ECE 545
Lecture 9

Design of Controllers

Finite State Machines and Algorithmic State Machine (ASM) Charts

Required reading

- P. Chu, *RTL Hardware Design using VHDL*

Chapter 10, Finite State Machine: Principle & Practice

Chapter 11, Register Transfer Methodology: Principle

Recommended reading

- P. Chu, *RTL Hardware Design using VHDL*

Chapter 12, Register Transfer Methodology: Practice

Slides based partially on

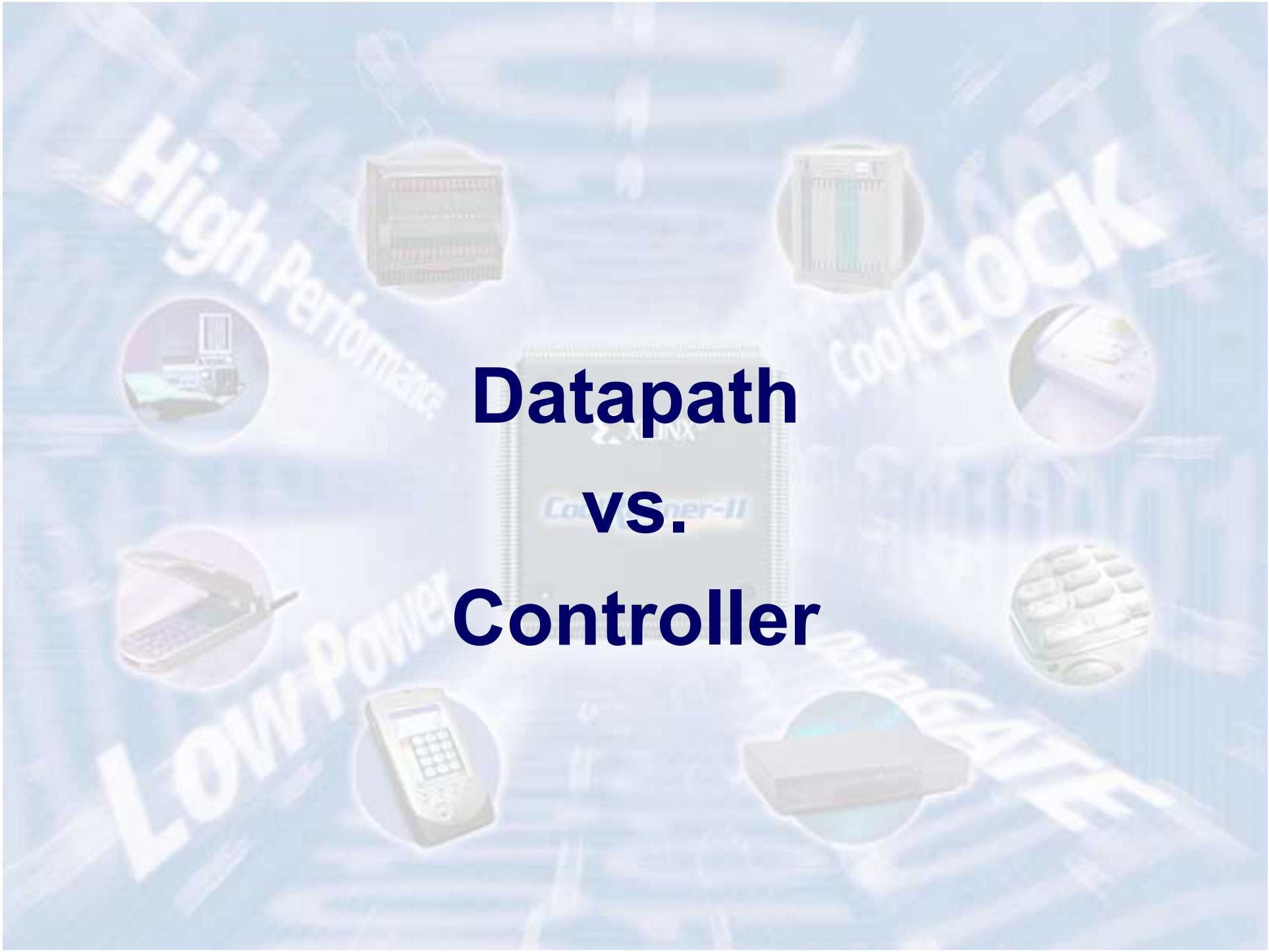
- S. Brown and Z. Vranesic,
  Fundamentals of Digital Logic with VHDL Design

  Chapter 8, Synchronous Sequential Circuits

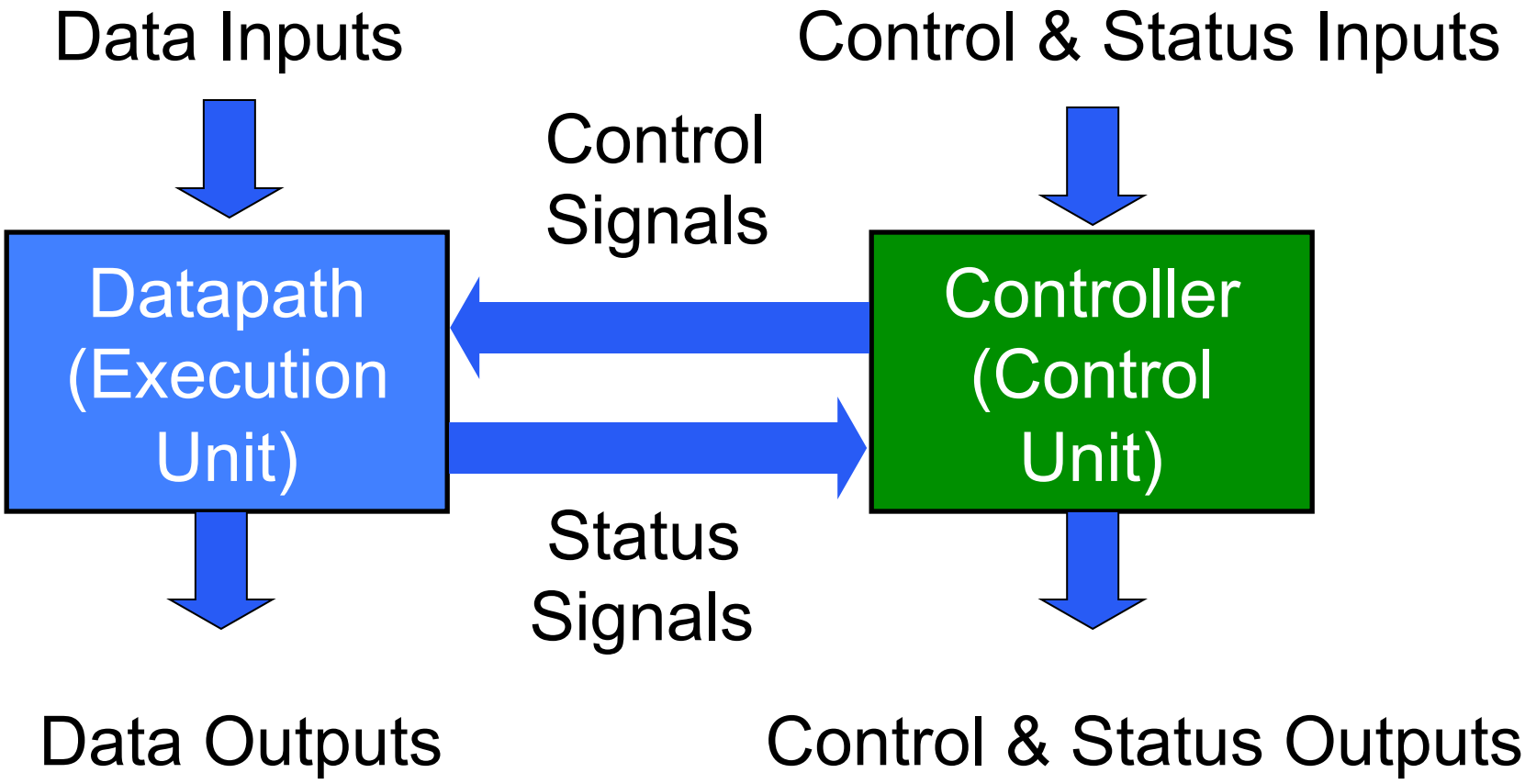
  Sections 8.1-8.5

  Chapter 8.10, Algorithmic State Machine (ASM) Charts

  Chapter 10.2 Design Examples


High Performance
Low Power
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 movements 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
- Determines the sequence of operations performed by the Datapath
- Follows Some ‘Program’ or Schedule

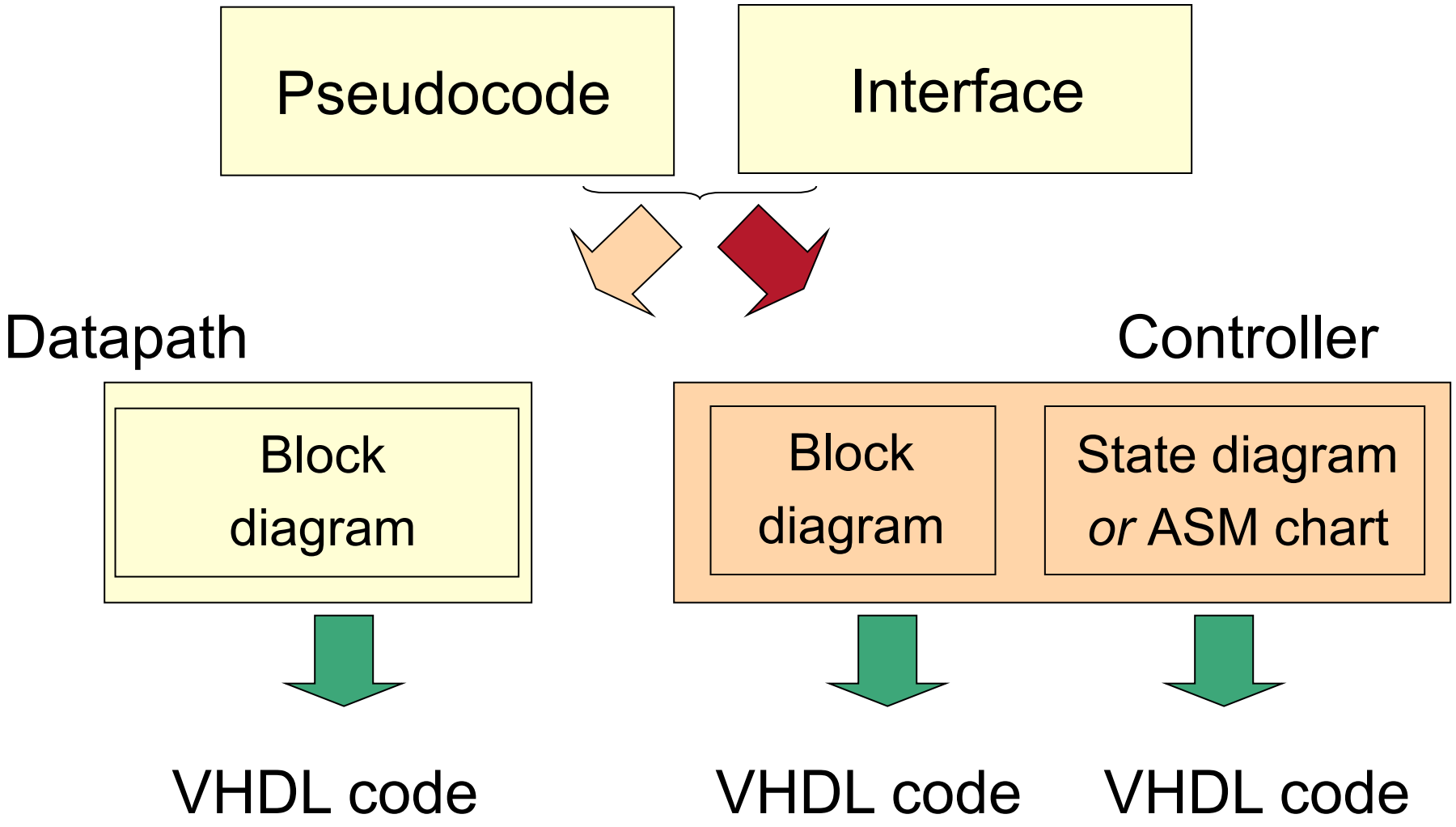
Programmable vs. Non-Programmable Controller

- Controller can be programmable or non-programmable
- Programmable
  - Has a program counter which points to next instruction
  - Instructions are held 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

- Digital Systems and especially their Controllers can be described as Finite State Machines (FSMs)
- Finite State Machines can be represented using
  - **State Diagrams and State Tables** - suitable for simple digital systems with a relatively few inputs and outputs
  - **Algorithmic State Machine (ASM) Charts** - suitable for complex digital systems with a large number of inputs and outputs
- All these descriptions can be easily translated to the corresponding synthesizable VHDL code

Hardware Design with RTL VHDL



Steps of the Design Process

1. Text description
2. Interface
3. Pseudocode
4. Block diagram of the Datapath
5. Interface with the division into Datapath and Controller
6. ASM chart of the Controller
7. RTL VHDL code of the Datapath, Controller, and Top-Level Unit
8. Testbench of the Datapath, Controller, and Top-Level Unit
9. Functional simulation and debugging
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing

Steps of the Design Process Introduced in Class Today

1. Text description
2. Interface
3. Pseudocode
4. Block diagram of the Datapath
5. Interface with the division into Datapath and Controller
6. **ASM chart of the Controller**
7. **RTL VHDL code of** the Datapath, **Controller**, and Top-Level Unit
8. **Testbench of** the Datapath, **Controller**, and Top-Level Unit
9. Functional simulation and debugging
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing



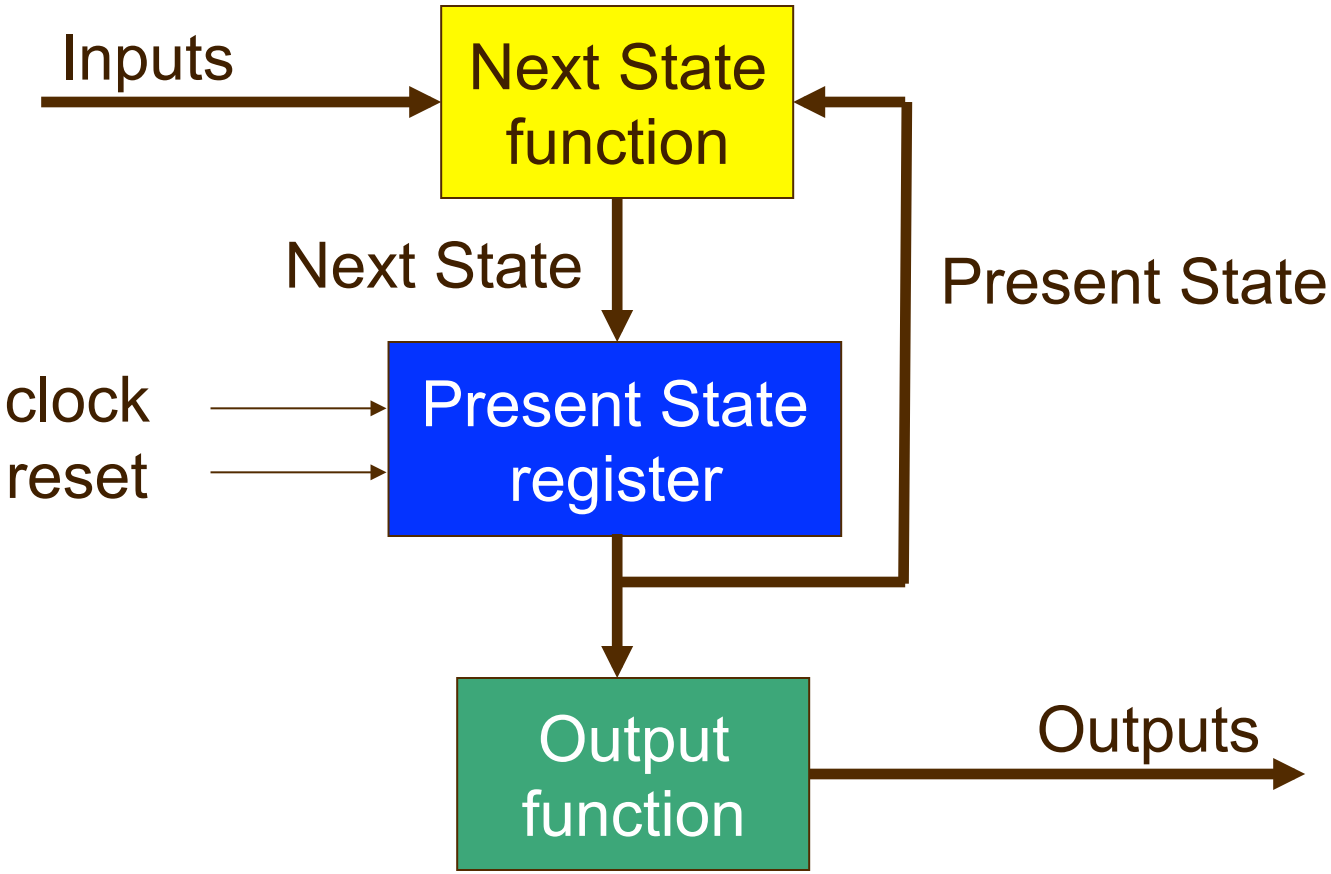
Finite State Machines Refresher

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 of a medium to large digital system
- Design of FSMs involves
  - Defining states
  - Defining next state and output functions
  - Optimization / minimization
- Manual optimization/minimization is practical for small FSMs only

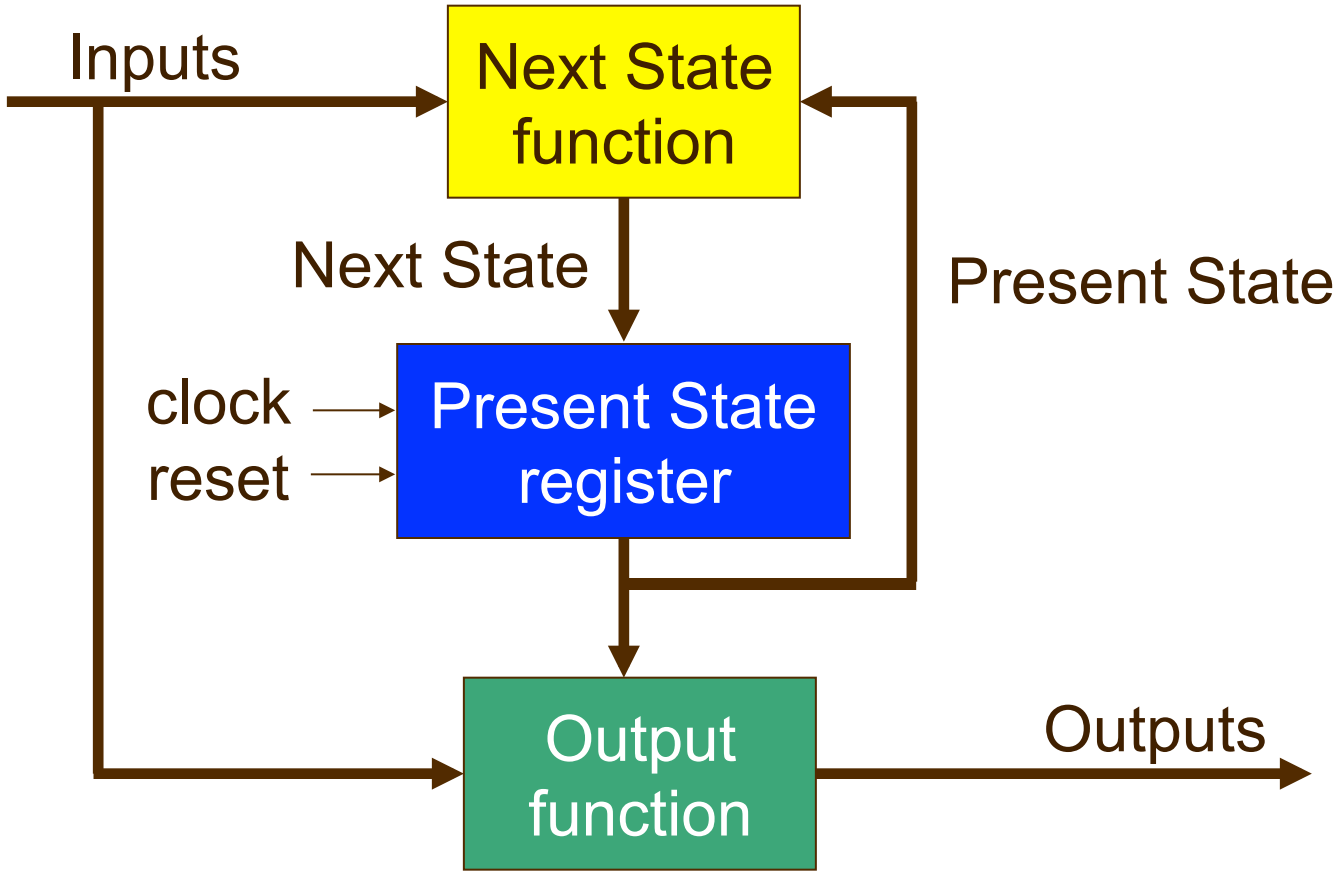
Moore FSM

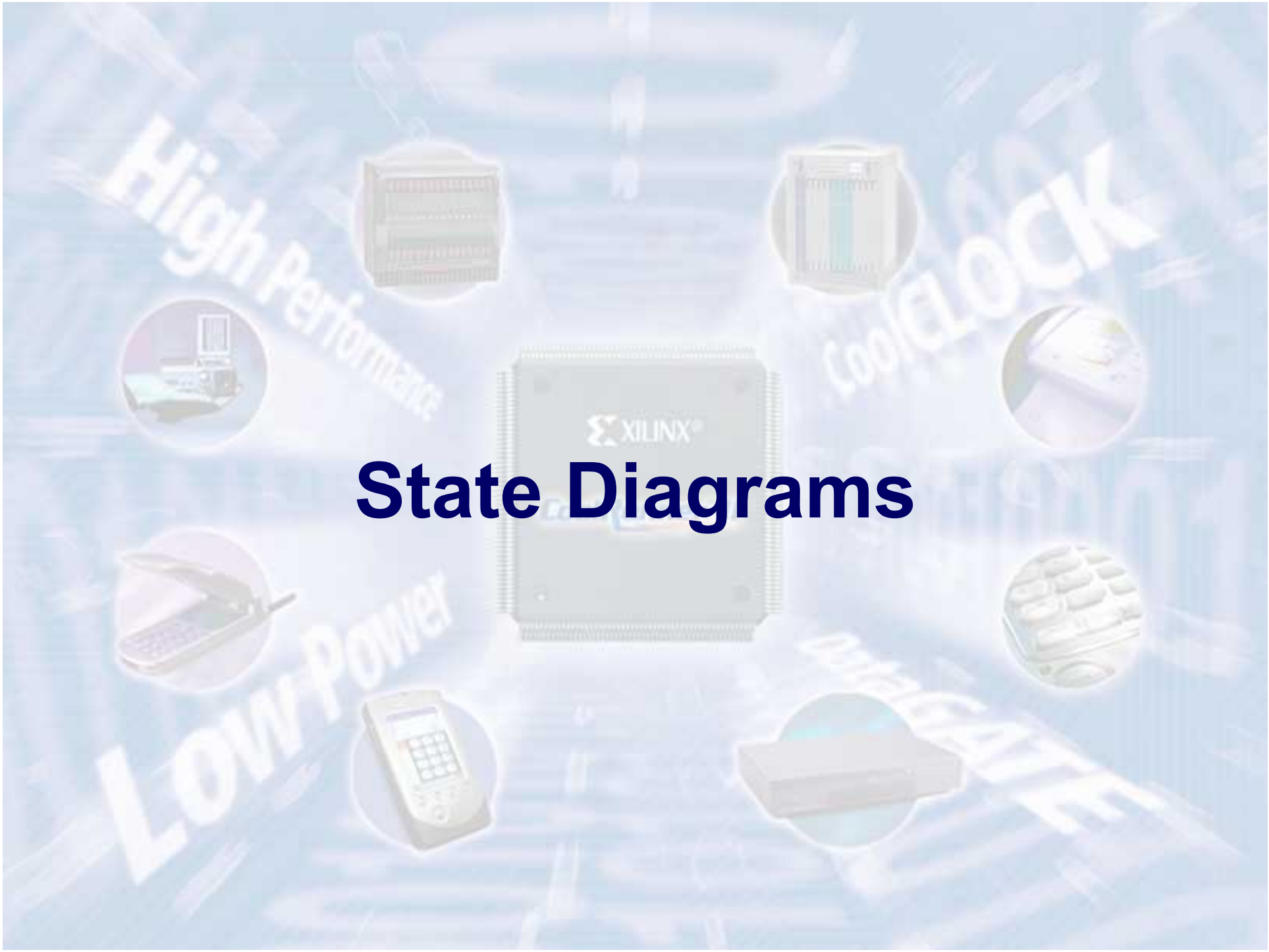
- Output Is a Function of a Present State Only



Mealy FSM

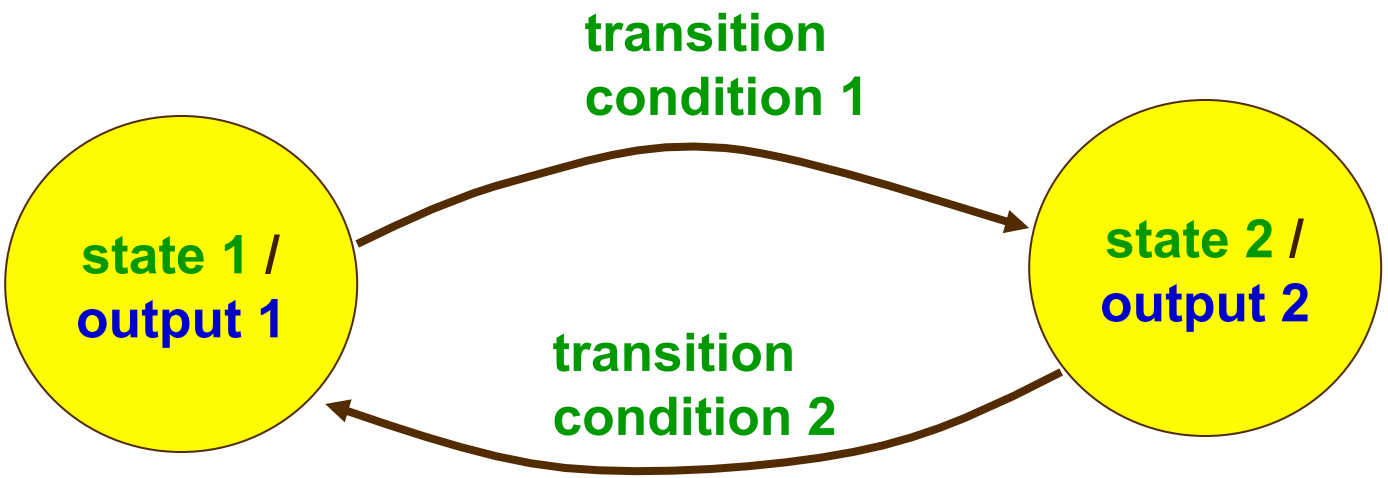
- Output Is a Function of a Present State and Inputs



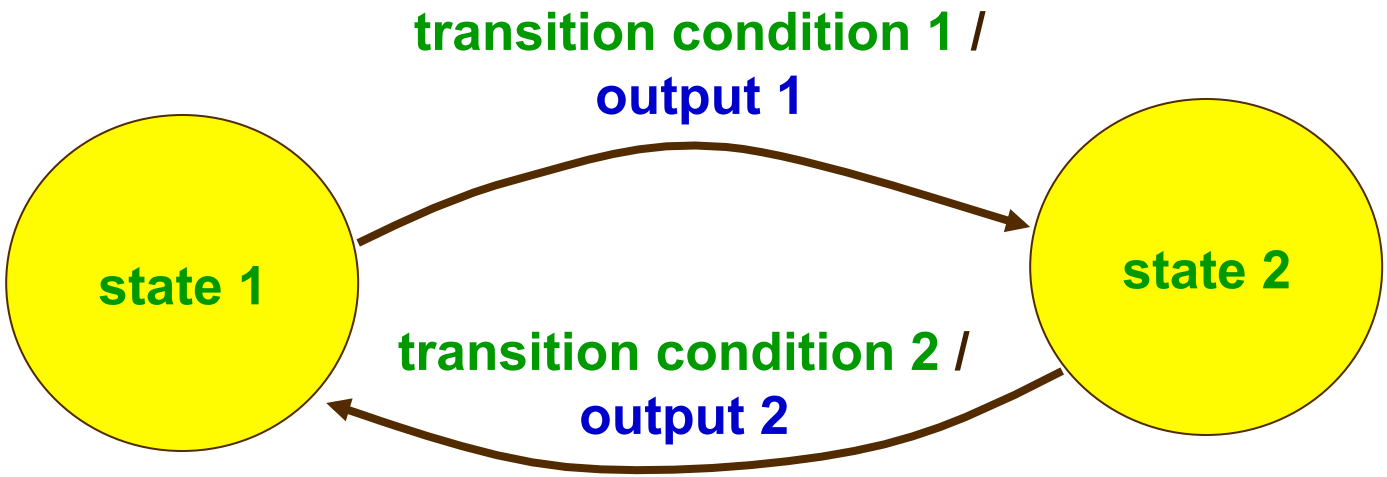


State Diagrams

Moore Machine

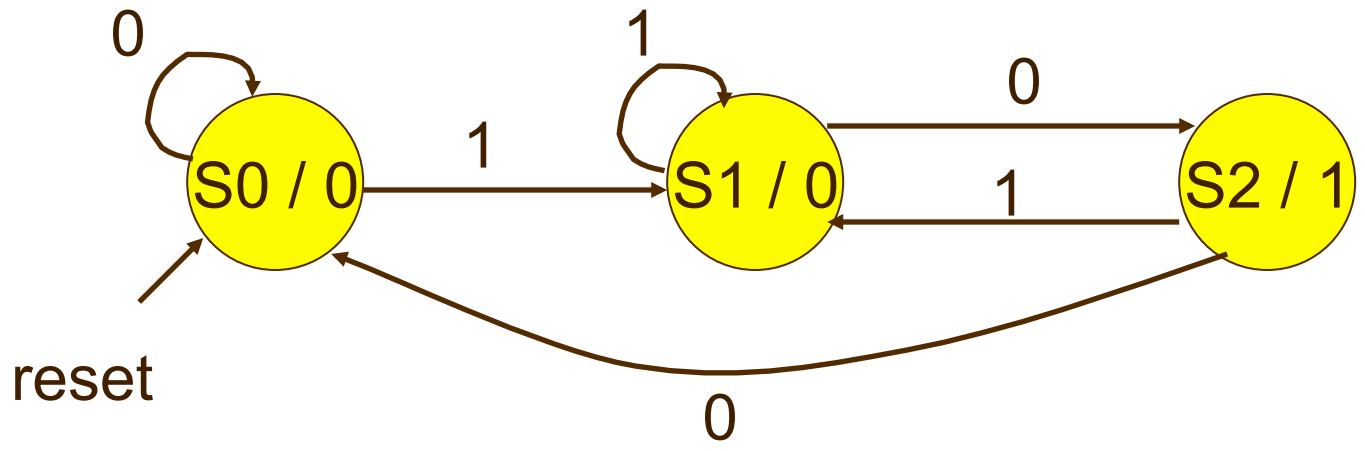


Mealy Machine



Moore FSM - Example 1

- Moore FSM that Recognizes Sequence “10”



Meaning of states:

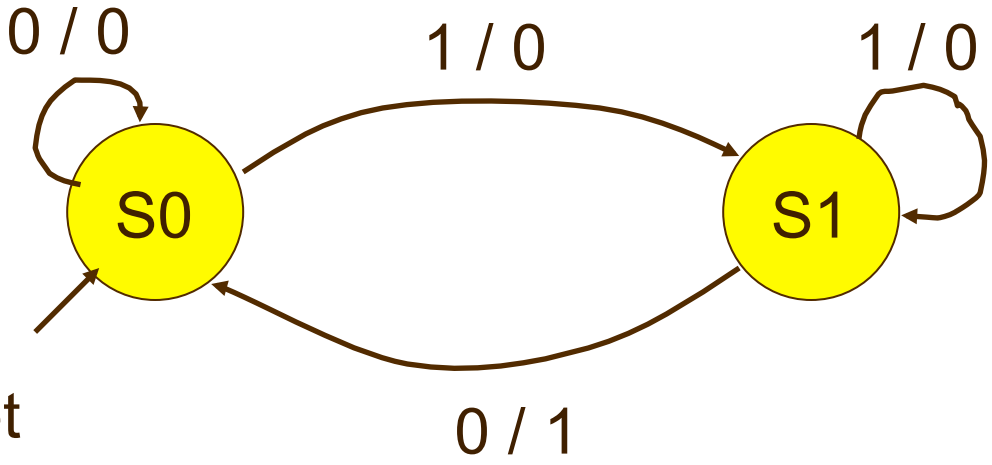
S0: No elements of the sequence observed

S1: “1” observed

S2: “10” observed

Mealy FSM - Example 1

- Mealy FSM that Recognizes Sequence "10"

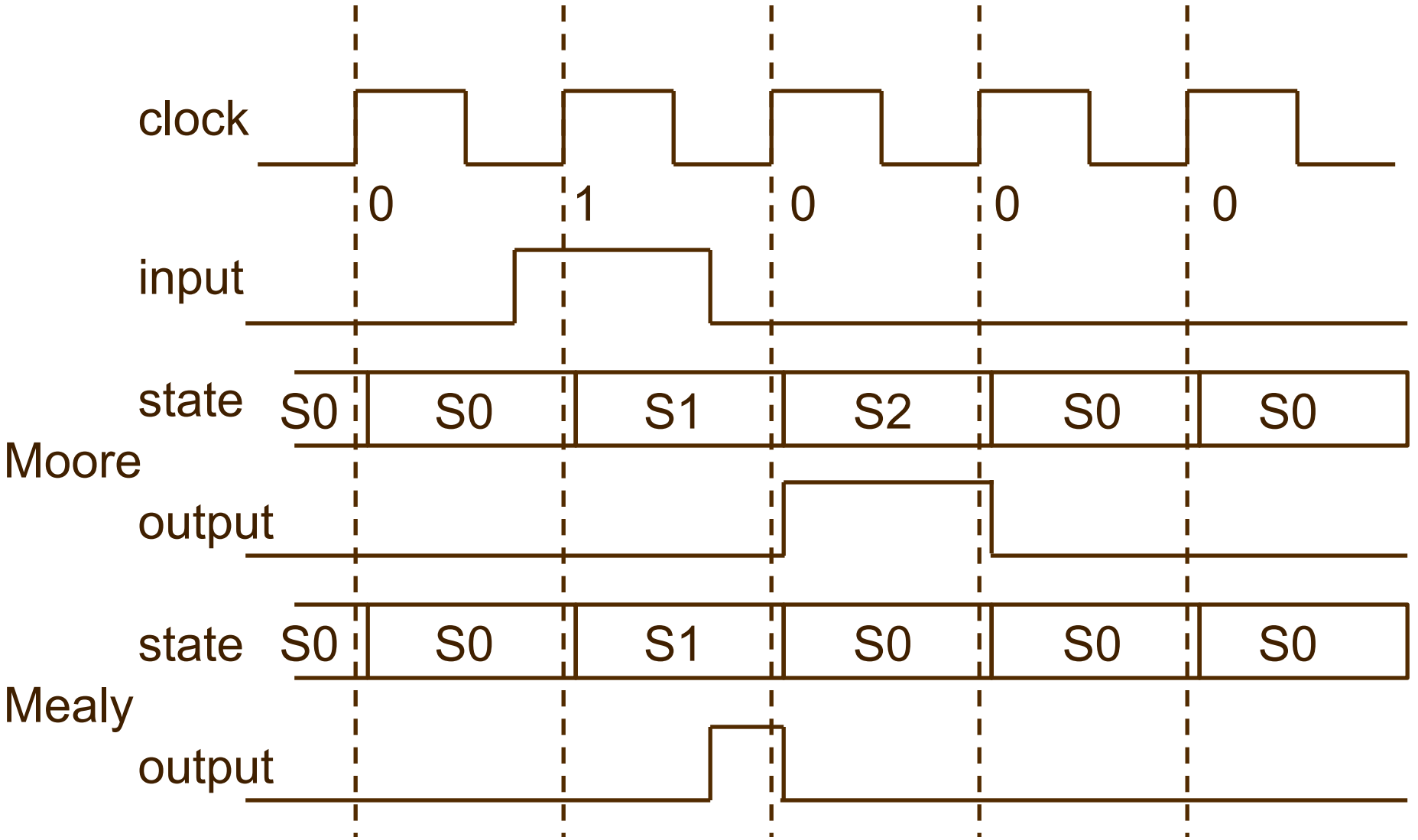


Meaning of states:

S0: No elements of the sequence observed

S1: "1" observed

Moore & Mealy FSMs – Example 1



Moore vs. Mealy FSM (1)

- Moore and Mealy FSMs Can Be Functionally Equivalent
  - Equivalent Mealy FSM can be derived from Moore FSM and vice versa
- Mealy FSM Has Richer Description and Usually Requires Smaller Number of States
  - Smaller circuit area

Moore vs. Mealy FSM (2)

- Mealy FSM Computes Outputs as soon as Inputs Change
  - Mealy FSM responds one clock cycle sooner than equivalent Moore FSM
- Moore FSM Has No Combinational Path Between Inputs and Outputs
  - Moore FSM is 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

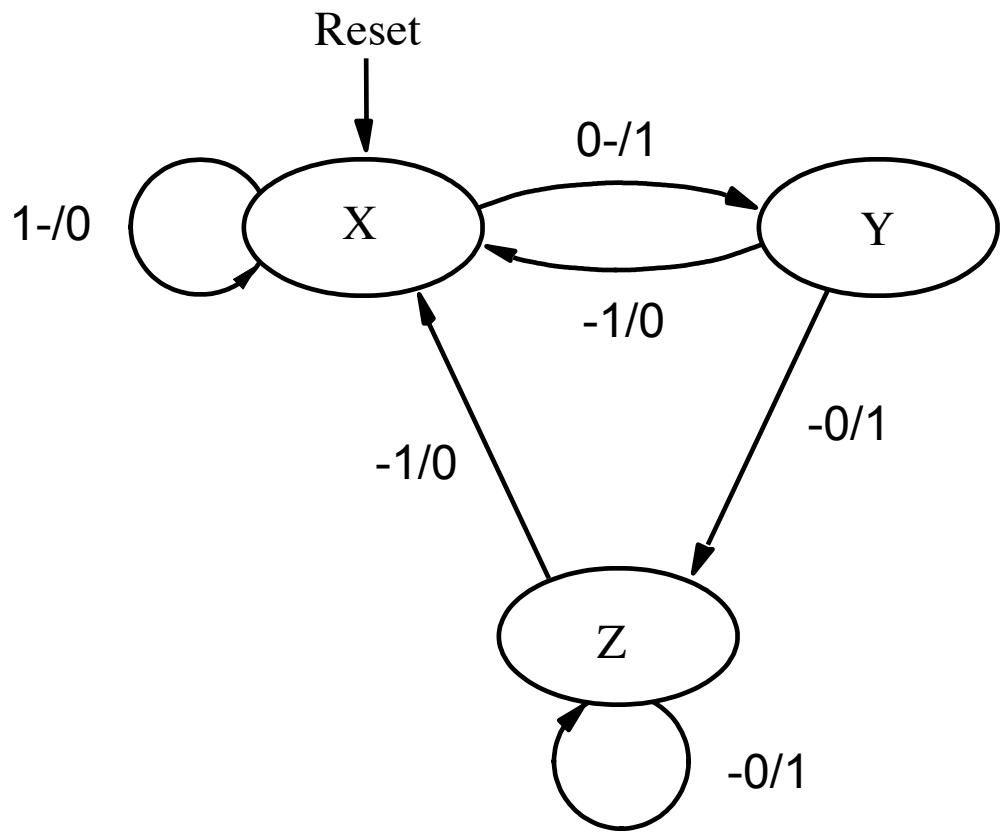
Moore FSM

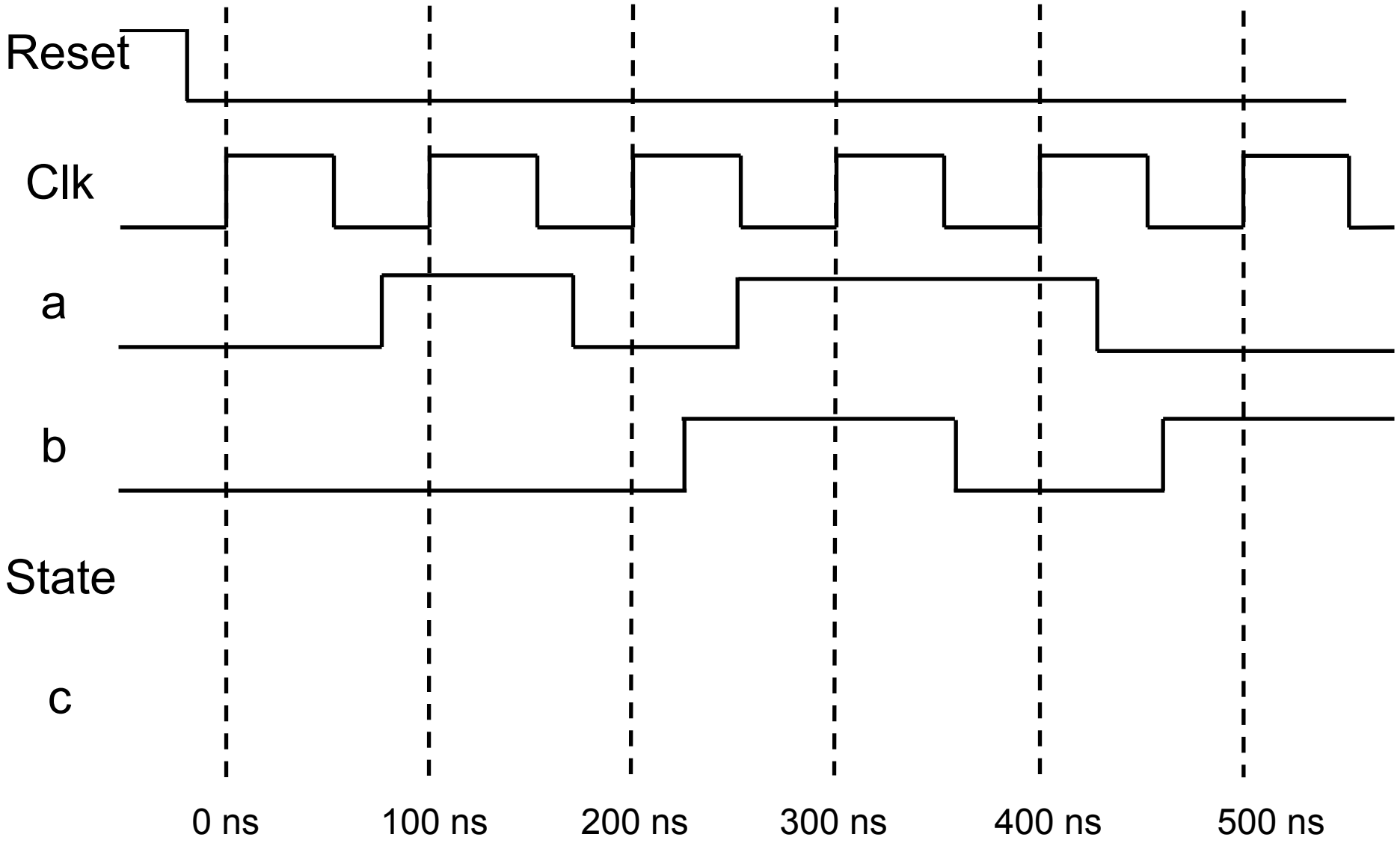
Safer.
Less likely to affect the critical path.



Problem 1

Assuming state diagram given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals **State** and **c**, in the interval from 0 to 575 ns.


Reset
0-/1
1-/0
X
Y
-1/0
-1/0
-0/1
Z
-0/1


Reset
Clk
a
b
State
c
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns



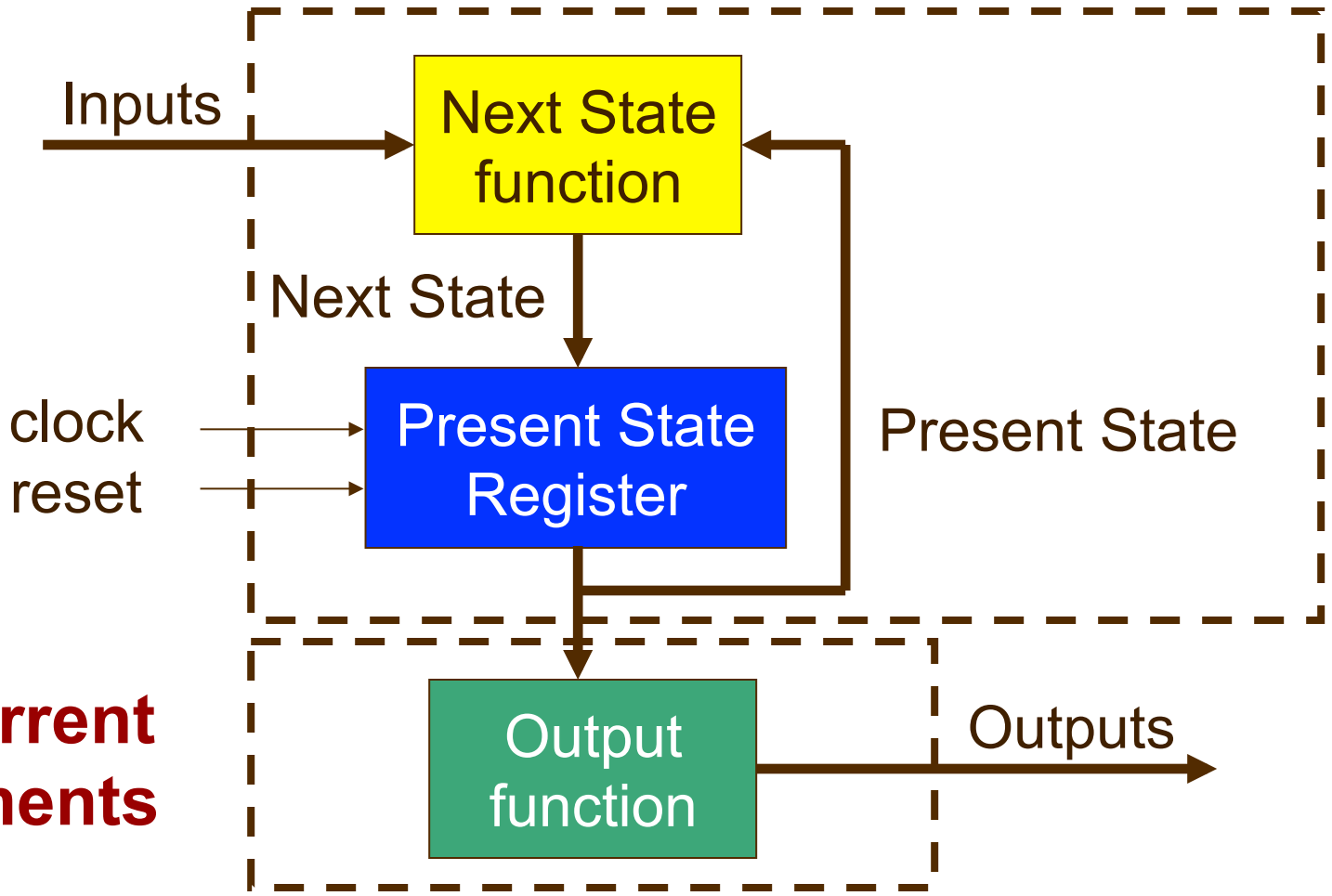
Finite State Machines in VHDL

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 in the sensitivity list

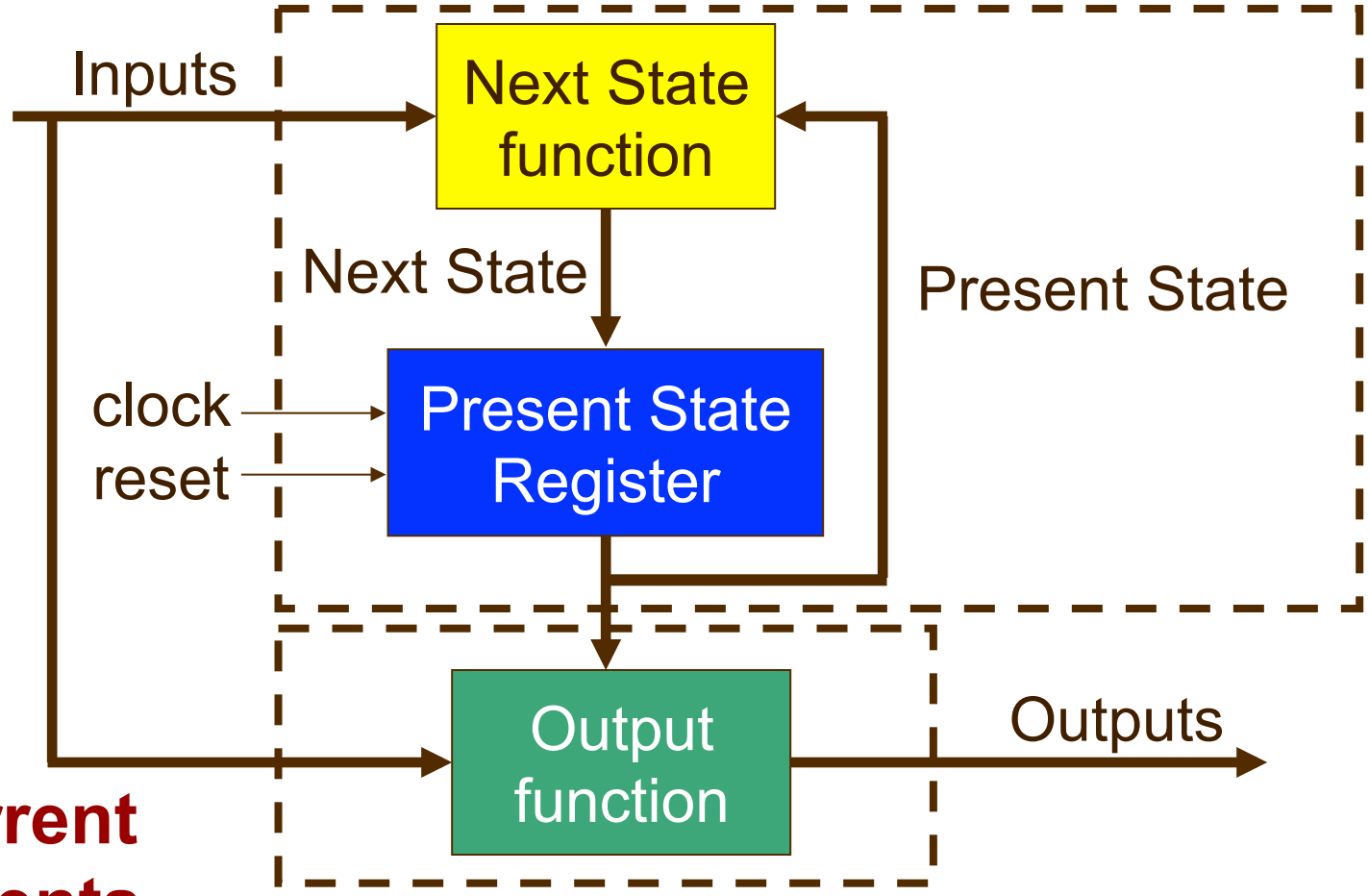
Moore FSM

process(clock, reset)



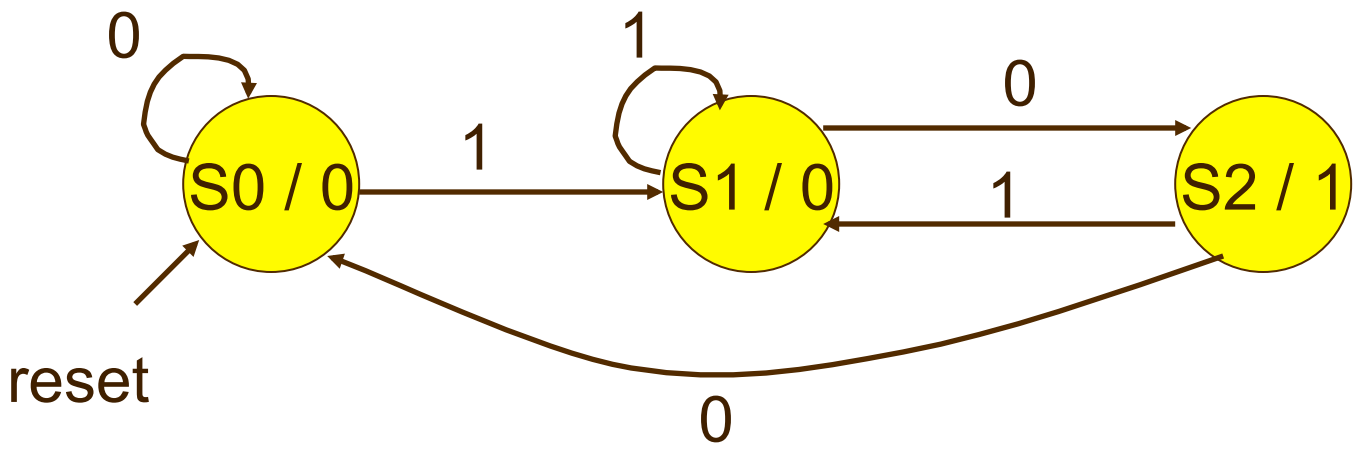
Mealy FSM

process(clock, reset)



Moore FSM - Example 1

- Moore FSM that Recognizes Sequence “10”



Moore FSM in VHDL (1)

```
TYPE state IS (S0, S1, S2);
SIGNAL Moore_state: state;

U_Moore: PROCESS (clock, reset)
BEGIN
      IF(reset = '1') THEN
         Moore_state <= S0;
      ELSIF (clock = '1' AND clock'event) THEN
         CASE Moore_state IS
            WHEN S0 =>
             IF input = '1' THEN
                Moore_state <= S1;
             ELSE
               Moore_state <= S0;
             END IF;
```

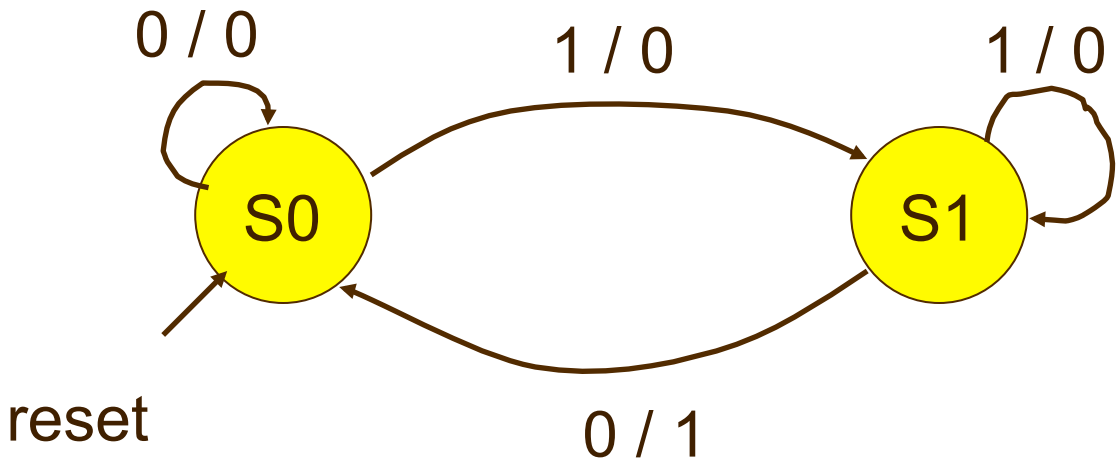
Moore FSM in VHDL (2)

```
            WHEN S1 =>
                IF input = '0' THEN
                    Moore_state <= S2;
                ELSE
                    Moore_state <= S1;
                END IF;
            WHEN S2 =>
                IF input = '0' THEN
                    Moore_state <= S0;
                ELSE
                    Moore_state <= S1;
                END IF;
        END CASE;
    END IF;
END PROCESS;

Output <= '1' WHEN Moore_state = S2 ELSE '0';
```

Mealy FSM - Example 1

- Mealy FSM that Recognizes Sequence "10"



Mealy FSM in VHDL (1)

```
TYPE state IS (S0, S1);
SIGNAL Mealy_state: state;

U_Mealy: PROCESS(clock, reset)
BEGIN
      IF(reset = '1') THEN
         Mealy_state <= S0;
      ELSIF (clock = '1' AND clock'event) THEN
         CASE Mealy_state IS
               WHEN S0 =>
                  IF input = '1' THEN
                      Mealy_state <= S1;
                  ELSE
                      Mealy_state <= S0;
                  END IF;
```

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 <= '1' WHEN (Mealy_state = S1 AND input = '0') ELSE '0';
```



Algorithmic State Machine (ASM) Charts

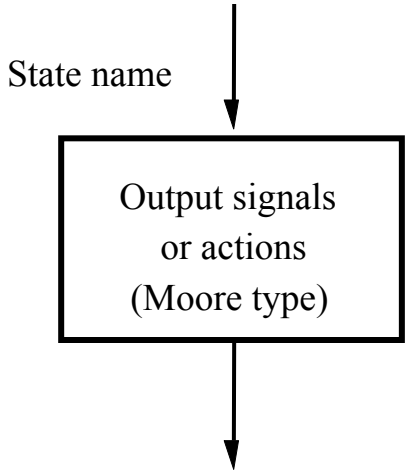
Algorithmic State Machine

Algorithmic State Machine –

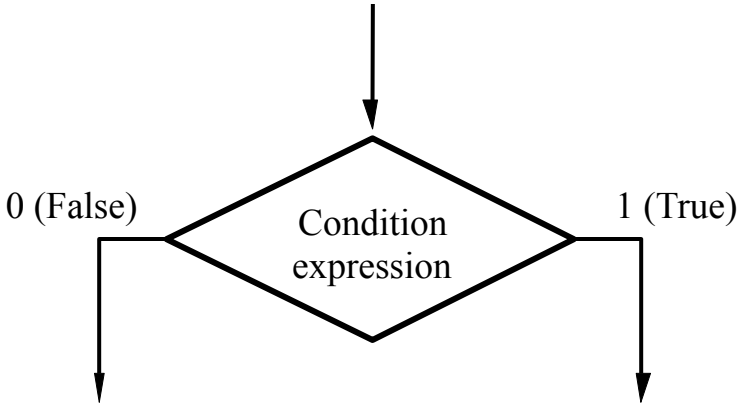
representation of a Finite State Machine

suitable for FSMs with a larger number of inputs and outputs compared to FSMs expressed using state diagrams and state tables.

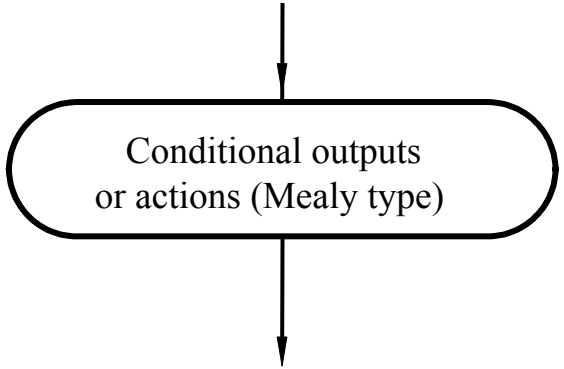
Elements used in ASM charts (1)



(a) State box



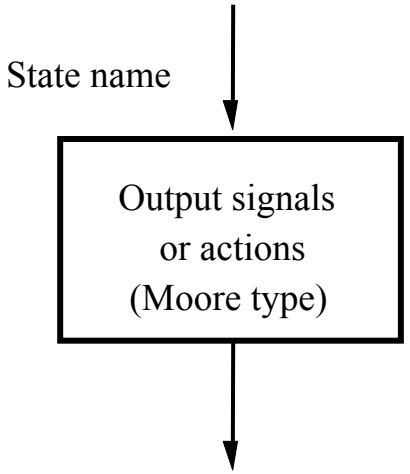
(b) Decision box



(c) Conditional output box

State Box

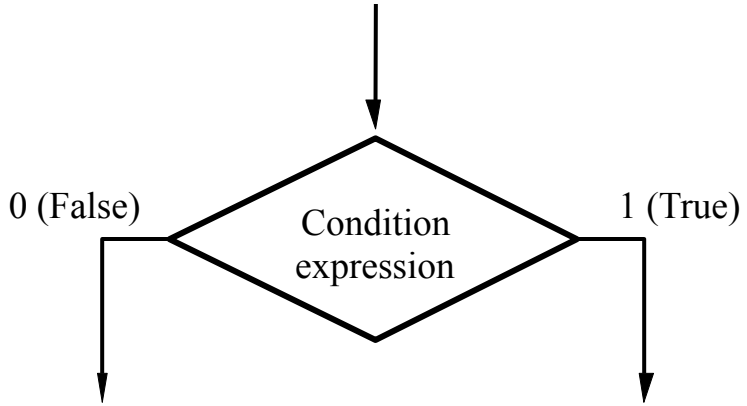
- **State box** – represents a state.
- Equivalent to a node in a state diagram or a row in a state table.
- Contains register transfer actions or output signals
- **Moore-type outputs are listed inside of the box.**
- It is customary to write only the name of the signal that has to be asserted in the given state, e.g., z instead of z<=1.
- Also, it might be useful to write an action to be taken, e.g., count <= 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

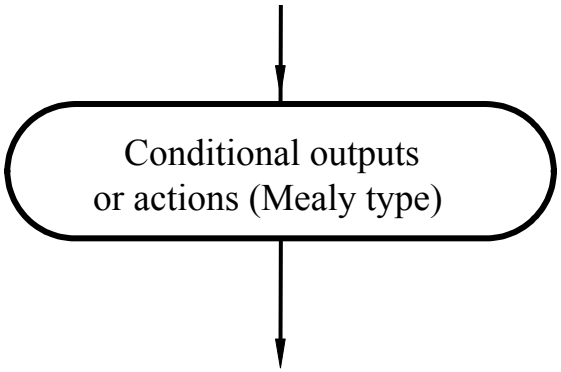
- **Decision box** – indicates that a given condition is to be tested and the exit path is to be chosen accordingly.

  The condition expression may include one or more inputs to the FSM.



Conditional Output Box

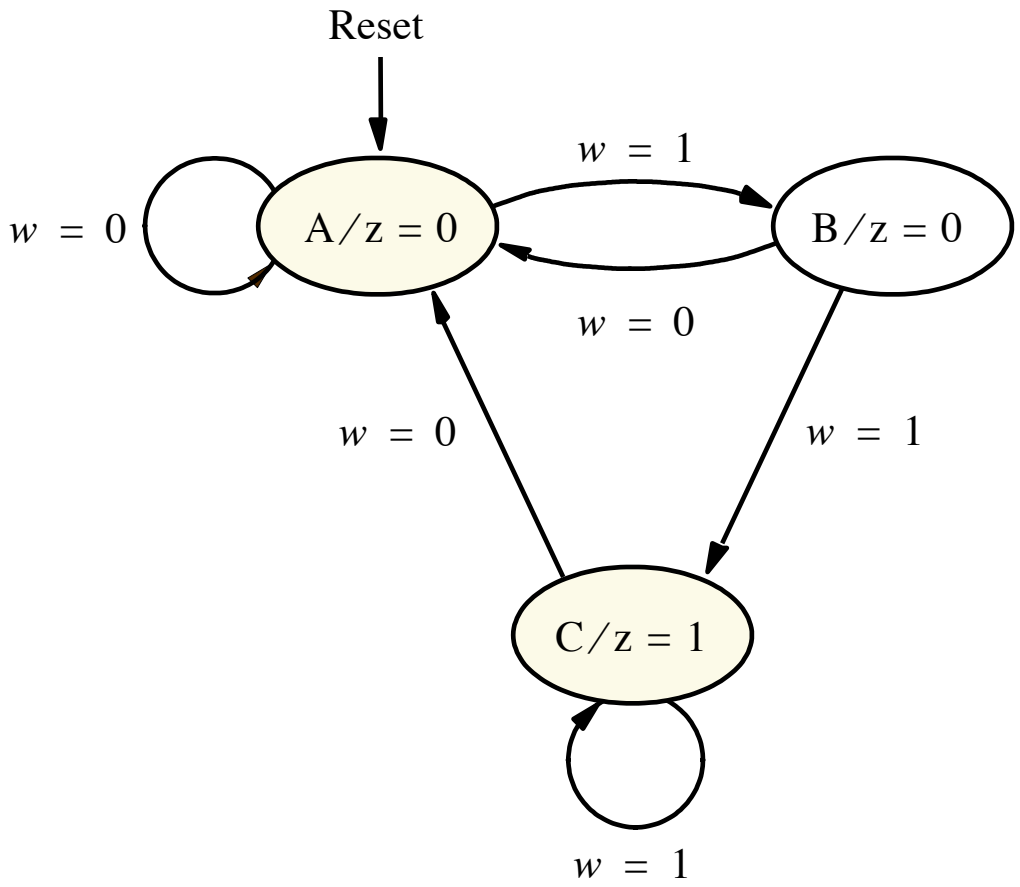
- **Conditional output box**
- **Denotes output signals that are of the Mealy type.**
- The condition that determines whether such outputs are generated is specified in the decision box.



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

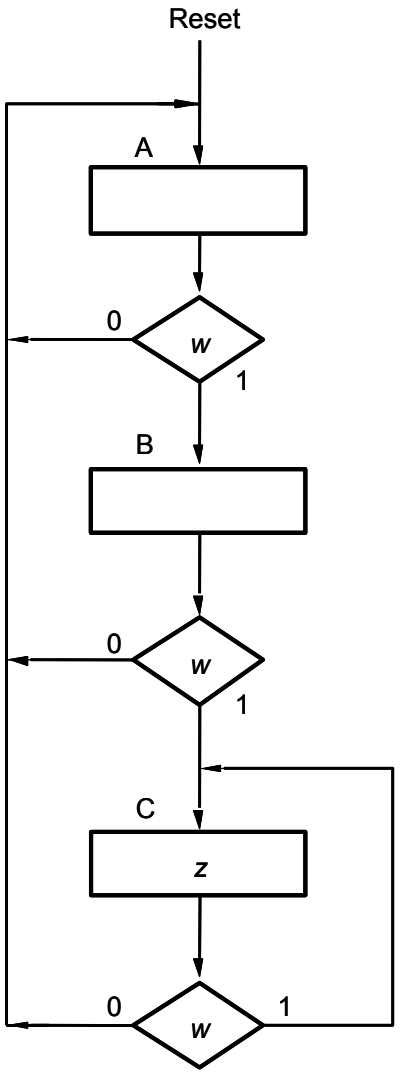
Moore FSM – Example 2: State diagram



Moore FSM – Example 2: State table

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

ASM Chart for Moore FSM – Example 2



Example 2: VHDL code (1)

```
USE ieee.std_logic_1164.all ;

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 y : State_type ;
BEGIN
      PROCESS ( resetn, clock )
      BEGIN
            IF resetn = '0' THEN
                  y <= A ;
            ELSIF (Clock'EVENT AND Clock = '1') THEN
```

Example 2: VHDL code (2)

```
CASE y IS
    WHEN A =>
        IF w = '0' THEN
            y <= A ;
        ELSE
            y <= B ;
        END IF ;
    WHEN B =>
        IF w = '0' THEN
            y <= A ;
        ELSE
            y <= C ;
        END IF ;
    WHEN C =>
        IF w = '0' THEN
            y <= A ;
        ELSE
            y <= C ;
        END IF ;
END CASE ;
```

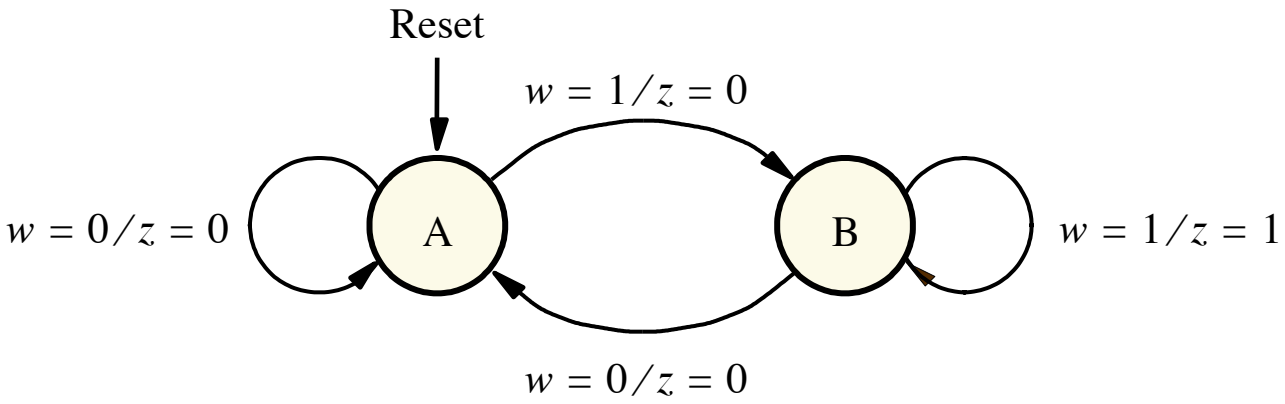
Example 2: VHDL code (3)

```
        END IF ;
    END PROCESS ;

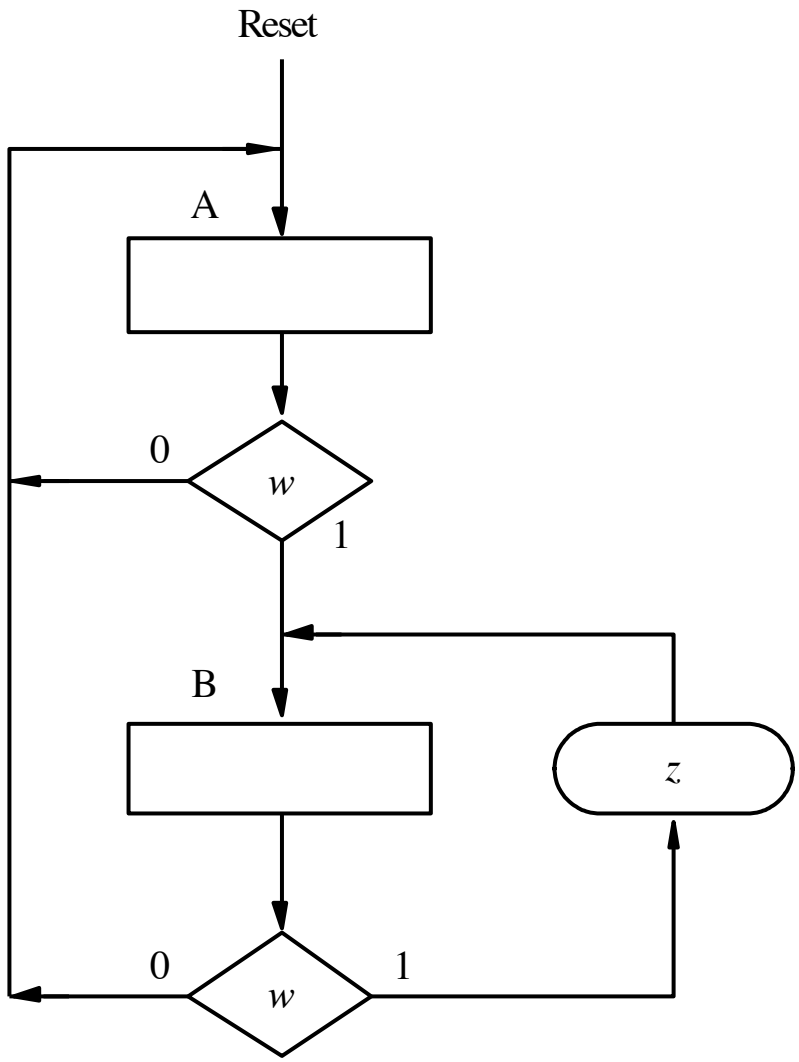
    z <= '1' WHEN y = C ELSE '0' ;

END Behavior ;
```

Mealy FSM – Example 3: State diagram



ASM Chart for Mealy FSM – Example 3



Example 3: VHDL code (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

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 y : State_type ;
BEGIN
      PROCESS ( resetn, clock )
      BEGIN
          IF resetn = '0' THEN
              y <= A ;
          ELSIF (clock'EVENT AND clock = '1') THEN
```

Example 3: VHDL code (2)

```
CASE y IS
    WHEN A =>
        IF w = '0' THEN
            y <= A ;
        ELSE
            y <= B ;
        END IF ;
    WHEN B =>
        IF w = '0' THEN
            y <= A ;
        ELSE
            y <= B ;
        END IF ;
END CASE ;
```

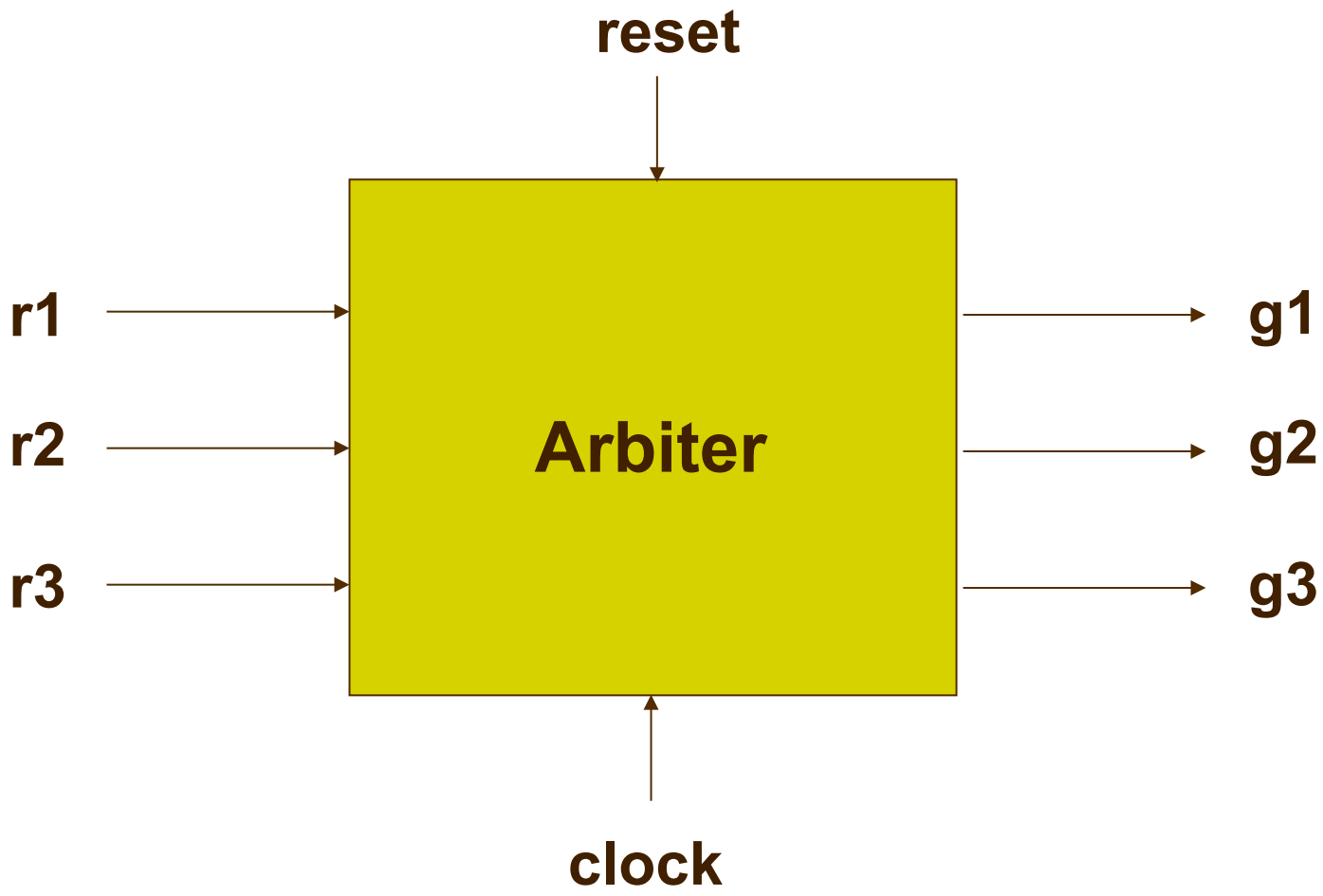
Example 3: VHDL code (3)

```
        END IF ;
    END PROCESS ;

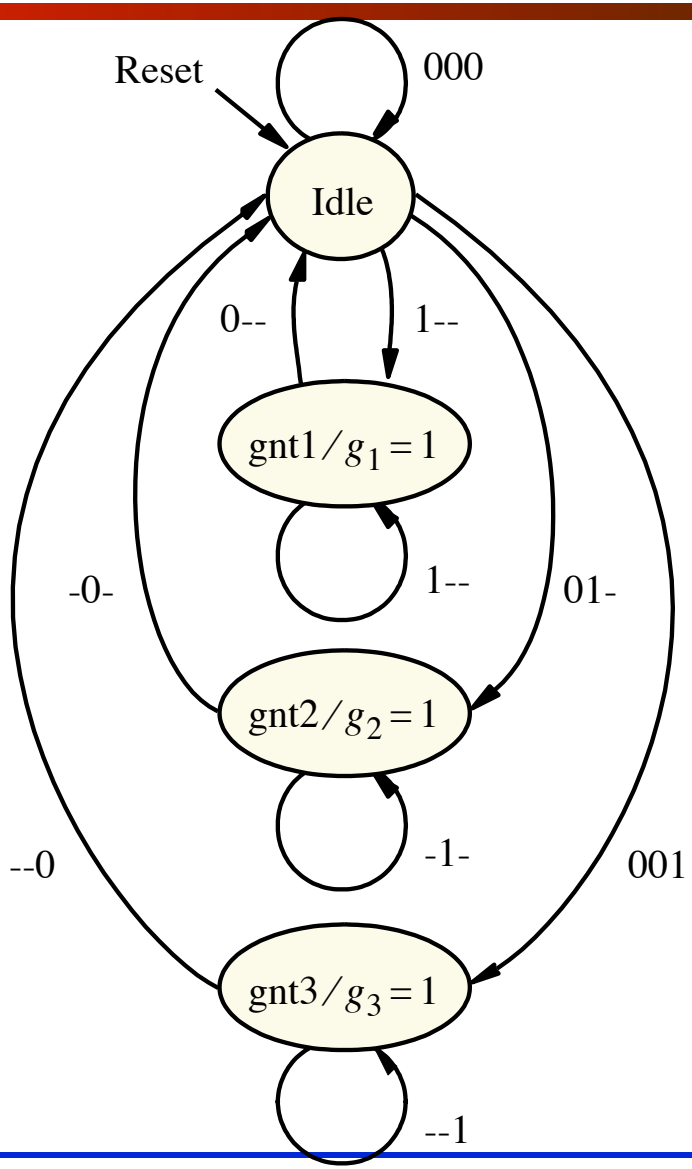
    z <= '1' WHEN (y = B) AND (w='1') ELSE '0' ;

END Behavior ;
```

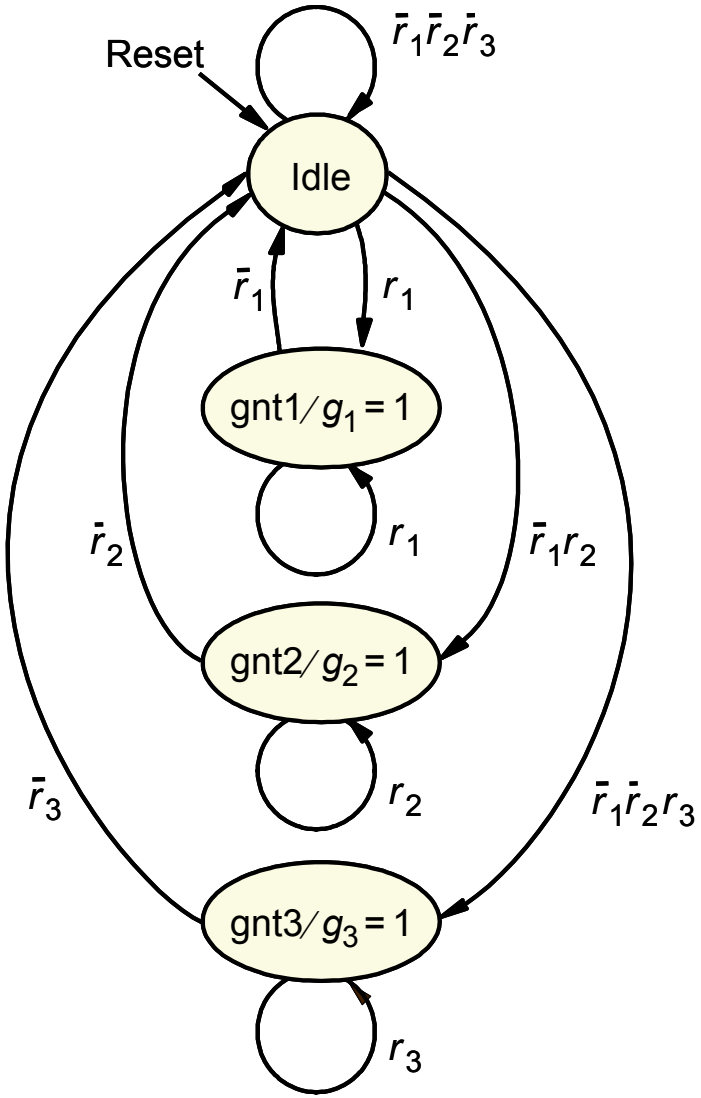
Control Unit Example: Arbiter (1)



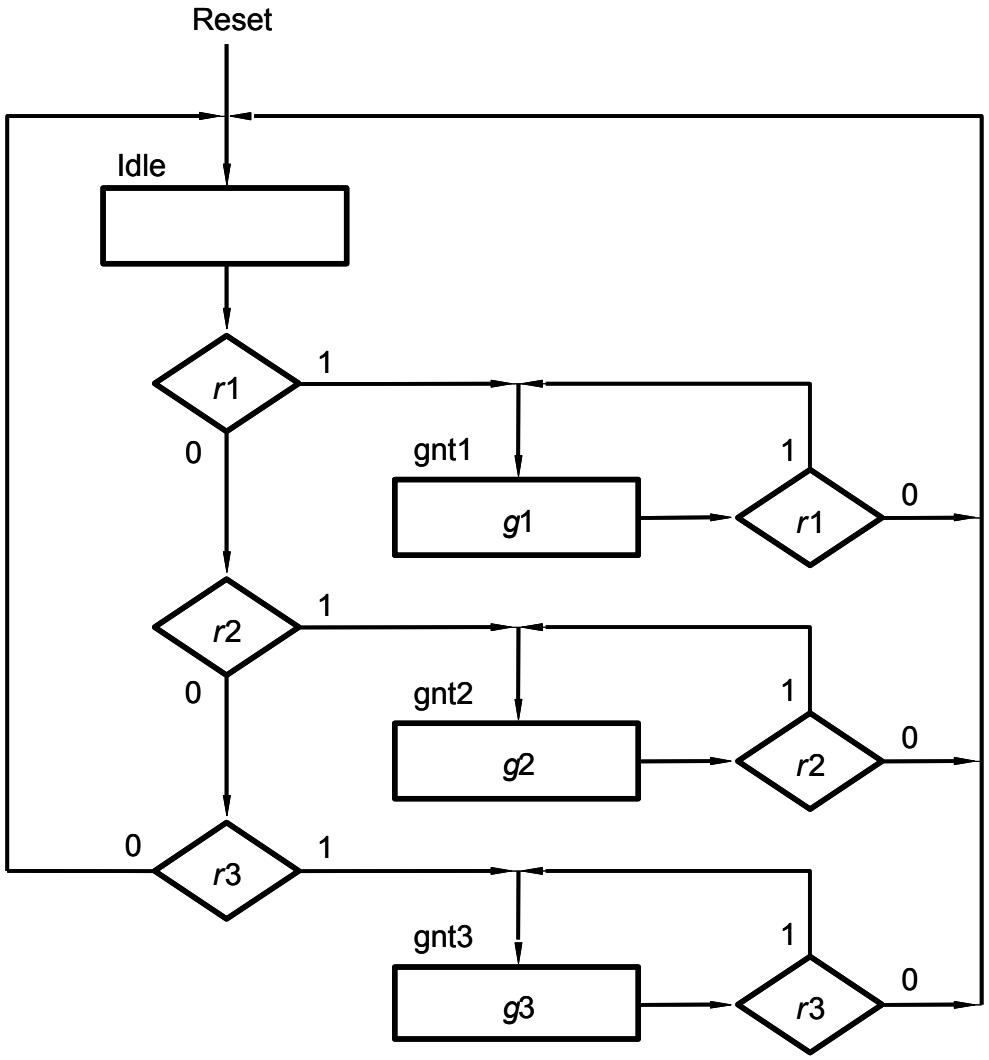
Control Unit Example: Arbiter (2)



Control Unit Example: Arbiter (3)



ASM Chart for Control Unit - Example 4



Example 4: VHDL code (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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 y : State_type ;
```

Example 4: VHDL code (2)

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

Example 4: VHDL code (3)

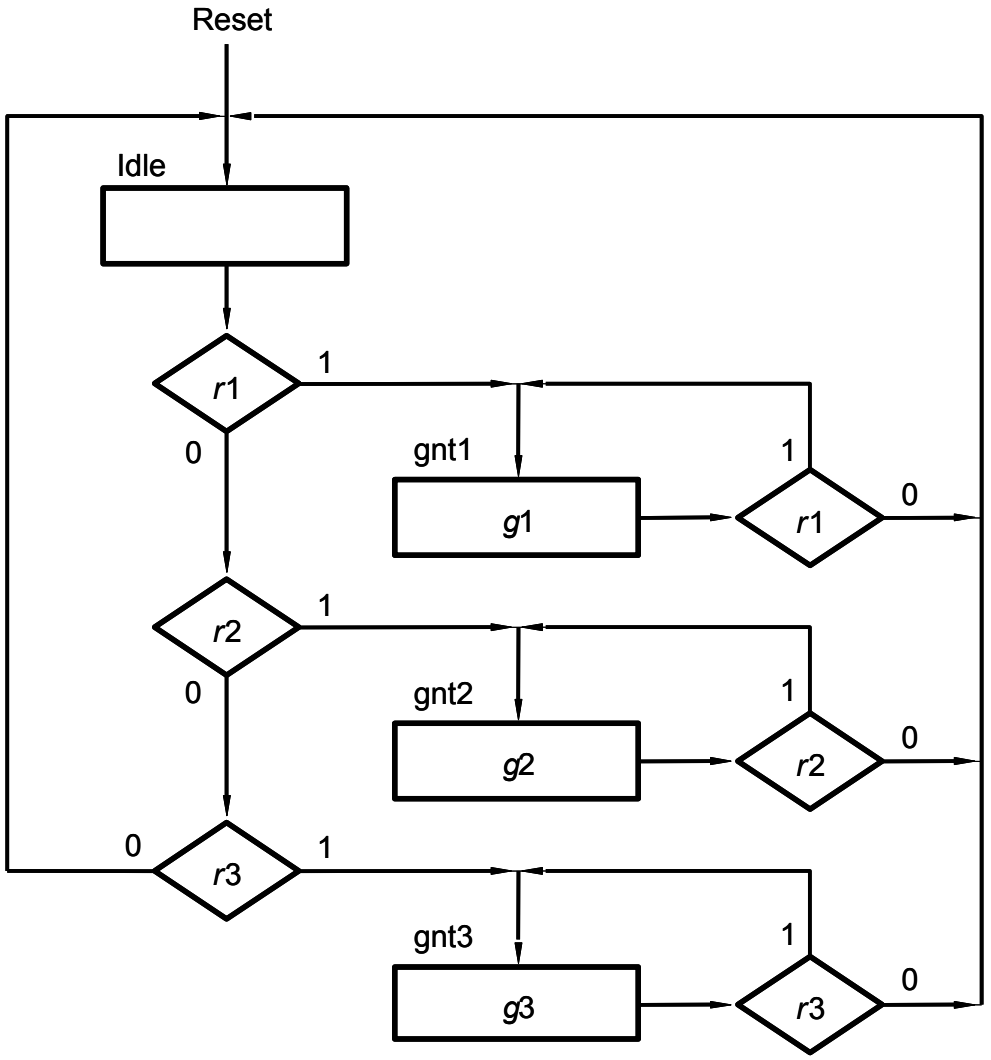
```
                                WHEN gnt3 =>
                                        IF r(3) = '1' THEN y <= gnt3 ;
                                        ELSE y <= Idle ;
                                        END IF ;
                        END CASE ;
                END IF ;
        END PROCESS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

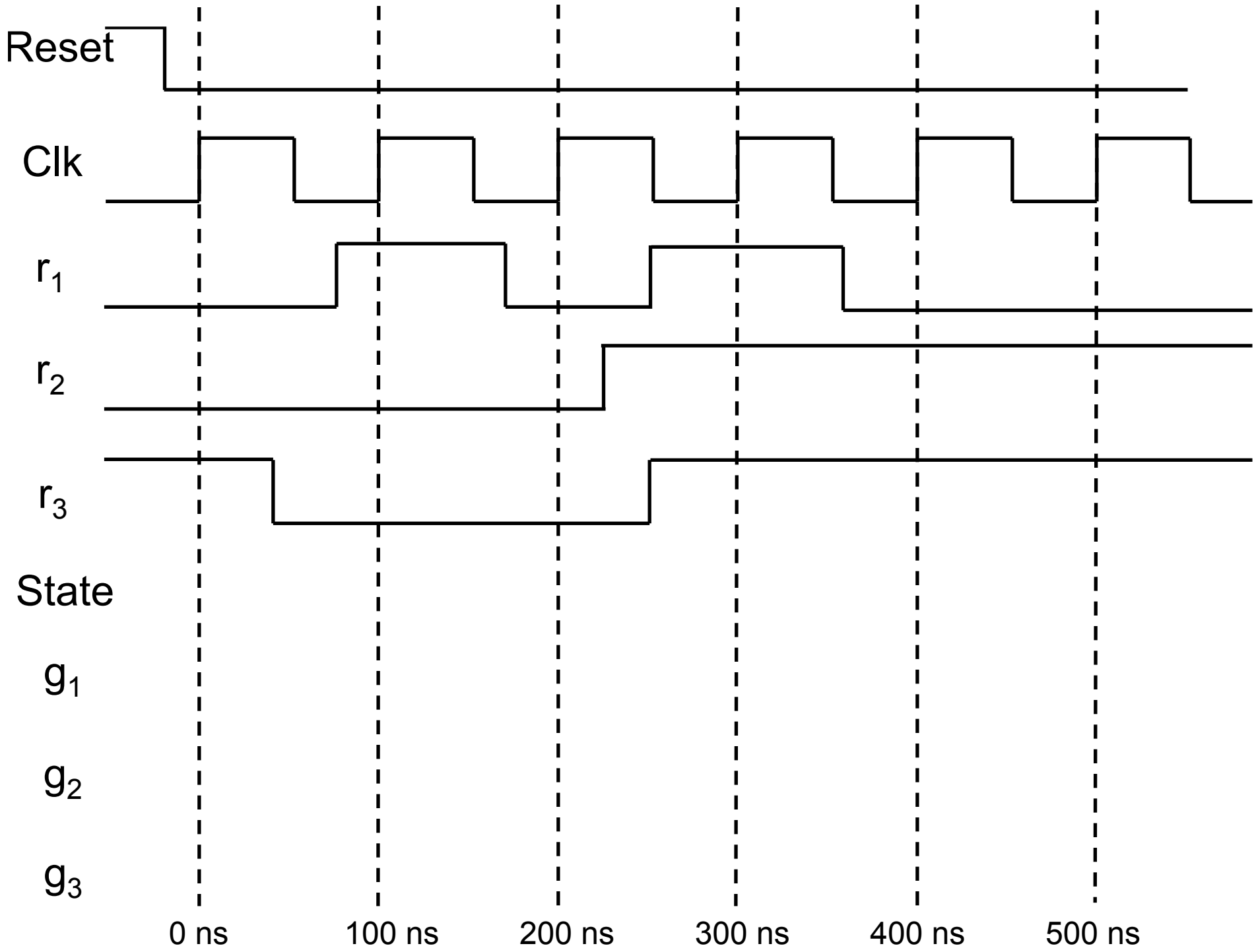
        g(1) <= '1' WHEN y = gnt1 ELSE '0' ;
        g(2) <= '1' WHEN y = gnt2 ELSE '0' ;
        g(3) <= '1' WHEN y = gnt3 ELSE '0' ;
END Behavior ;
```

Problem 2

Assuming ASM chart given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals **State, g1, g2, g3**, in the interval from 0 to 575 ns.

ASM Chart


Reset
Idle
r1
1
0
gnt1
1
g1
r1
0
r2
1
0
gnt2
1
g2
r2
0
0
r3
1
gnt3
1
g3
r3
0


Reset
Clk
r_1
r_2
r_3
State
g_1
g_2
g_3
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns

ASM Summary by Prof. Chu

- 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

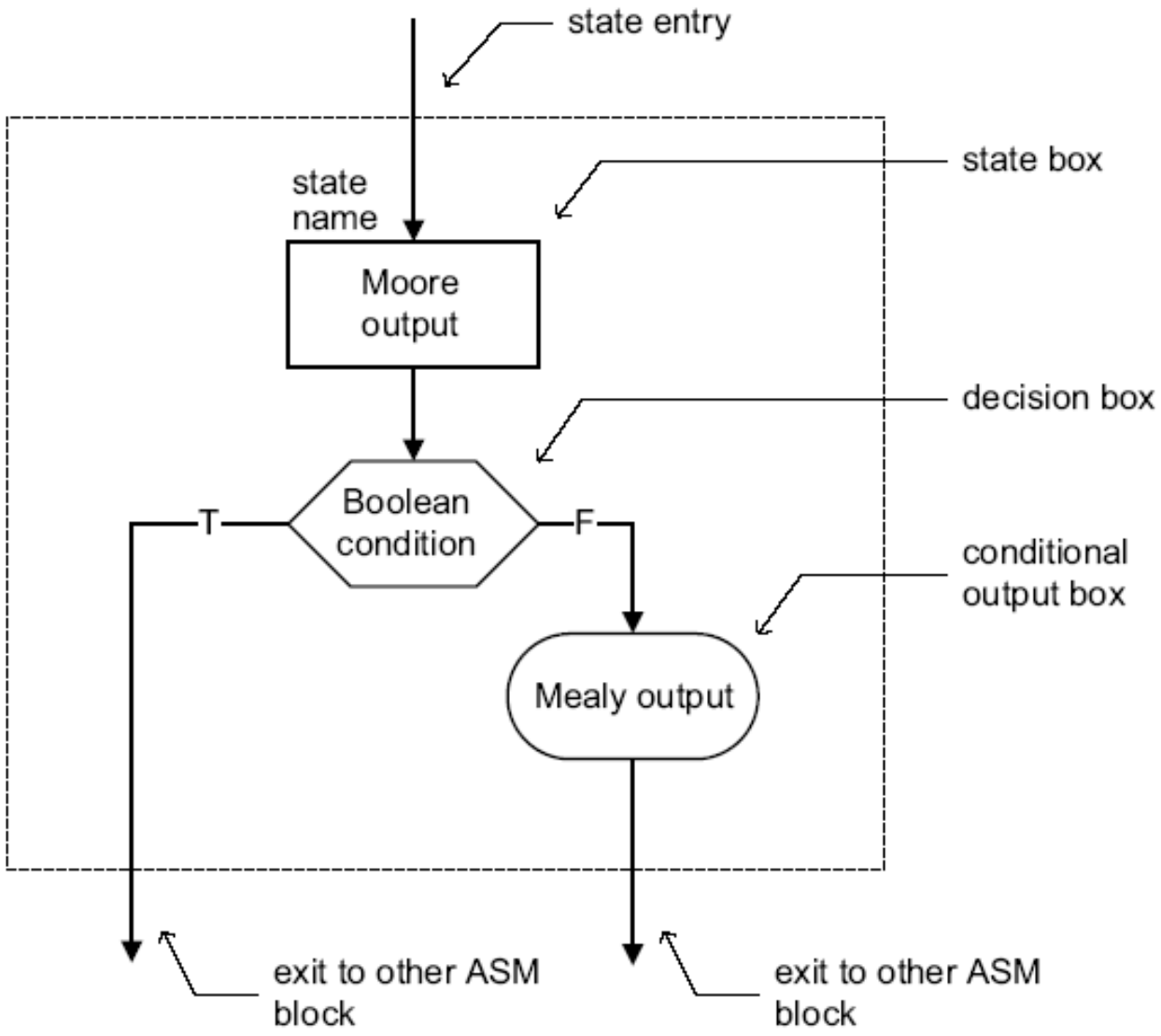


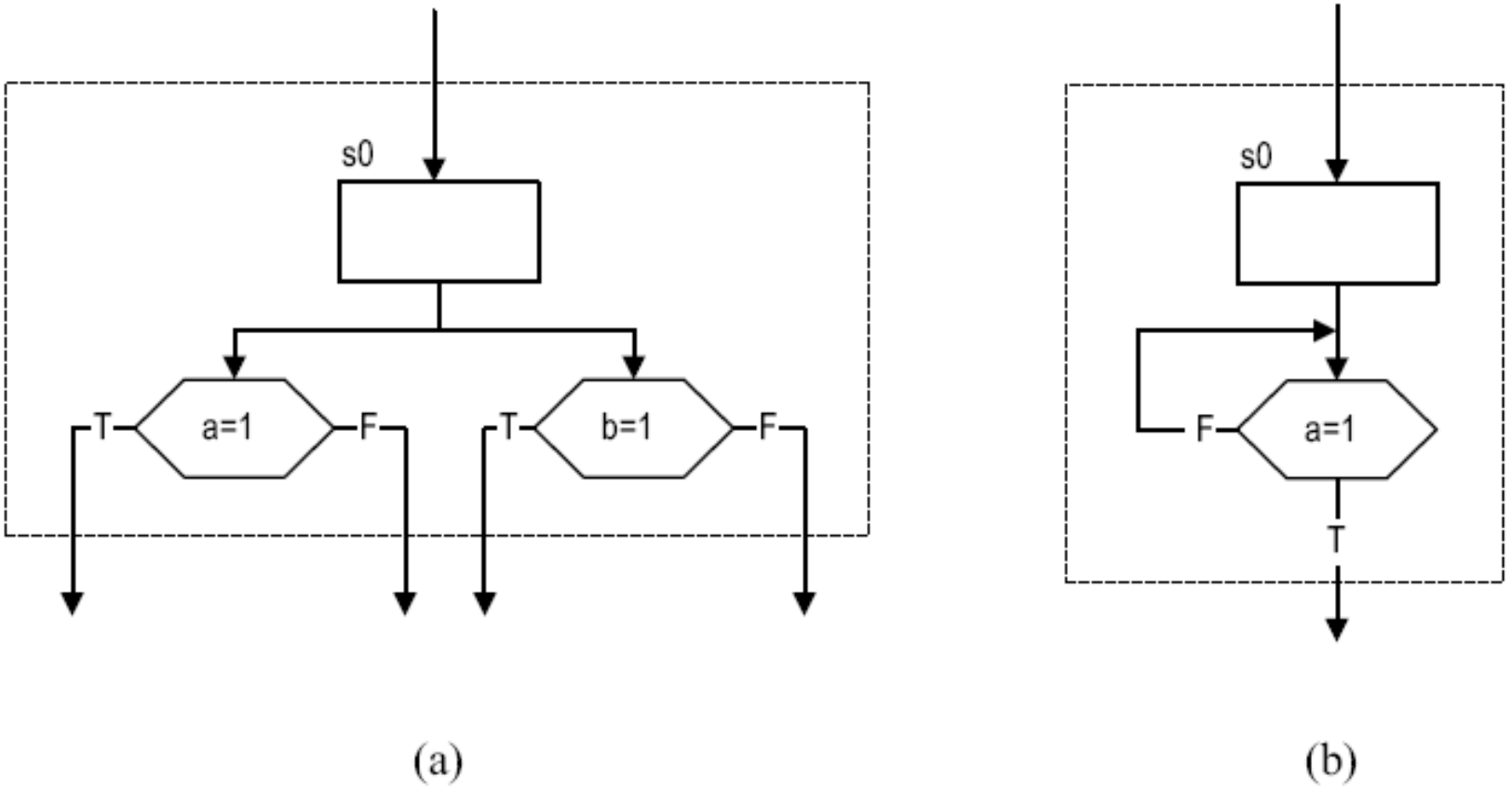
Figure 10.4 ASM block.

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

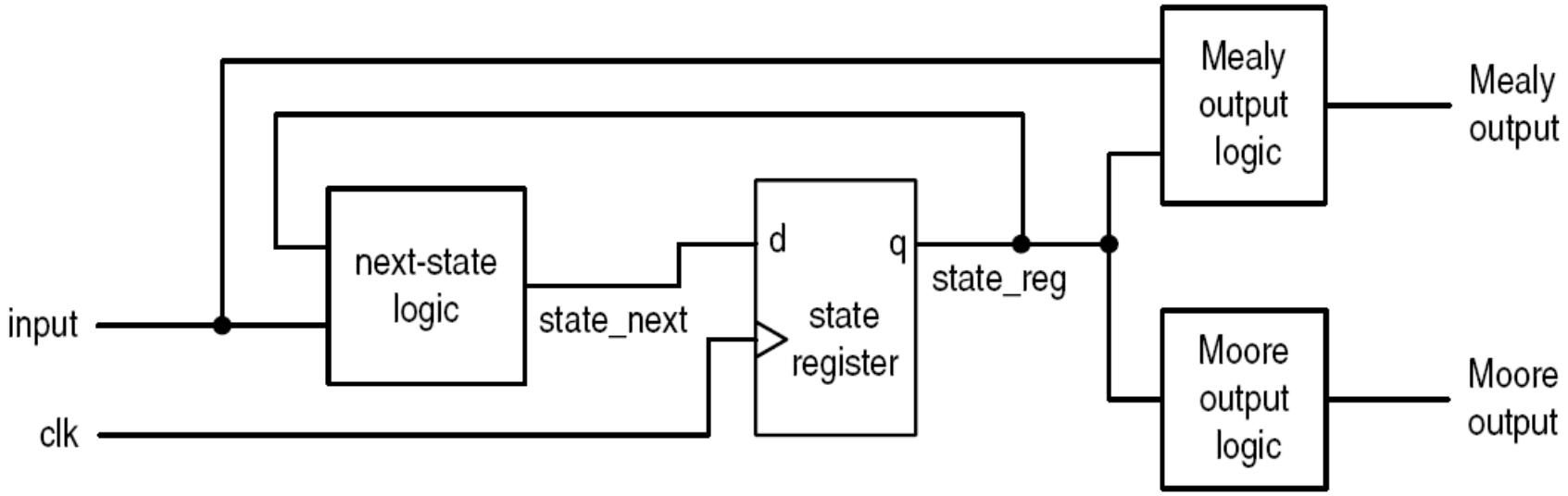
Based on RTL Hardware Design by P. Chu

Incorrect ASM Charts



Based on RTL Hardware Design by P. Chu

Generalized FSM



Based on RTL Hardware Design by P. Chu

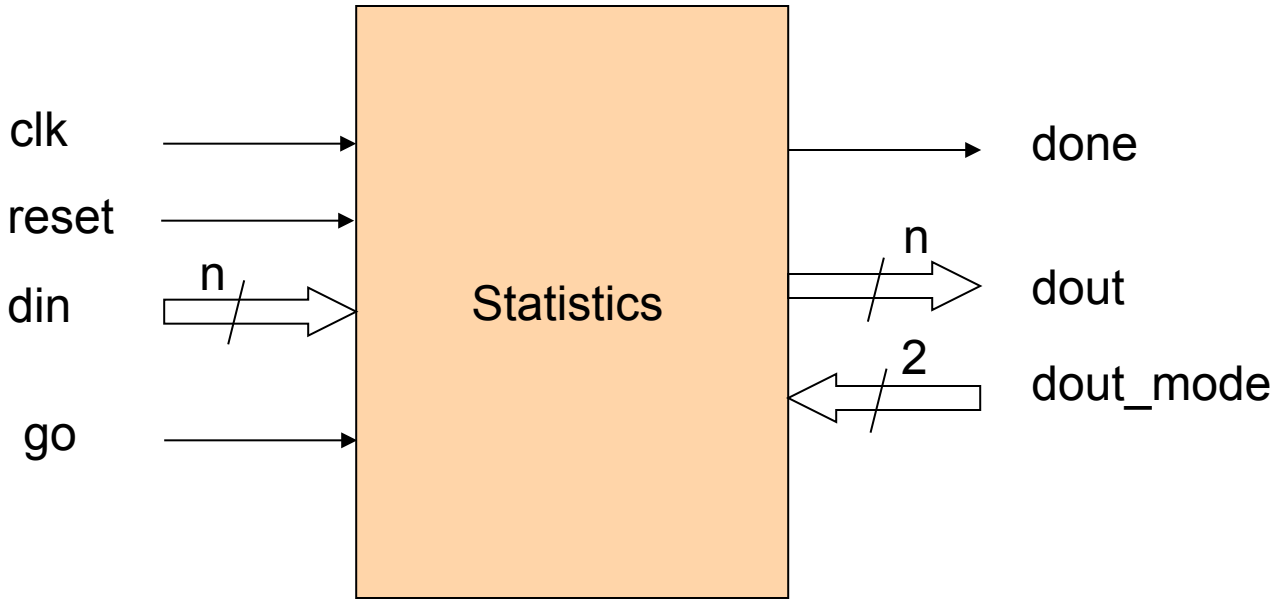


Class Exercise 1
STATISTICS

Pseudocode

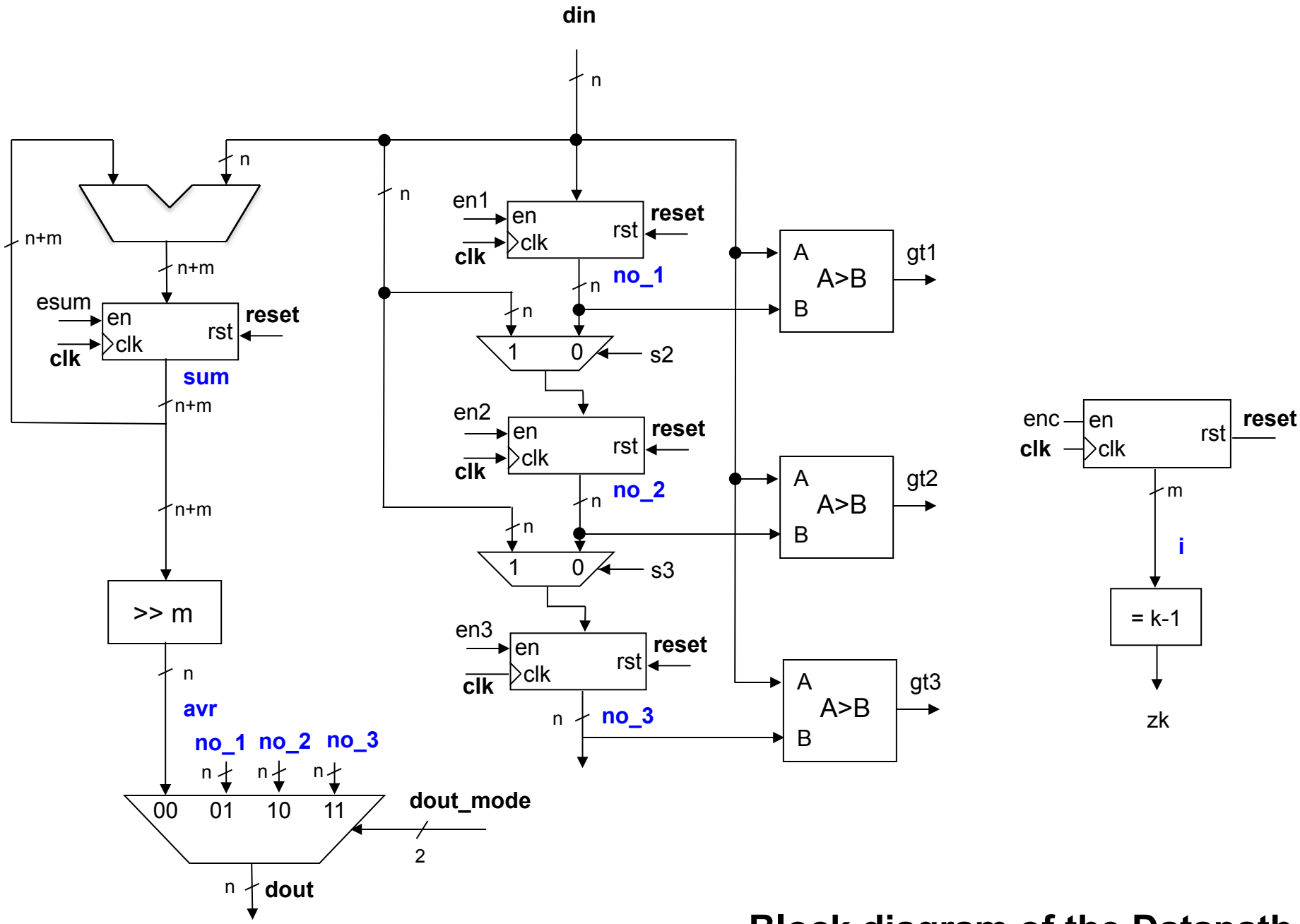
```
no_1 = no_2 = no_3 = sum = 0
for i=0 to k-1 do
    sum = sum + din
    if  din > no_1 then
            no_3 = no_2
            no_2 = no_1
            no_1 = din
    elseif (din > no_2) then
            no_3 = no_2
            no_2 = din
    elseif (din > no_3) then
            no_3 = din
    end if
end for
avr = sum / k
```

Circuit Interface



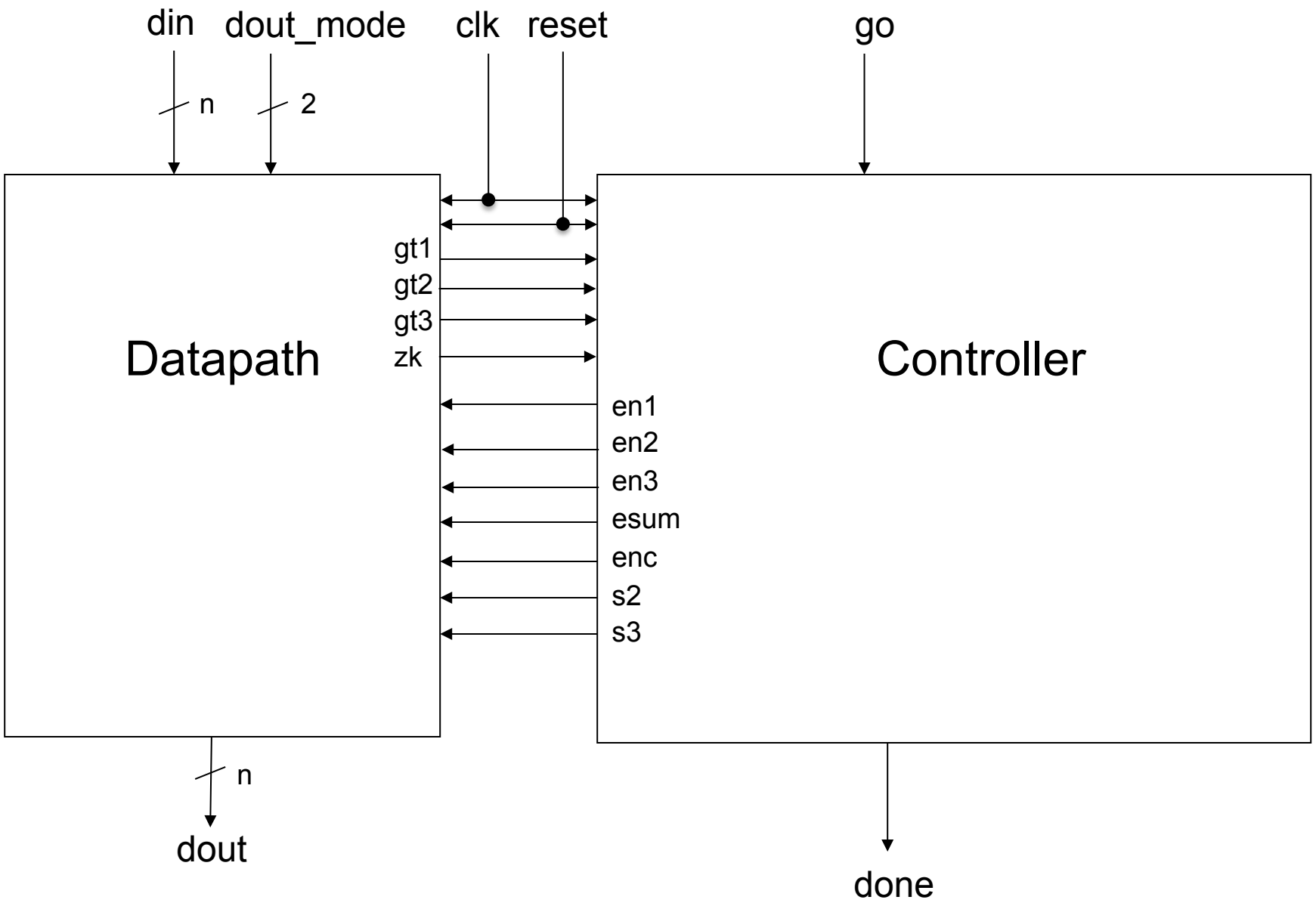
Interface Table

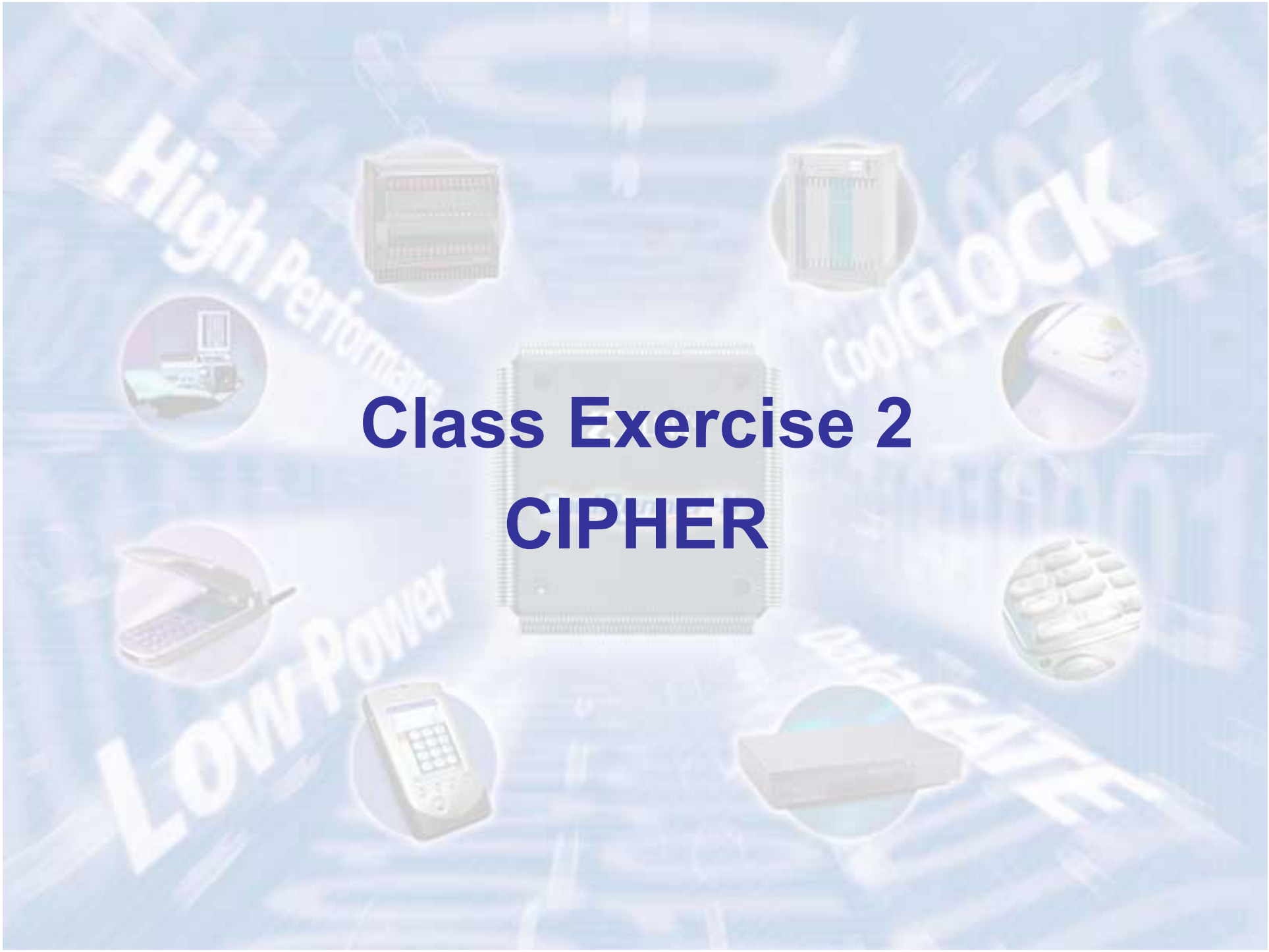
Port	Width	Meaning
clk	1	System clock.
reset	1	System reset.
din	n	Input Data.
go	1	Signal active high for k clock cycles during which Input Data is read by the circuit.
done	1	Signal set to high after the output is ready.
dout	n	Output dependent on the dout_mode input.
dout_mode	2	Control signal determining value available at the output. 00: avr, 01: no_1, 10: no_2, 11: no_3.



Block diagram of the Datapath

Interface with the division into the Datapath and the Controller





Class Exercise 2

CIPHER

Pseudocode

Split M into two halves A and B, **w** bits each

A = A + S[0]
B = B + S[1]

for j= 1 to **r** do
{
A’ = ((A⊕B) <<< B) + S[2j]
B’ = ((B⊕A’) <<< A’) + S[2j+1]
A = A’
B = B’
}

C= A || B

Notation

A, B, A', B' = w-bit variables

S[2j], S[2j+1] = a pair of round keys, each round key is a w-bit variable

$\oplus$ = an XOR of two w-bit words

+ = unsigned addition mod 2^w

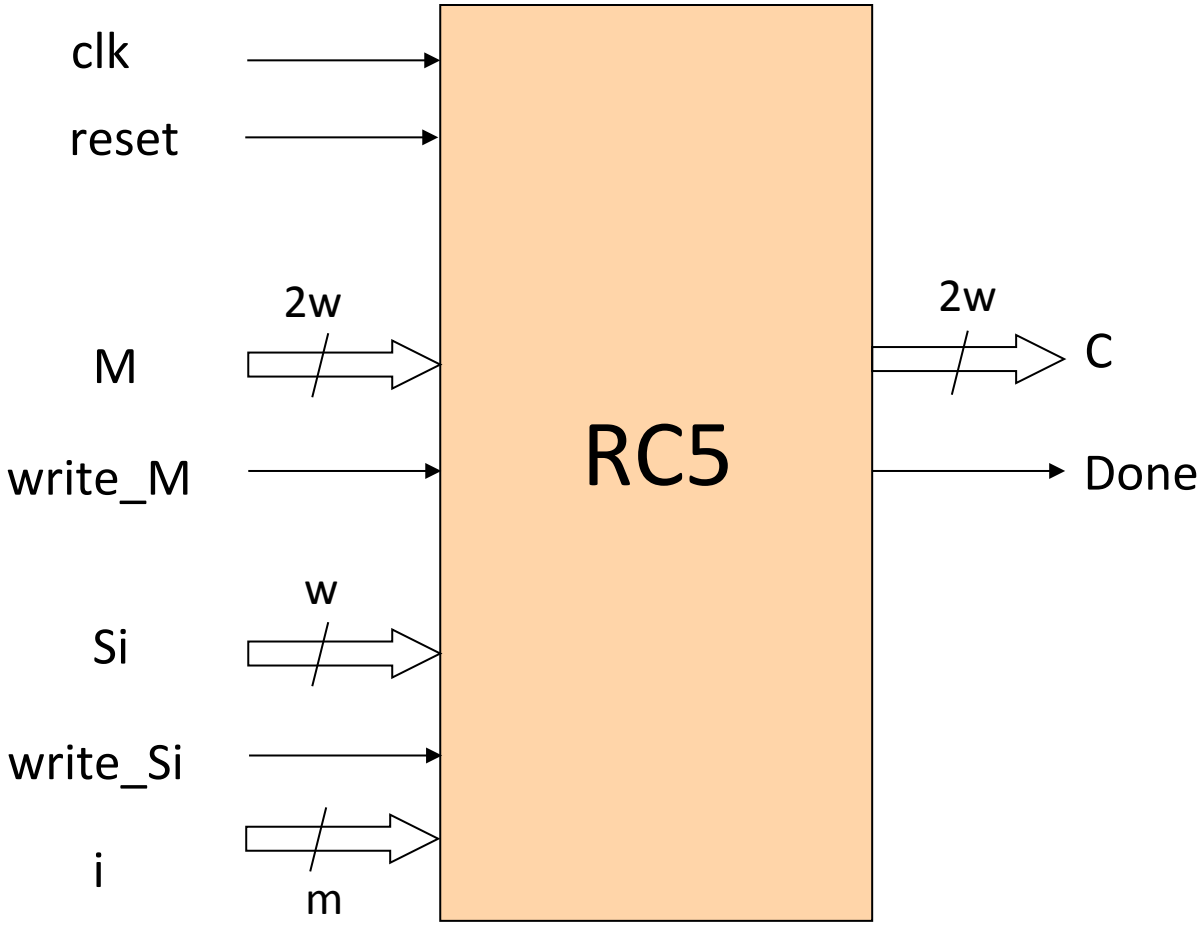
A <<< B = rotation of the variable A by a number of positions given by the current value of the variable B

A || B = concatenation of A and B

The algorithms has two parameters:

- r = number of rounds (e.g., 3)
- w = word size (always a power of 2, e.g., w = 2^4 = 16)

Circuit Interface



Circuit Interface

Port	Width	Meaning
clk	1	System clock.
reset	1	System reset – clears internal registers.
M	$2w$	Message block.
write_M	1	Synchronous write control signal for the message block M. After the block M is written to the RC5 unit, the encryption of M starts automatically.
Si	w	Round key S[i] loaded to one of the two internal memories. The first memory stores values of S[i=2j], i.e., only round keys with even indices. The second memory stores values of S[i=2j+1], i.e. only round keys with odd indices.
write_Si	1	Synchronous write control signal for the round key S[i].
i	m	Index of the round key S[i] loaded using input Si.
C	$2w$	Ciphertext block = Encrypted block M.
DONE	1	Asserted when ciphertext is ready, and available at the output.

Note:
m is a size of index i. It is a minimum integer, such that $2^m \geq 2r+2$.

Protocol (1)

An external circuit first loads all round keys
S[0], S[1], S[2], …, S[2r], [2r+1]
to the two internal memories of the RC5 unit.

The first memory stores values of S[i=2j], i.e., only round keys with even indices. The second memory stores values of S[i=2j+1], i.e. only round keys with odd indices.

Loading round keys is performed using inputs: Si, i, write_Si, clk.

Then, the external circuits, loads a message block M to the RC5 unit, using inputs: M, write_M, clk.

After the message block M is loaded to the RC5 unit, the encryption starts automatically.

Protocol (2)

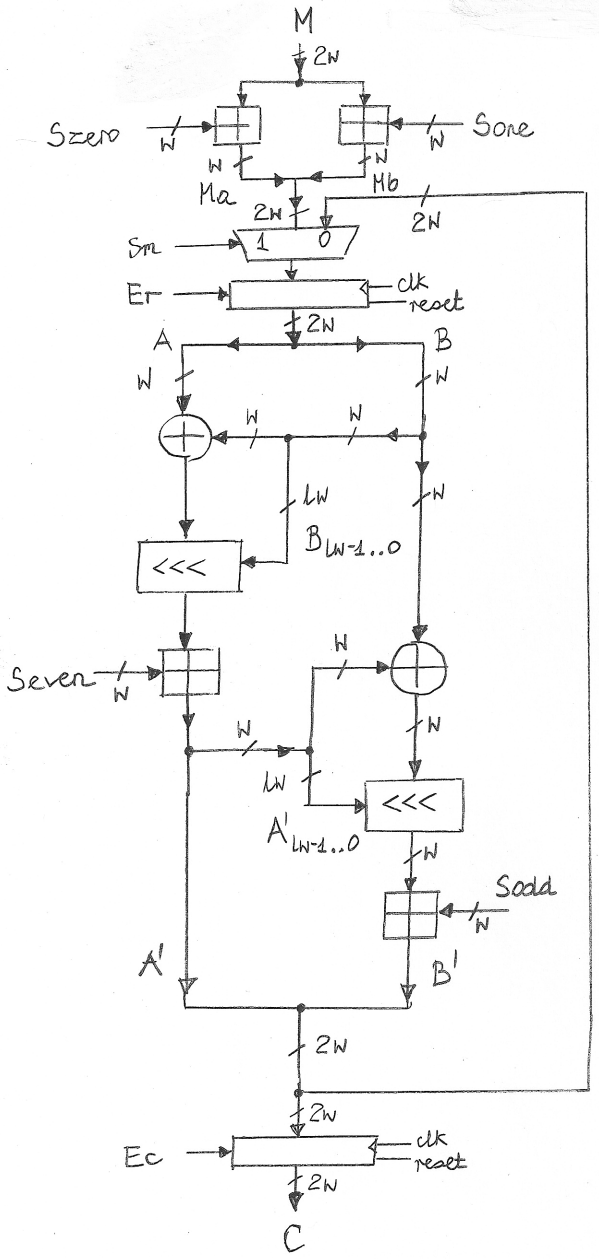
When the encryption is completed, signal Done becomes active, and the output C changes to the new value of the ciphertext.

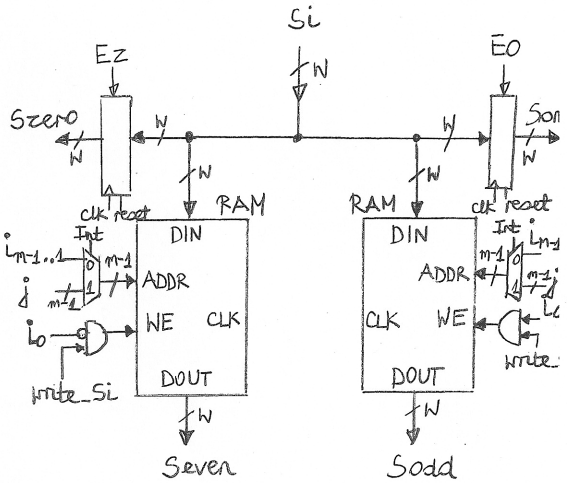
The output C keeps the last value of the ciphertext at the output, until the next encryption is completed. Before the first encryption is completed, this output should be equal to zero.

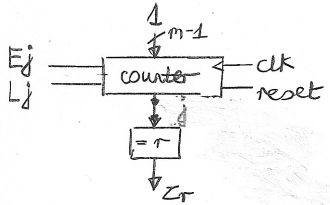
Assumptions

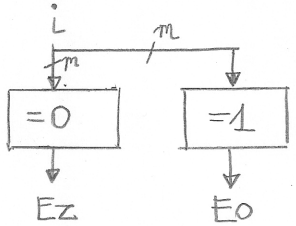
- one round of the main for loop of the pseudocode executes in one clock cycle
- you can access only one position of each internal memory of round keys per clock cycle

As a result, the entire encryption of a single message block M should last r+1 clock cycles.



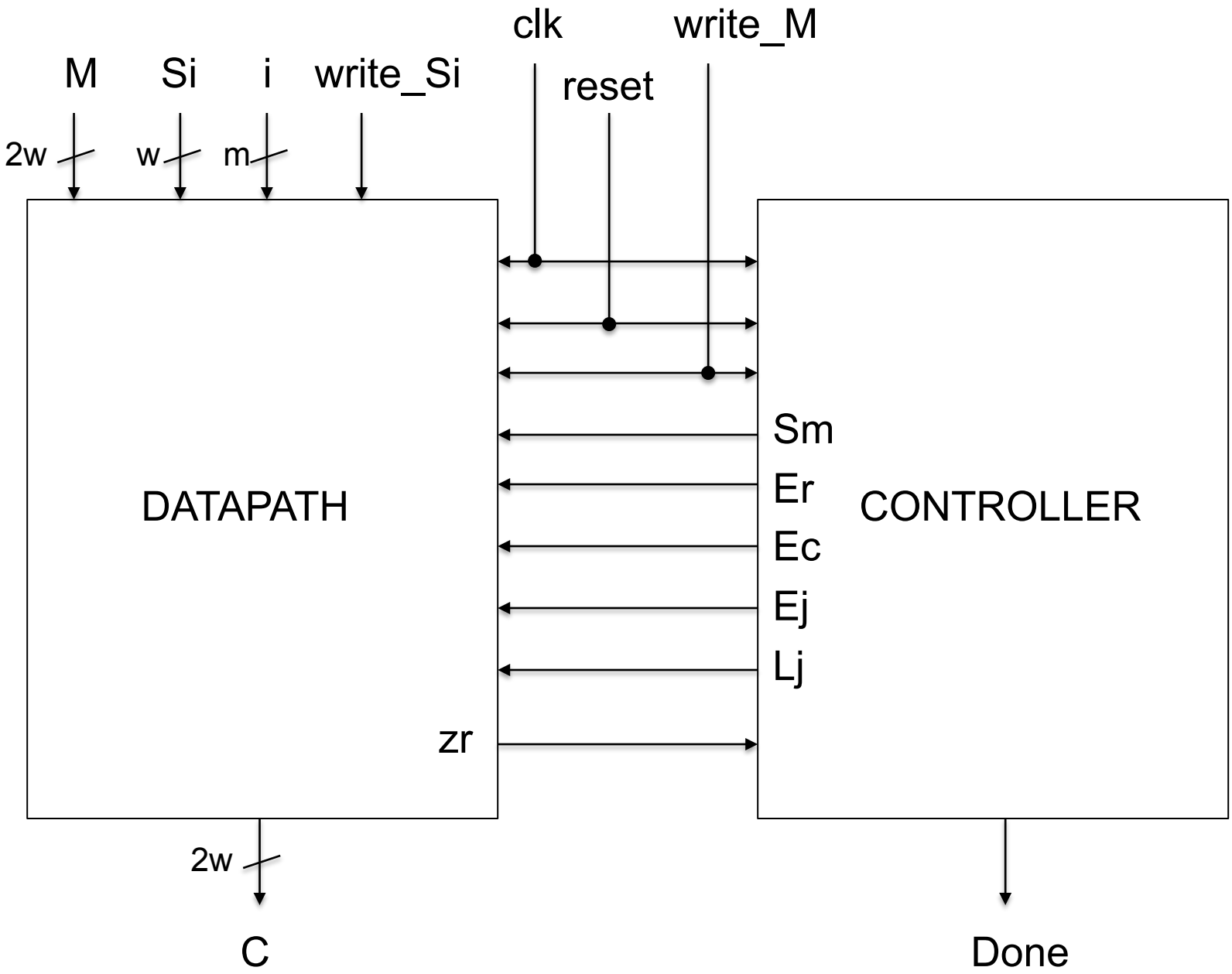


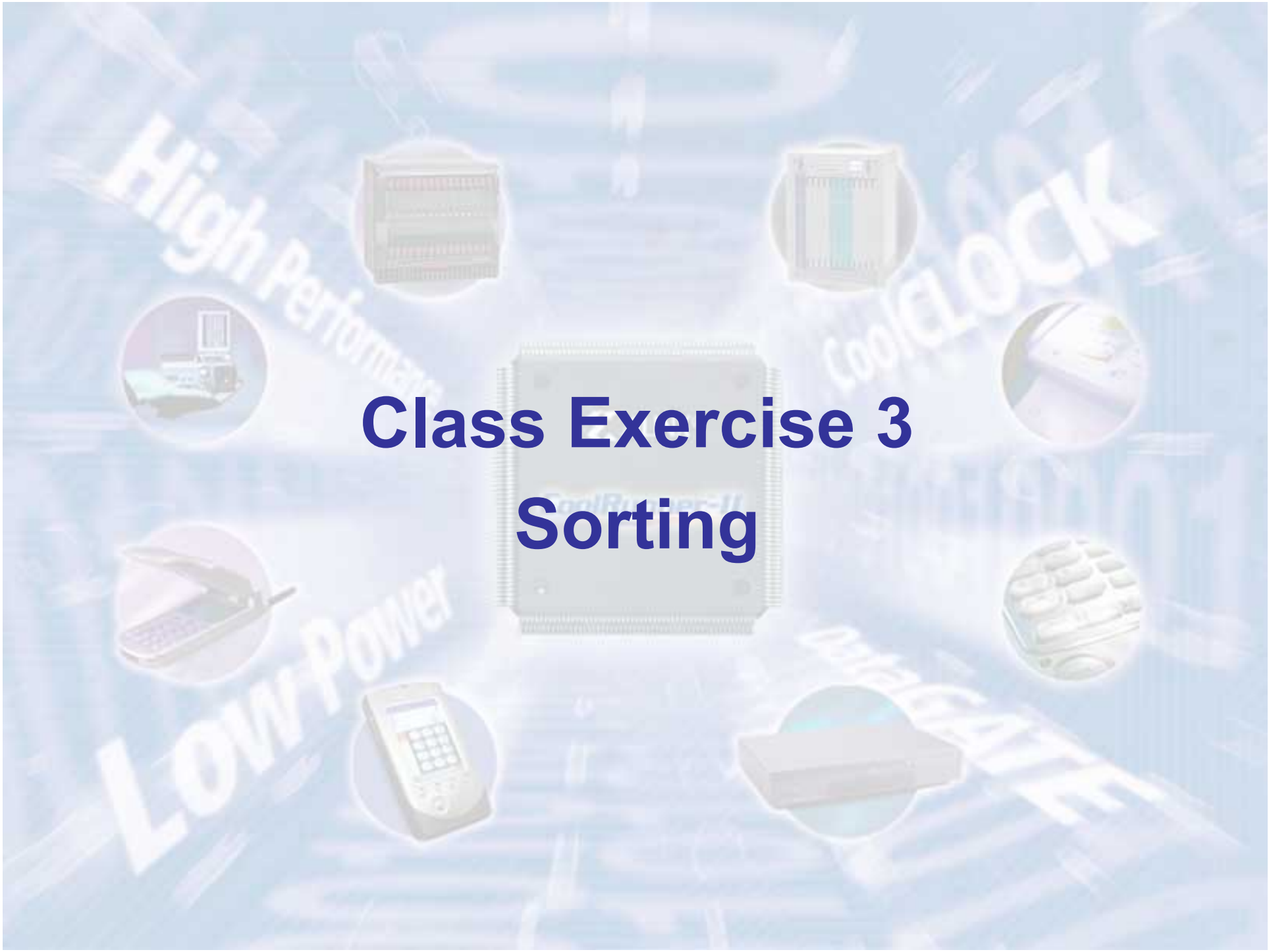




$$L_W = \lceil \log_2 W \rceil$$

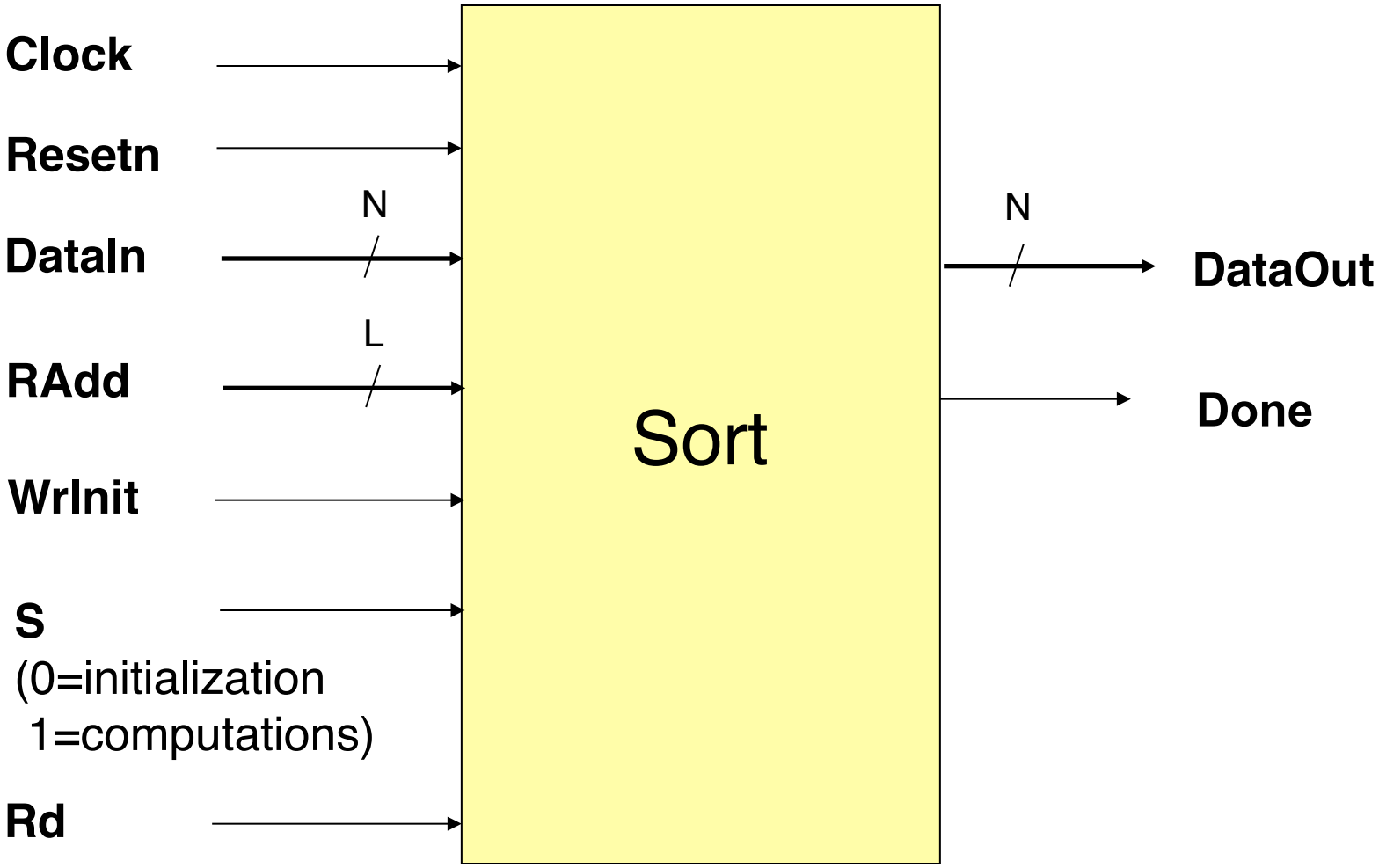
$$m = \lceil \log_2 (2r+2) \rceil$$


clk
write_M
M
Si
i
write_Si
reset
2w
w
m
Sm
Er
Ec
Ej
Lj
DATAPATH
CONTROLLER
zr
2w
C
Done



Class Exercise 3
Sorting

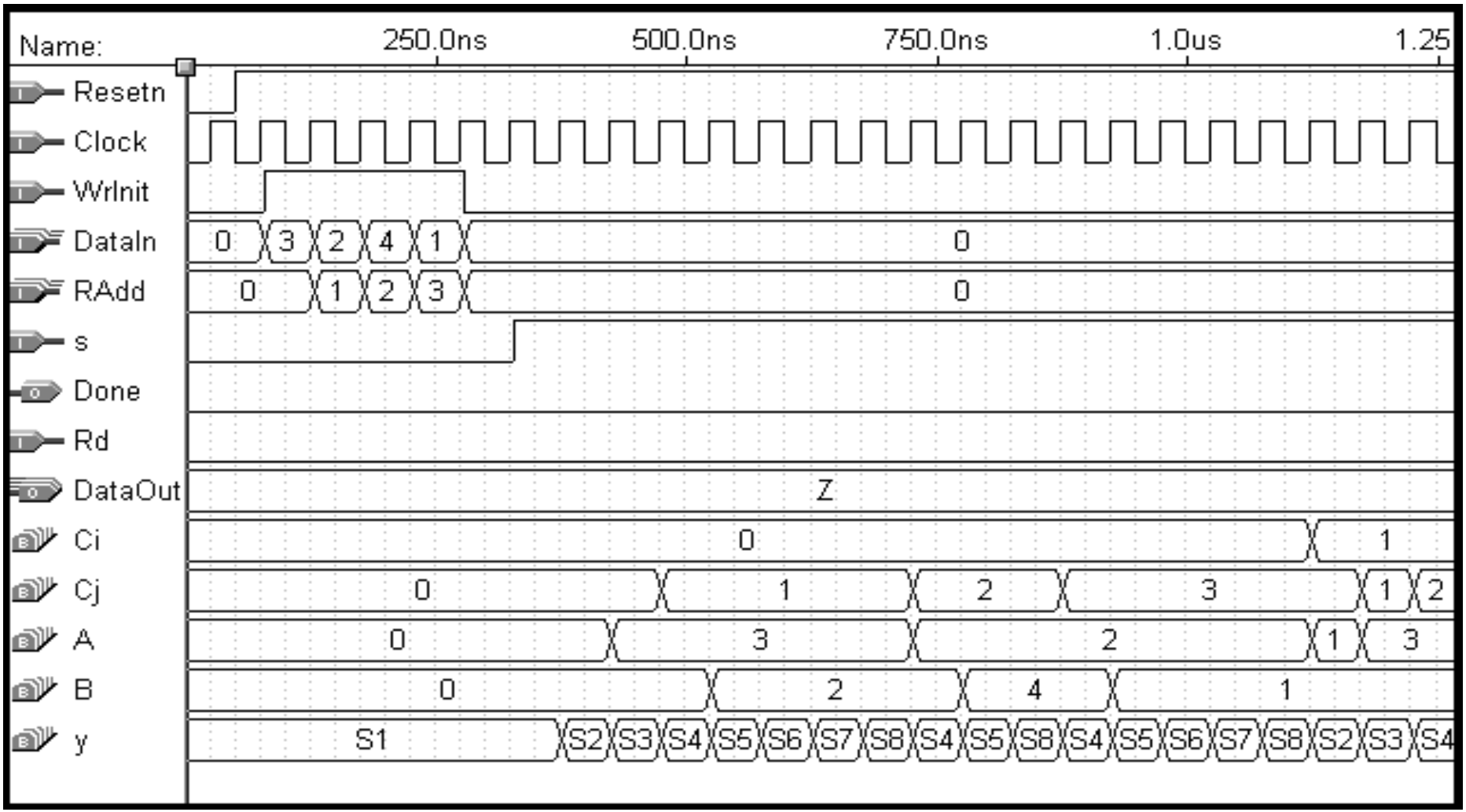
Sorting - Required Interface



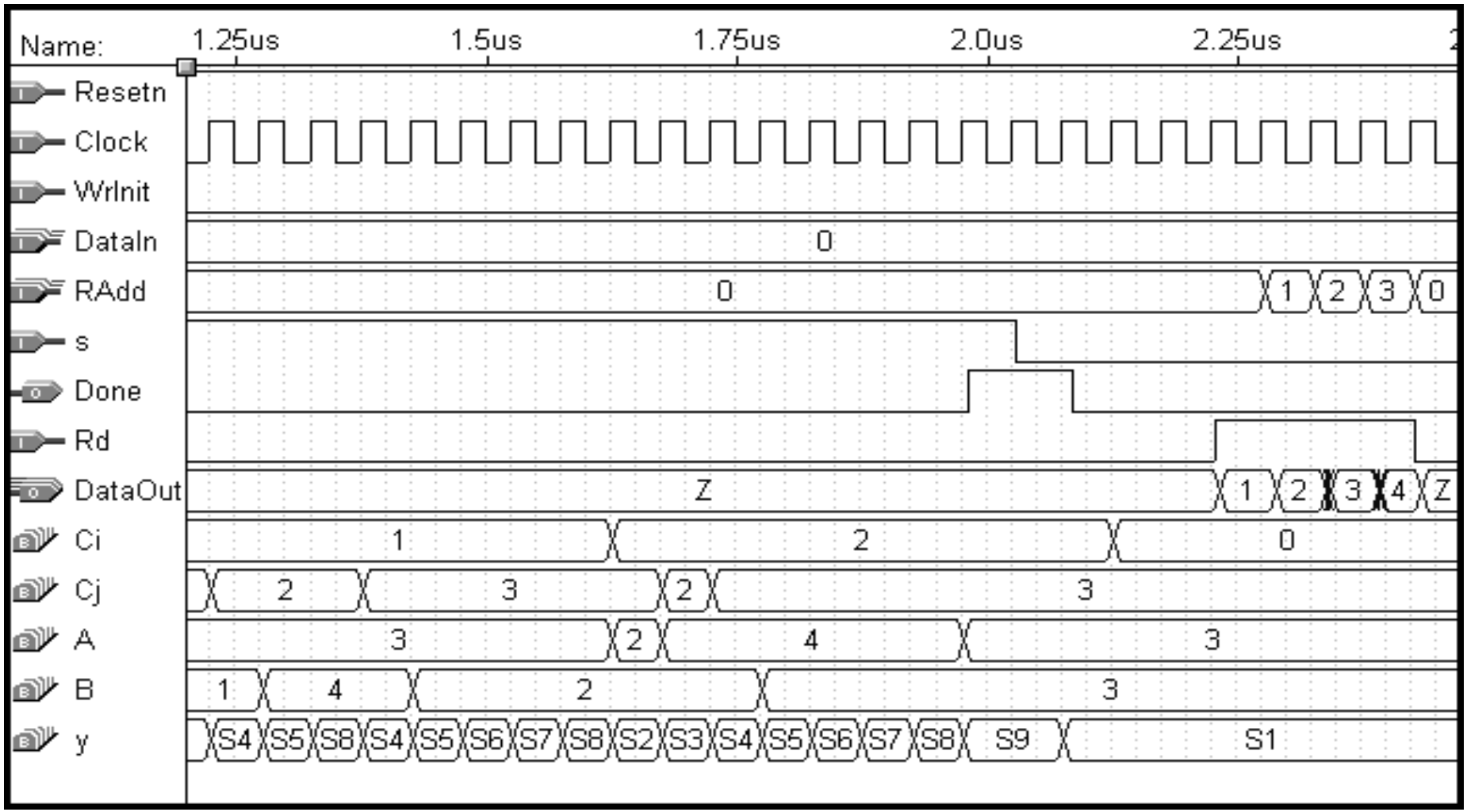
Sorting - Required Interface

Port	Width	Meaning
clk	1	System clock
Resetn	1	System reset – clears internal registers. Active low.
DataIn	N	Input data bus
RAdd	L	Address of the internal memory where input data is stored
WrInit	1	Synchronous write control signal
s	1	Operating mode: 0 = initialization, 1 = computations.
Rd	1	Read enable. 0 = high impedance on the output bus, 1 = valid output on the output data bus.
DataOut	N	Output data bus used to read results
Done	1	Asserted when all results are ready

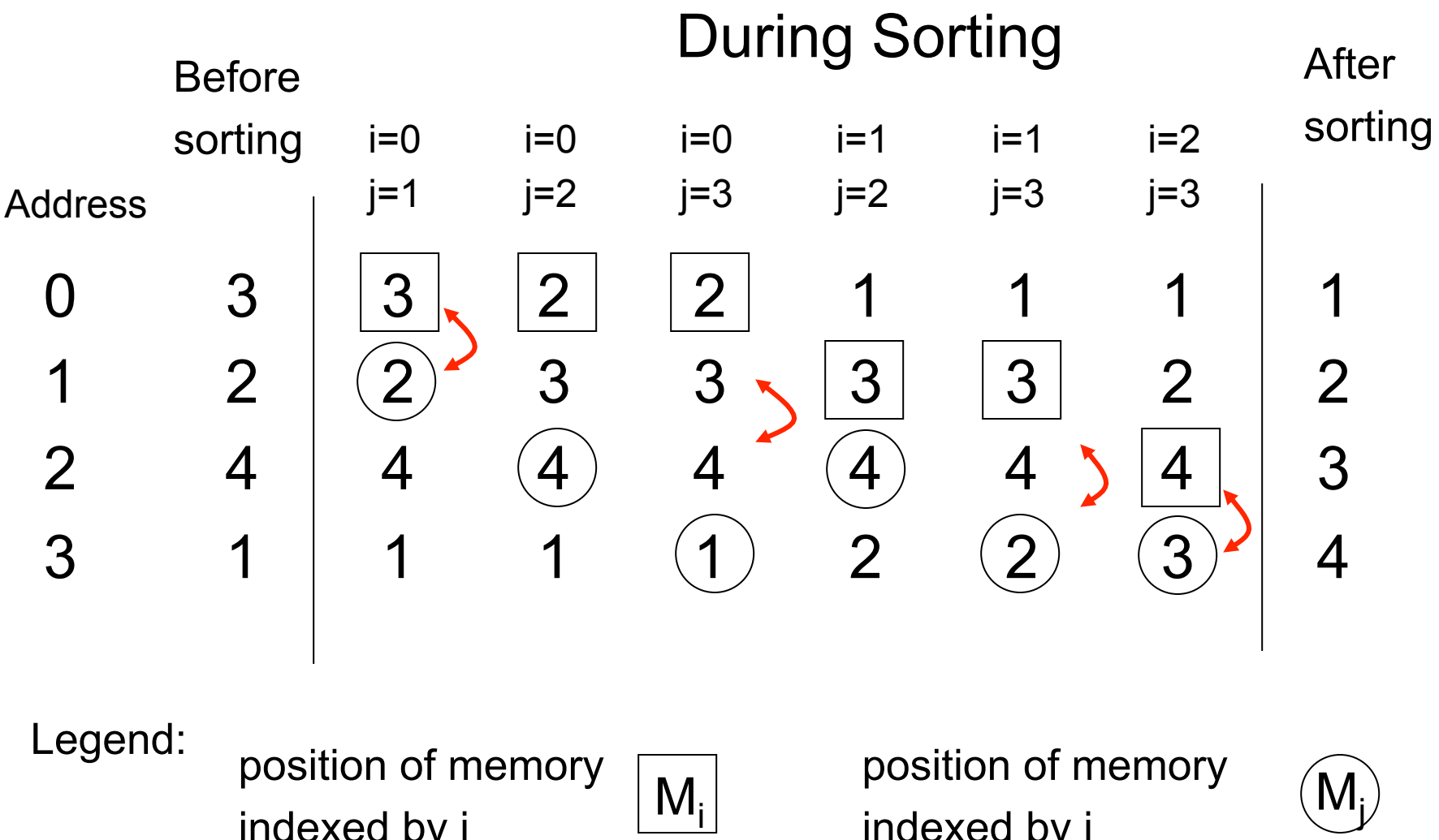
Simulation results for the sort operation (1)
Loading memory and starting sorting



Simulation results for the sort operation (2)
Completing sorting and reading out memory



Sorting - Example



Legend:

position of memory indexed by i M_i

position of memory indexed by j M_j

Pseudocode

FOR k = 4

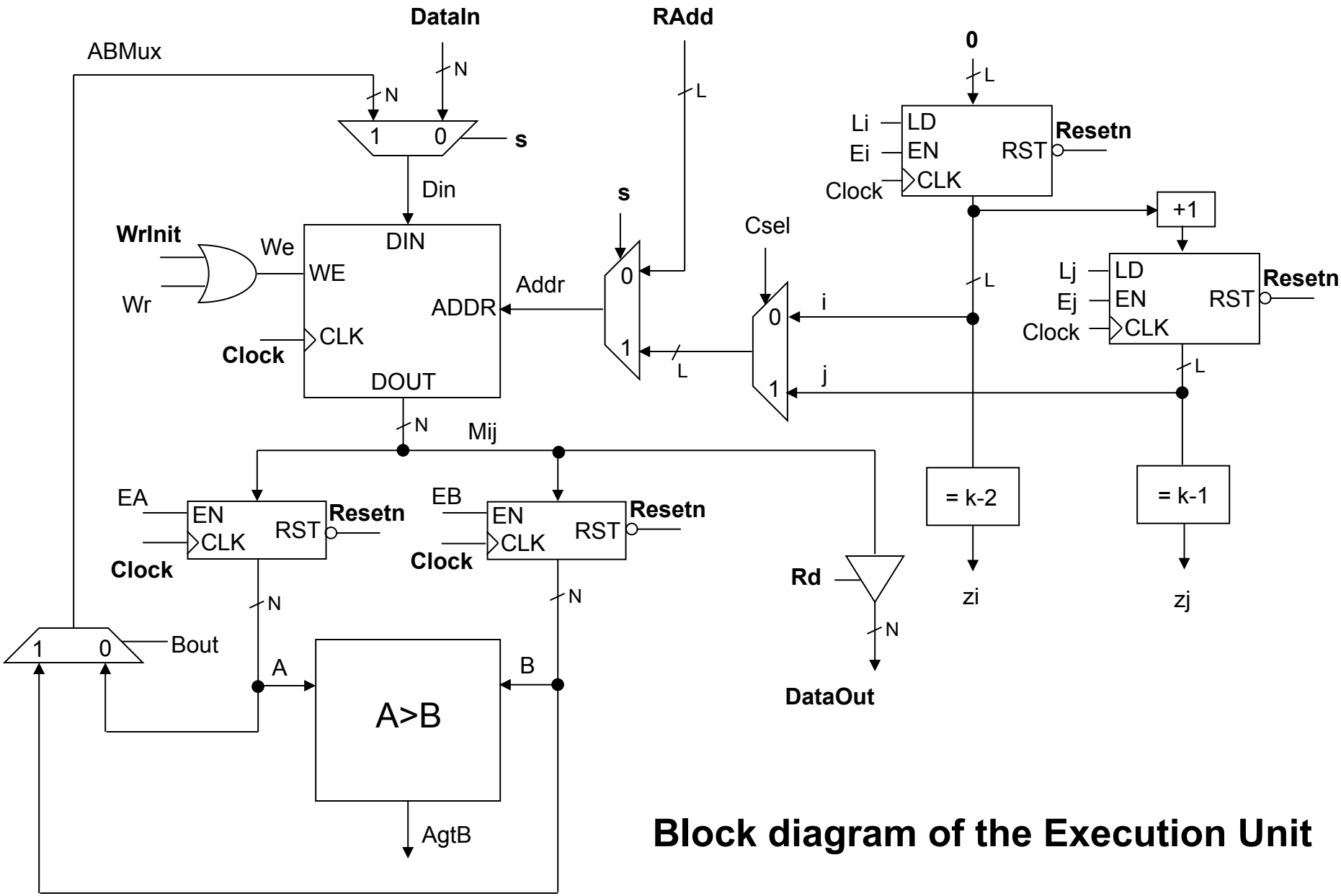
```
[load input data]
for i = 0 to 2 do
      A = Mi ;
      for j = i + 1 to 3 do
            B = Mj ;
            if B < A then
                  Mi = B ;
                  Mj = A ;
                  A = Mi ;
            endif ;
      endfor;
endfor;
[read output data]
```

FOR any k ≥ 2

```
[load input data]
for i = 0 to k-2 do
      A = Mi ;
      for j = i + 1 to k – 1 do
            B = Mj ;
            if B < A then
                  Mi = B ;
                  Mj = A ;
                  A = Mi ;
            endif ;
      endfor;
endfor;
[read output data]
```

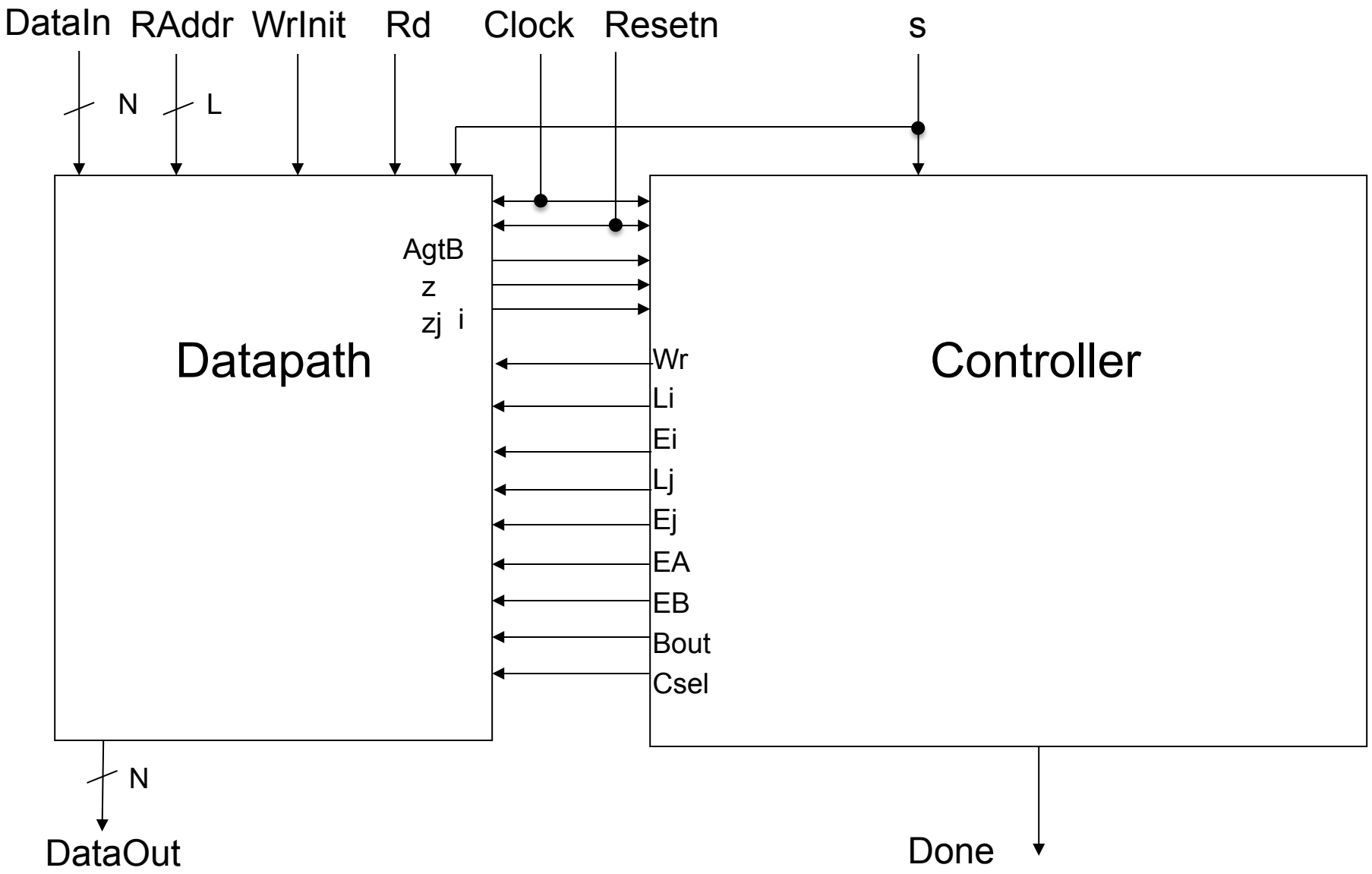
Pseudocode

```
wait for s=1
for i=0 to k-2 do
    A = M_i
    for j=i+1 to k-1 do
        B = M_j
        if A > B then
            M_i = B
            M_j = A
            A = M_i
        end if
    end for
end for
Done
wait for s=0
go to the beginning
```



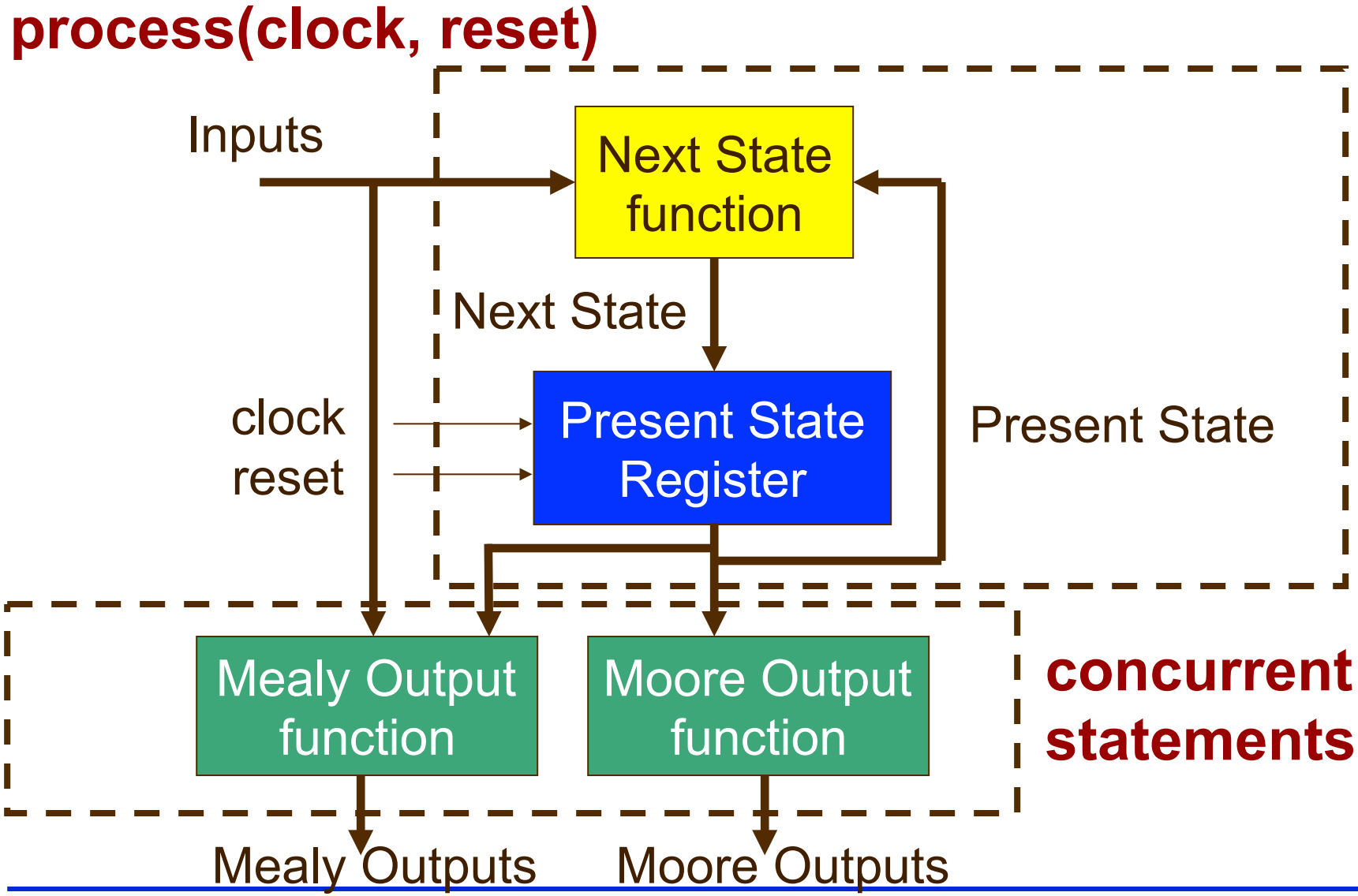
Block diagram of the Execution Unit

Interface with the division into the Datapath and the Controller

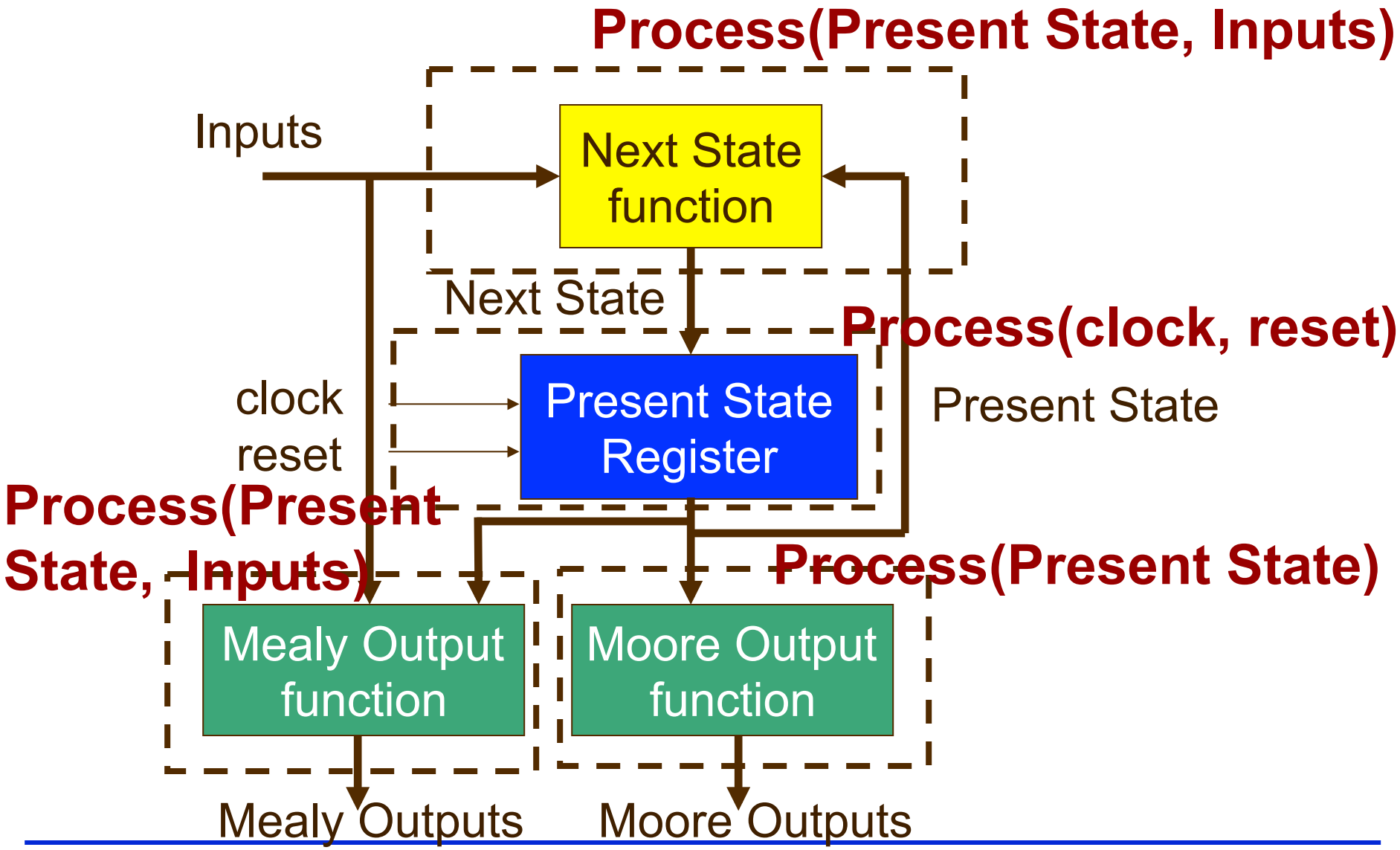


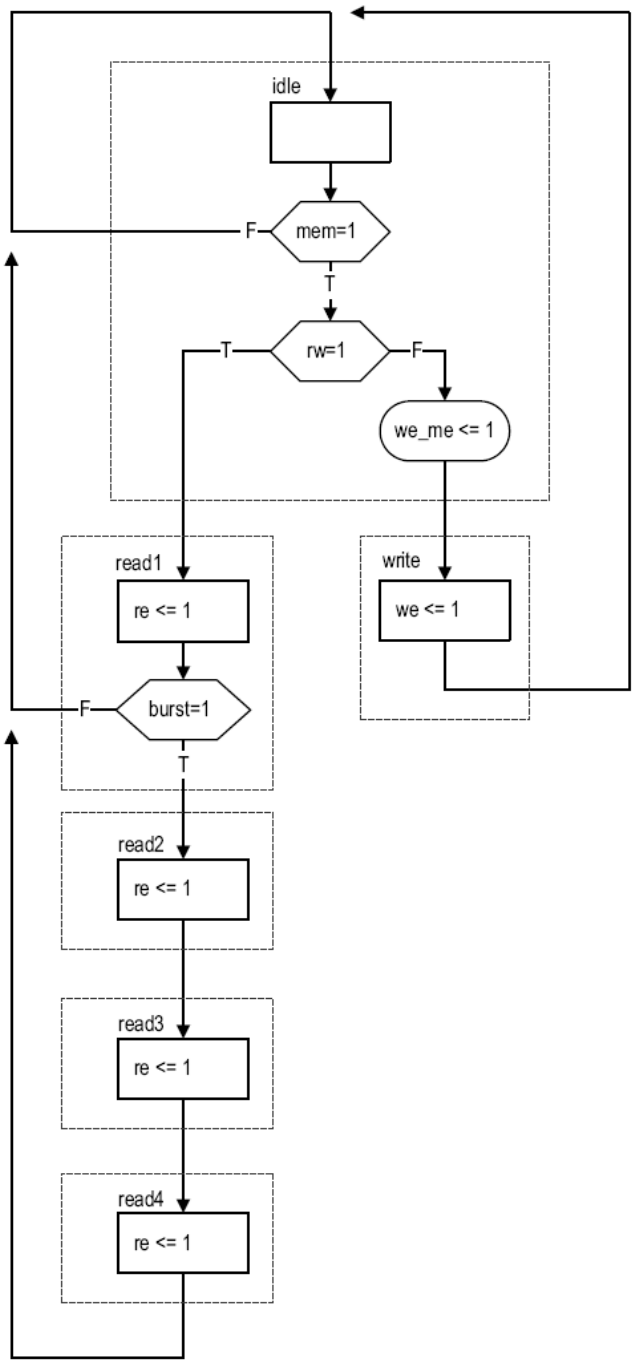

Alternative Coding Styles
by Dr. Chu
(to be used with caution)

Traditional Coding Style



Alternative Coding Style 1




idle
F
mem=1
T
T
rw=1
F
we_me <= 1
read1
re <= 1
write
we <= 1
F
burst=1
T
read2
re <= 1
read3
re <= 1
read4
re <= 1

```
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;
```

```
-- 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 write =>
          state_next <= idle;
```

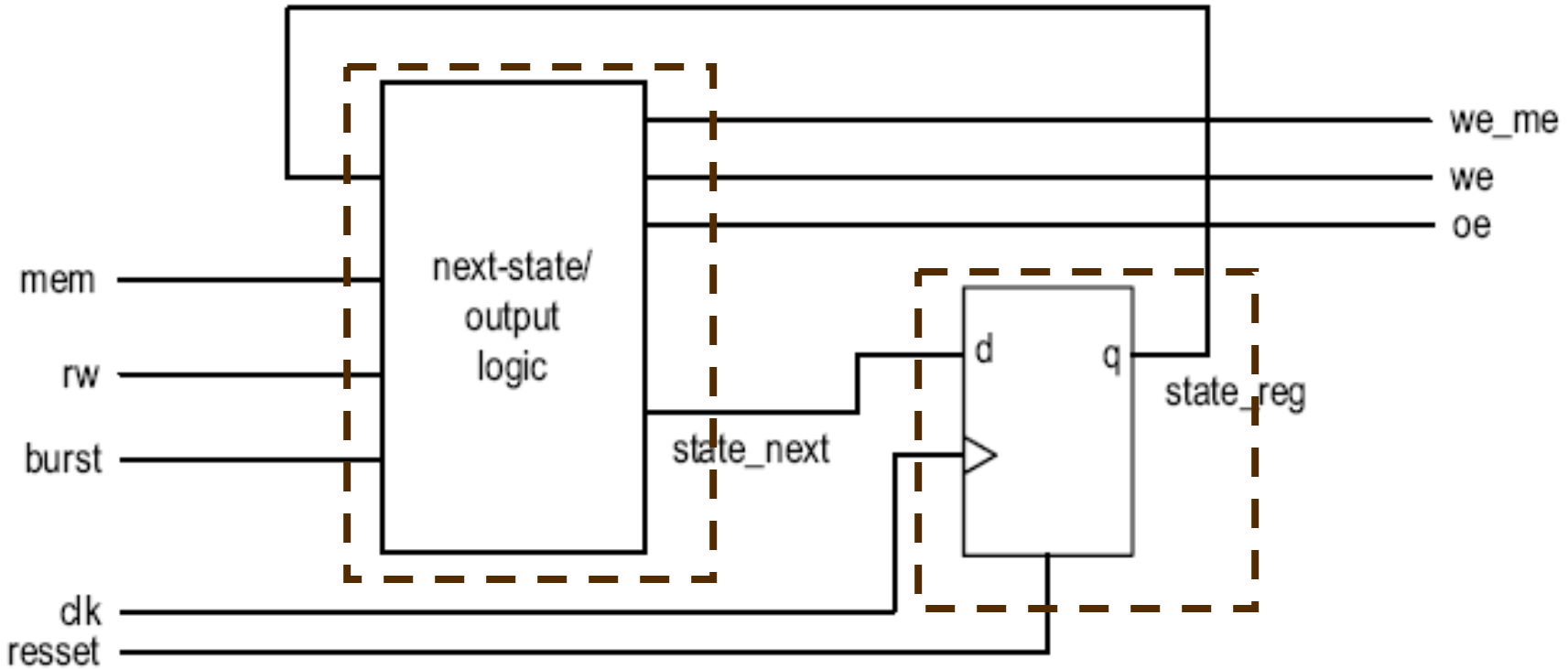
```
        when read1 =>
            if (burst='1') then
                state_next <= read2;
            else
                state_next <= idle;
            end if;
        when read2 =>
            state_next <= read3;
        when read3 =>
            state_next <= read4;
        when read4 =>
            state_next <= idle;
    end case;
end process;
```

```
-- moore output 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 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)

```
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';
```

```
        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;
```