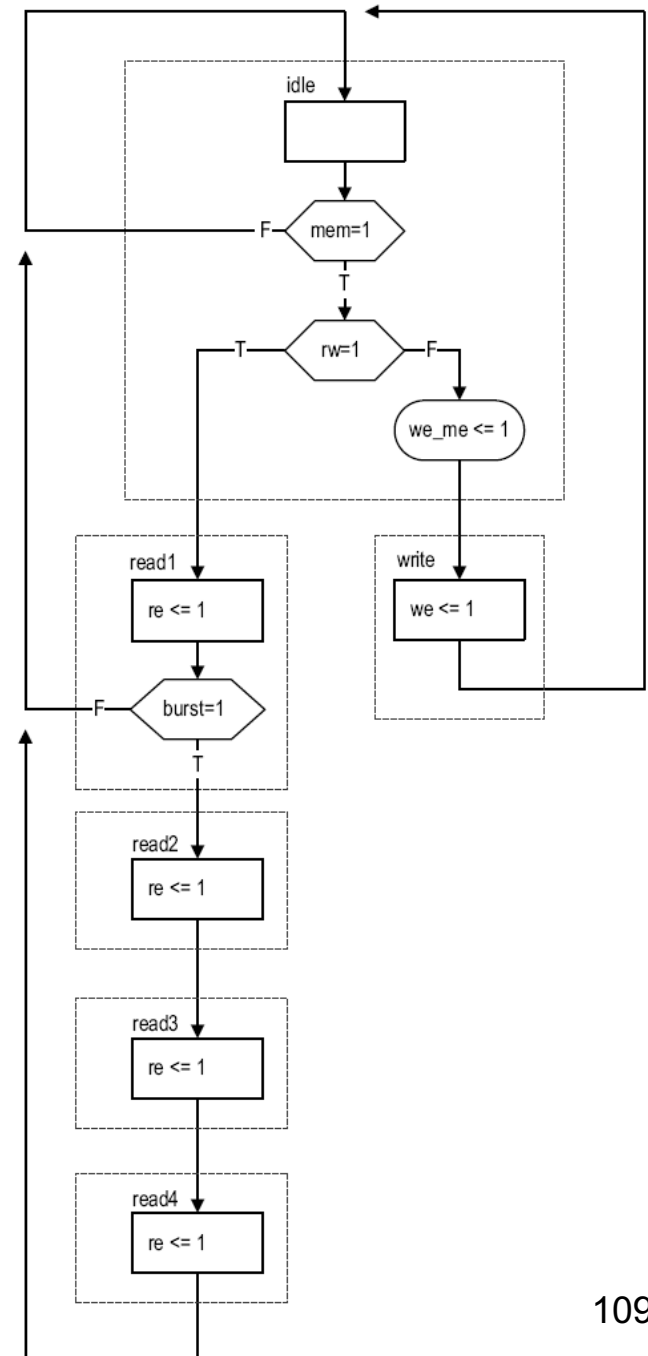
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
port (
;   clk, reset: in std_logic;
      mem, rw, burst: in std_logic;
      oe, we, we_me: out std_logic);
end mem_ctrl ;

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
  -- 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;
end mem_ctrl;

```

Next state logic depends on **mem**, **rw**, and **burst**.

```
-- next-state logic
process(state_reg, mem, rw, burst)
begin
  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 read1 =>
      if (burst='1') then
        state_next <= read2;
      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;
```



Moore output: **re** and **we**.

-- 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 =>
```

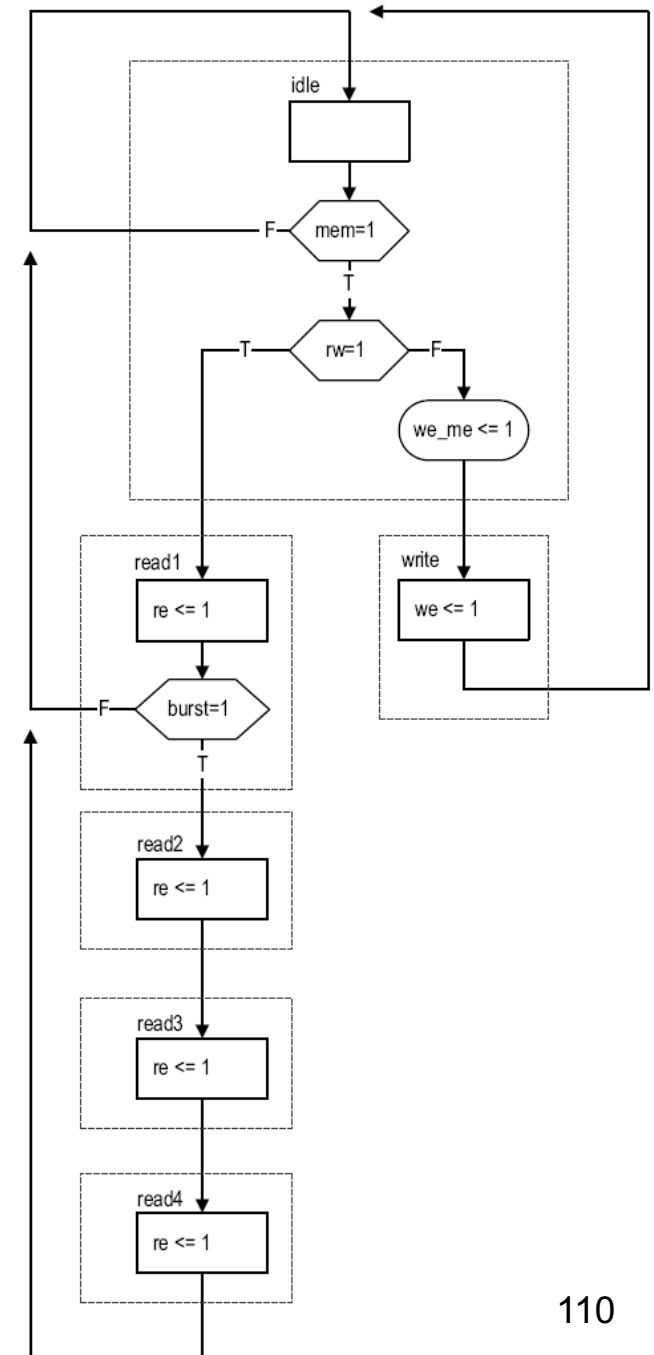
```
            oe <= '1';
```

```
        when read4 =>
```

```
            oe <= '1';
```

```
    end case;
```

```
end process;
```



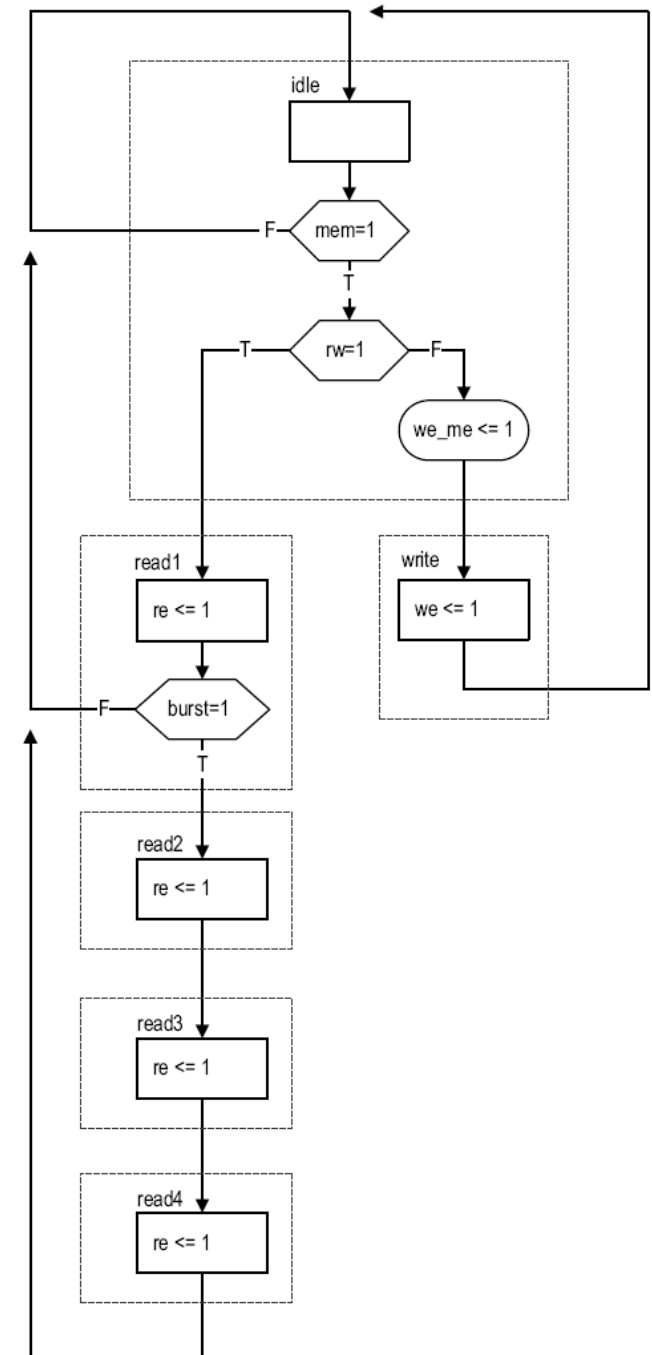
Mealy output: **we_me** that depends on **mem** and **rw**.

— *mealy output logic*

```

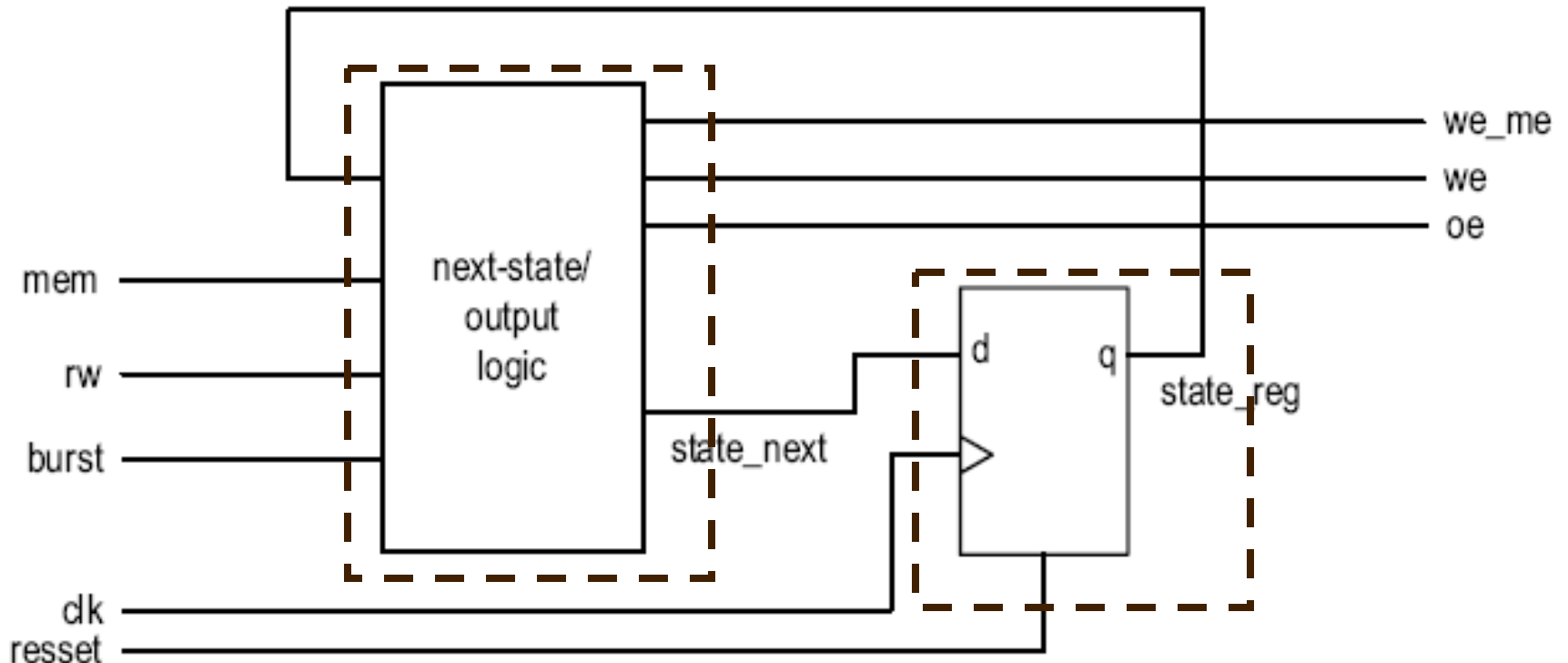
process(state_reg, mem, rw)
begin
    we_me <= '0'; — default value
    case state_reg is
        when idle =>
            if (mem='1') and (rw='0') then
                we_me <= '1';
            end if;
        when write =>
        when read1 =>
        when read2 =>
        when read3 =>
        when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

```



Alternative Coding Style 2

Process(Present State,Inputs)



Process(clk, reset)


```

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
  -- 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;
end mult_seg_arch;

```

```

process(state_reg, mem, rw, burst)
begin
    oe <= '0';      — default values
    we <= '0';
    we_me <= '0';
    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';
    end case;
end process;

```

```

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

```

VHDL Variables

```
entity variable_in_process is
    port (
        A,B      : in  std_logic_vector (3 downto 0);
        ADD_SUB   : in  std_logic;
        S         : out std_logic_vector (3 downto 0) );
end variable_in_process;
```

```
architecture archi of variable_in_process is
begin
```

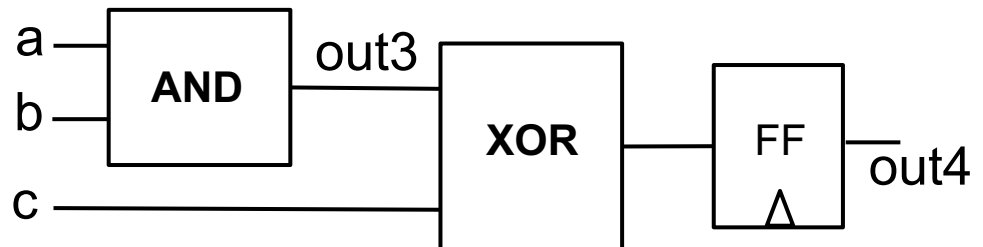
```
    process (A, B, ADD_SUB)
        variable AUX : std_logic_vector (3 downto 0);
    begin
        if ADD_SUB = '1' then
            AUX := A + B ;
        else
            AUX := A - B ;
        end if;
        S <= AUX;
    end process;
end archi;
```

Differences: Signals vs Variables

- Variables can only be declared and used within processes or procedures.
 - Used to hold temporary results.
- Signals can only be declared in architecture.
 - Used for inter-process communications.
- Variables are updated immediately.
- Signals are updated after current execution of a process is finished.
- Synthesis results:
 - Variables: wires or nothing
 - Signals: wires, registers, or latches.

Differences: Signals vs Variables

```
architecture var_ex of test is
begin
  process (clk)
    variable out3 : std_logic;
  begin
    if rising_edge(clk) then
      out3 := a and b;
      out4 <= out3 xor c;
    end if;
  end process;
end var_ex;
```



Differences: Signals vs Variables

architecture sig_ex of test is

```
    signal out1, out2 : std_logic;
```

```
begin
```

```
    process (clk)
```

```
    begin
```

```
        if rising_edge(clk) then
```

```
            out1 <= a and b;
```

```
            out2 <= out1 xor c;
```

```
        end if;
```

```
    end process;
```

```
end sig_ex;
```

