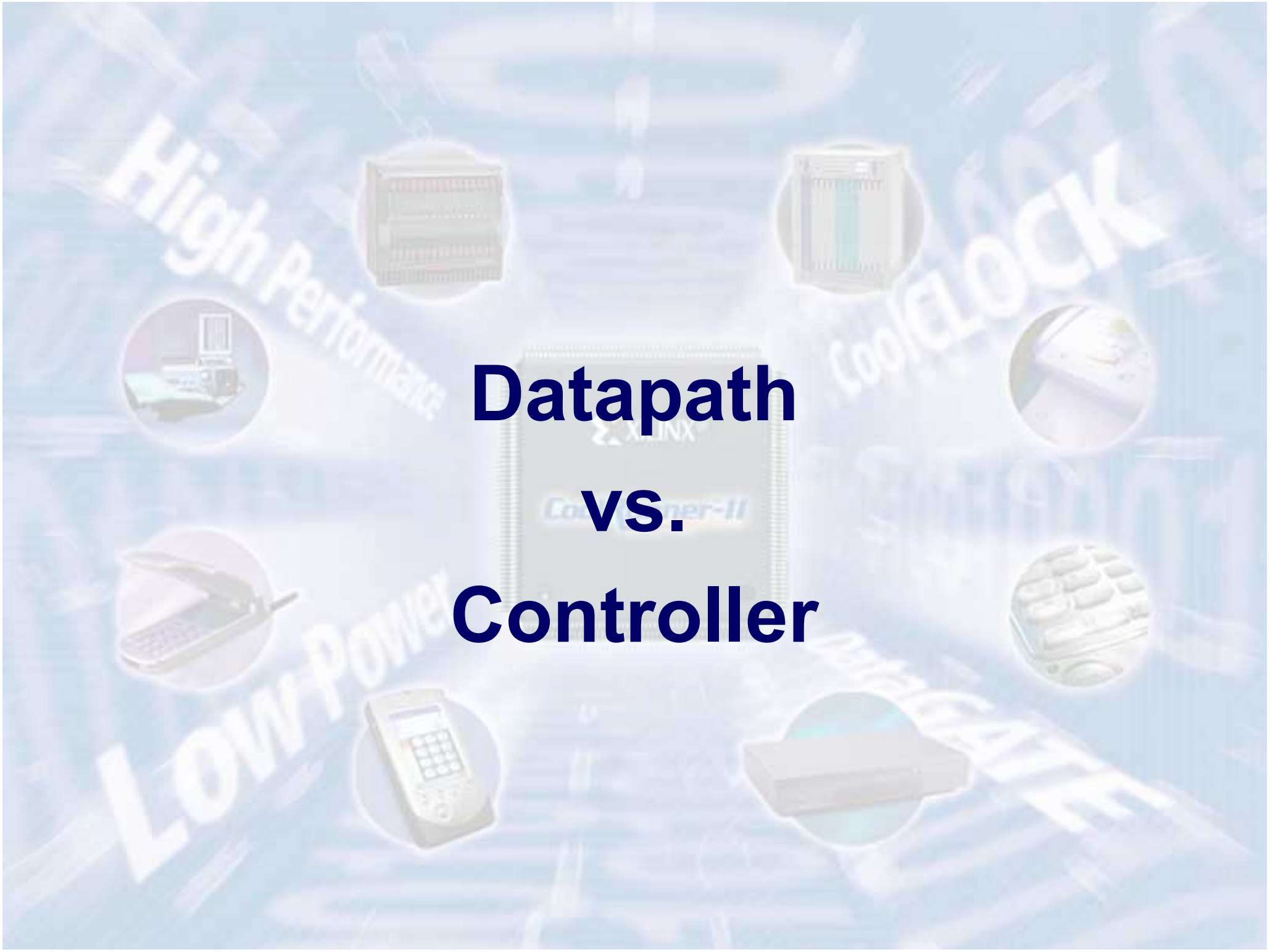
ECE 545
Lecture 9

Finite State Machines.
Design of Controllers
using
Algorithmic State Machine (ASM) Charts.

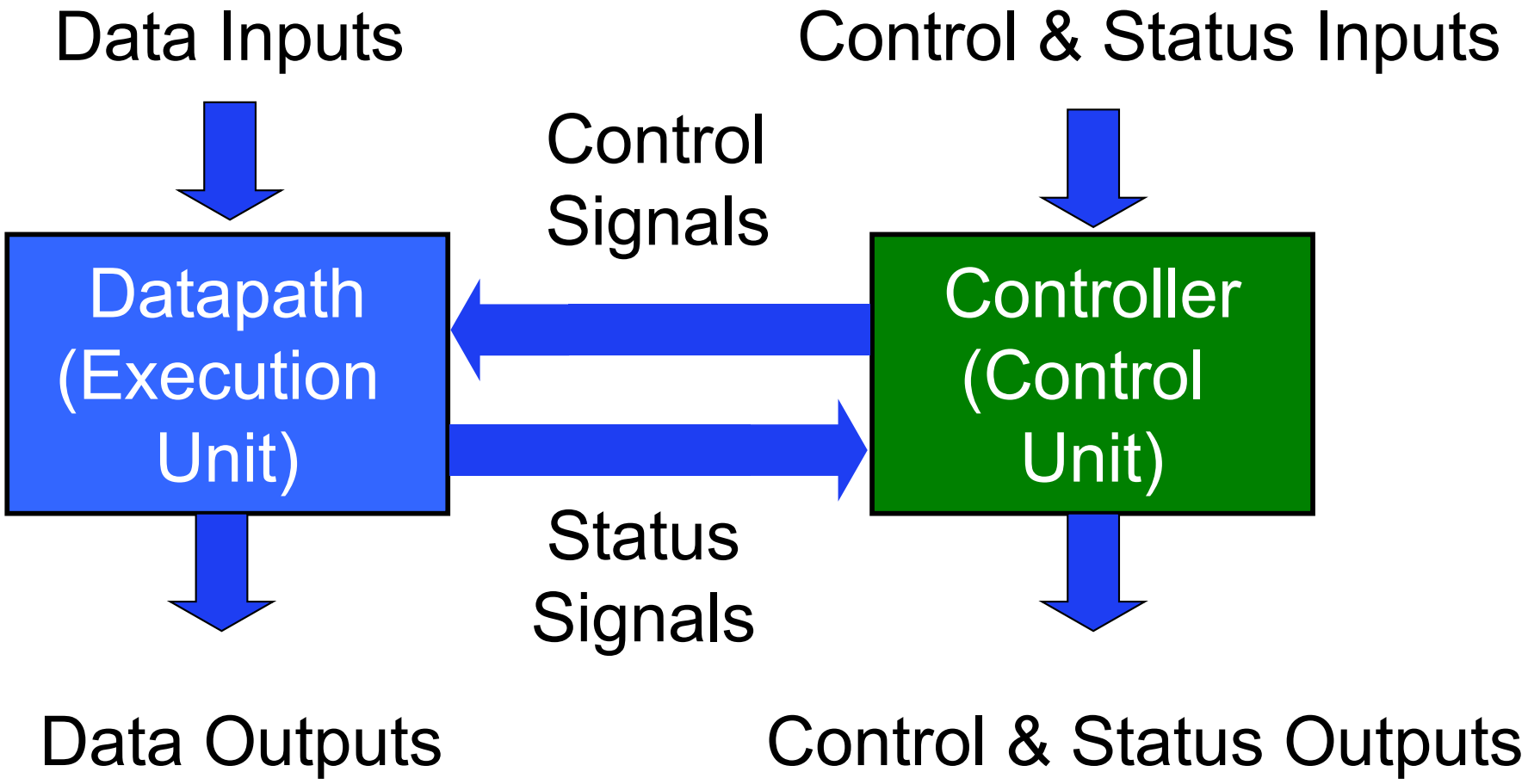
Required reading

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

Chapter 10, Finite State Machine: Principle & Practice


Datapath
vs.
Controller

Structure of a Typical Digital System



Controller (Control Unit)

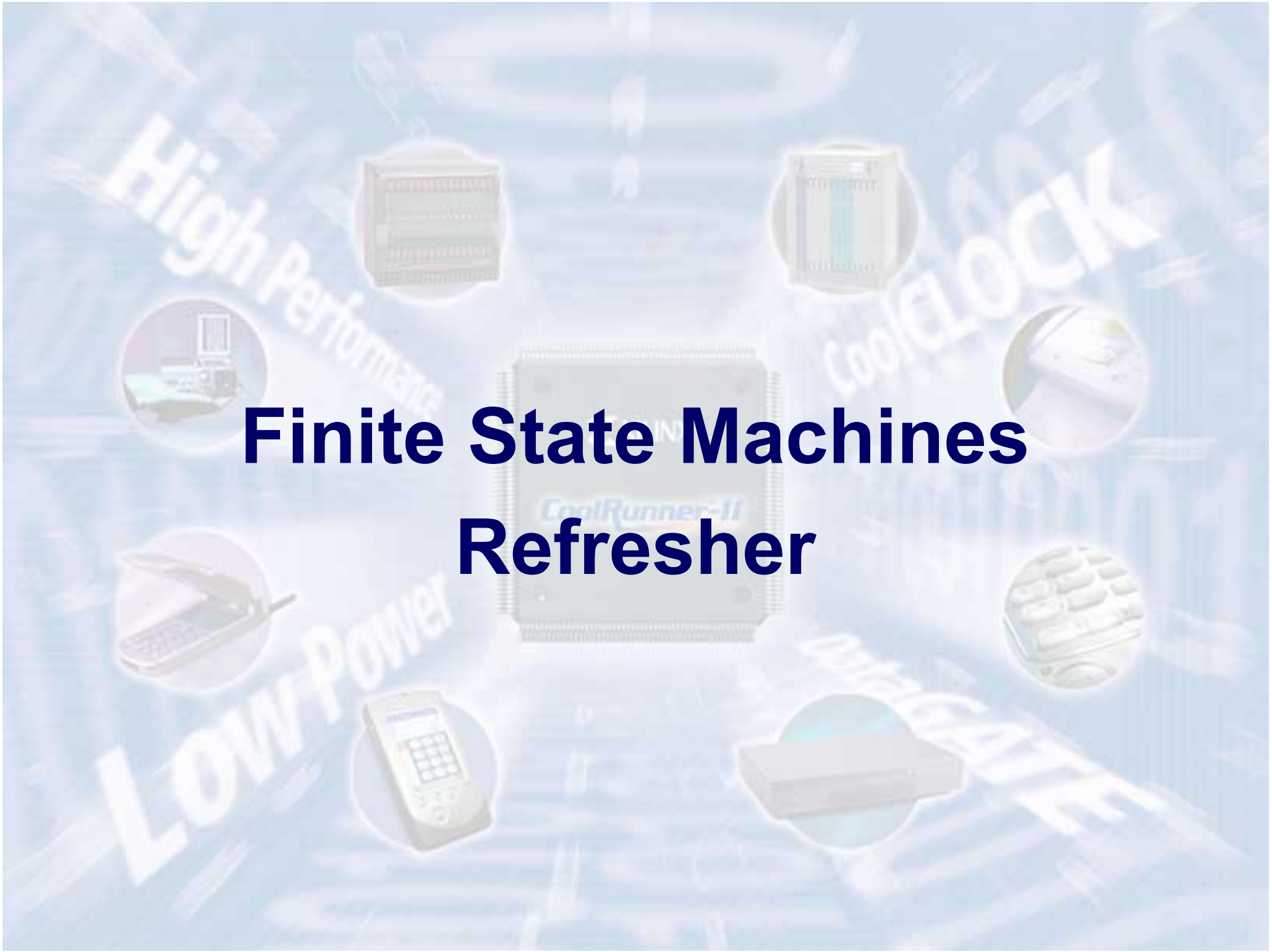
- Controls data movements in the Datapath by switching multiplexers and enabling or disabling resources
  - Example: enable signals for registers
  - Example: select signals for muxes
- Provides signals to activate various processing tasks in the Datapath
- Determines the sequence of operations performed 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 the next instruction
  - Instructions are held in a RAM or ROM
  - Microprocessor is an example of a 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



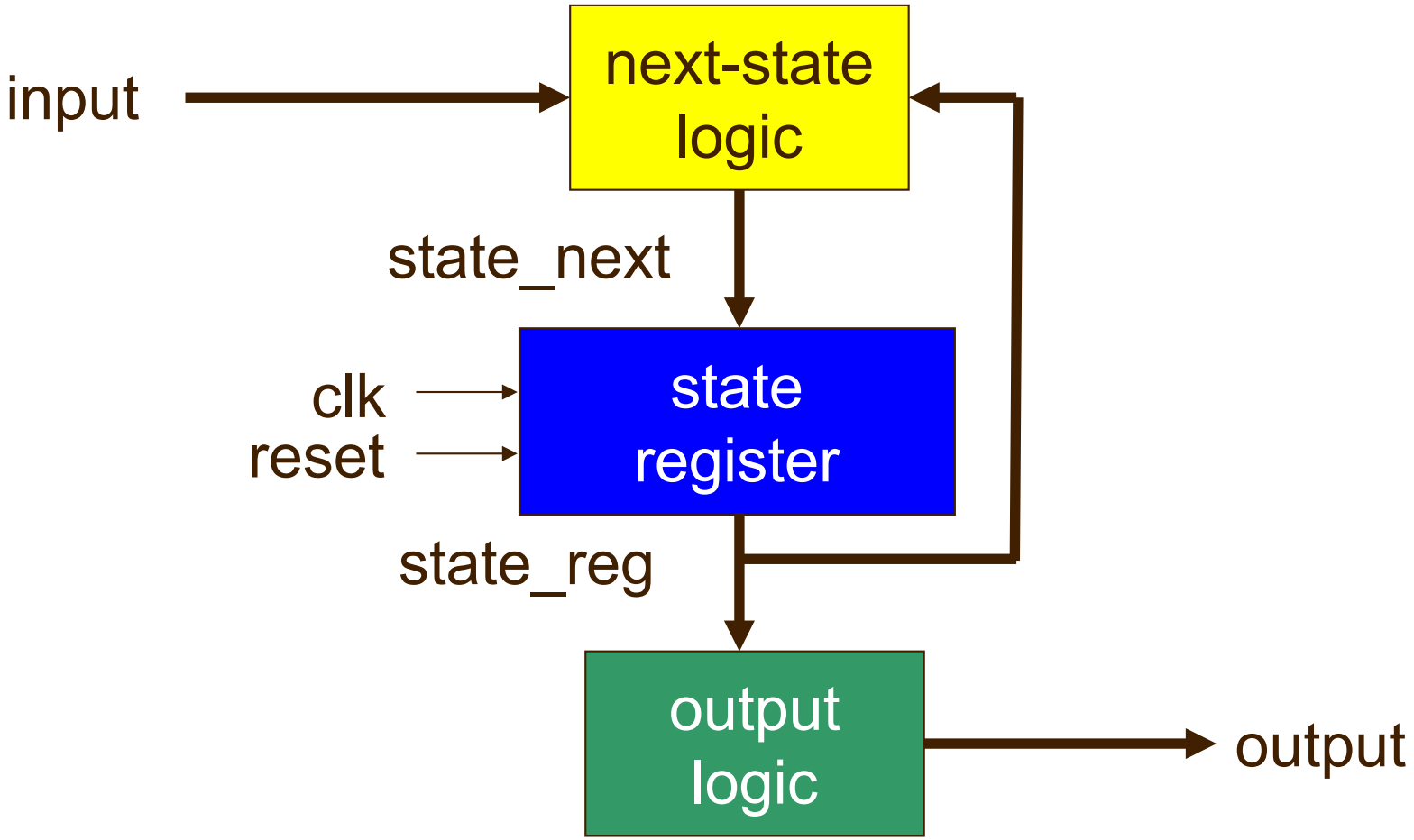
Finite State Machines Refresher

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

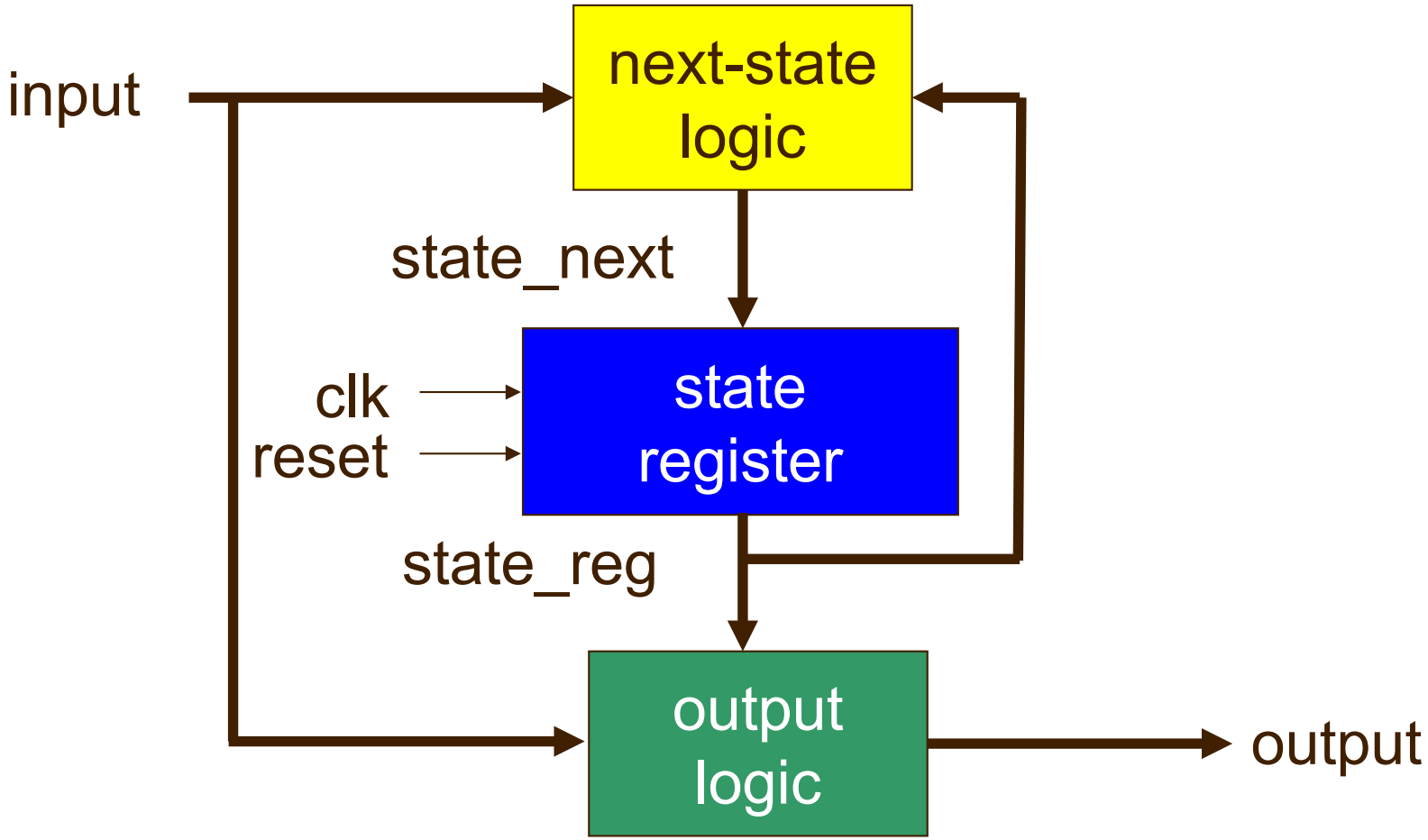
Moore FSM

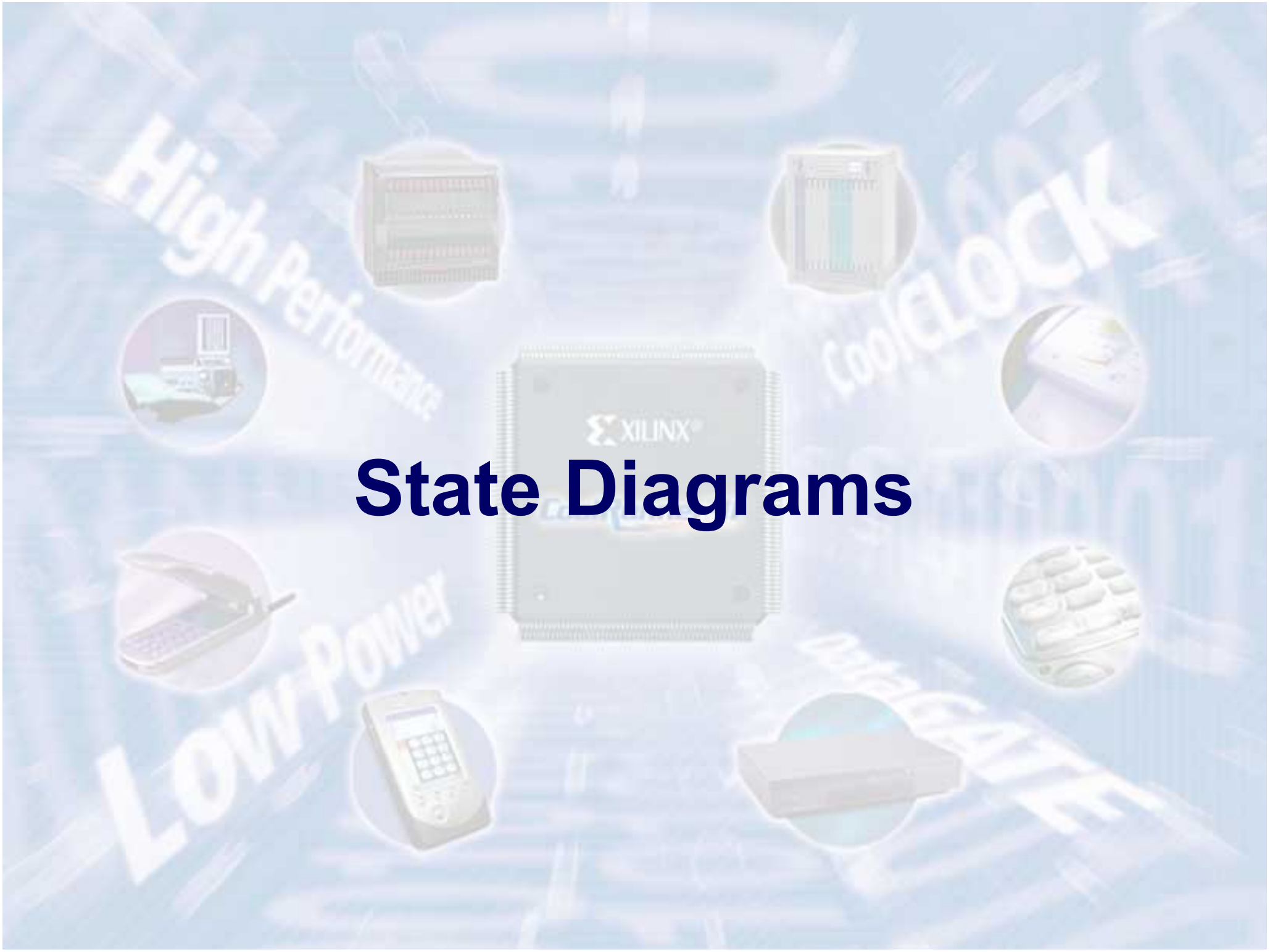
- output is a function of the state only



Mealy FSM

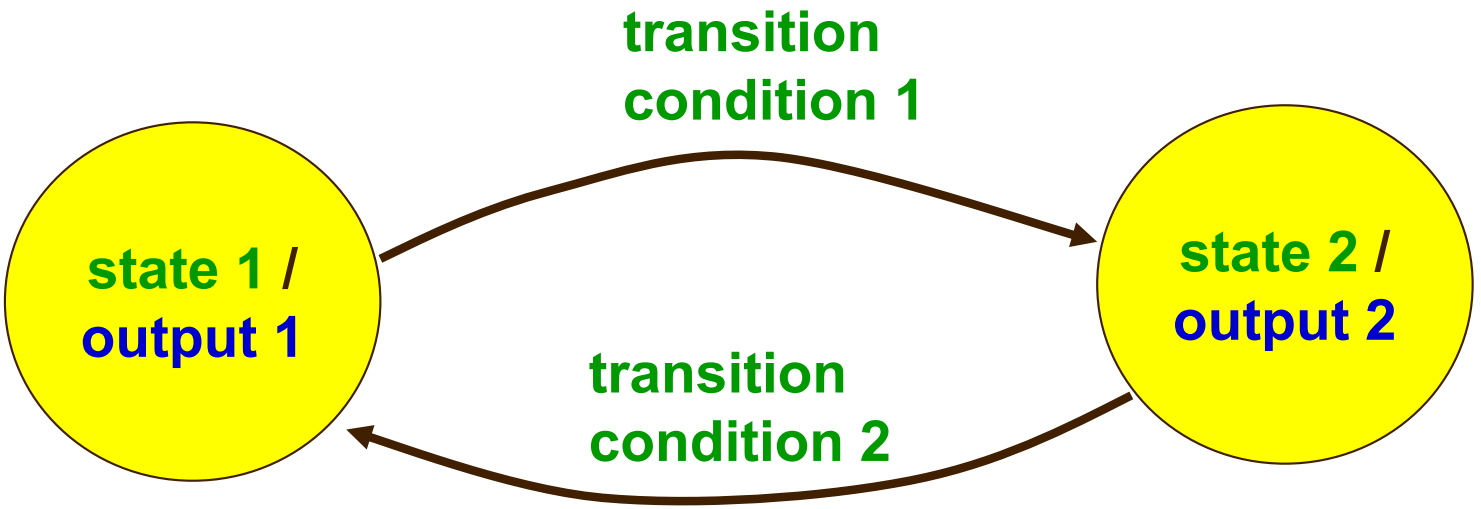
- output is a function of the state and input signals



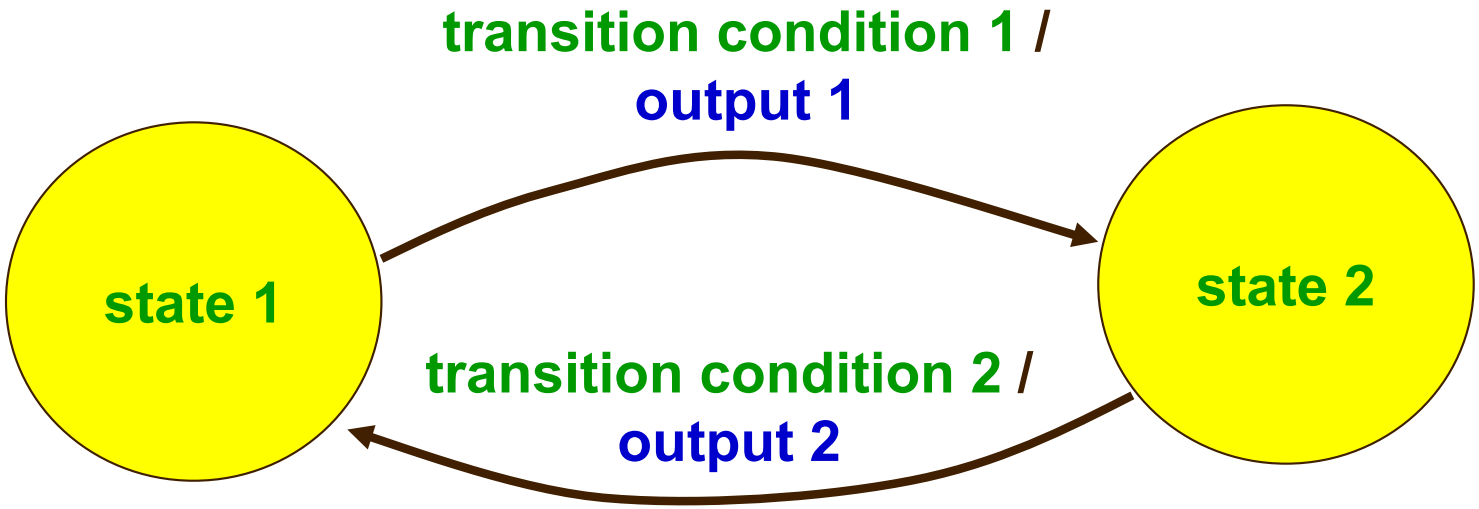


State Diagrams

Moore Machine

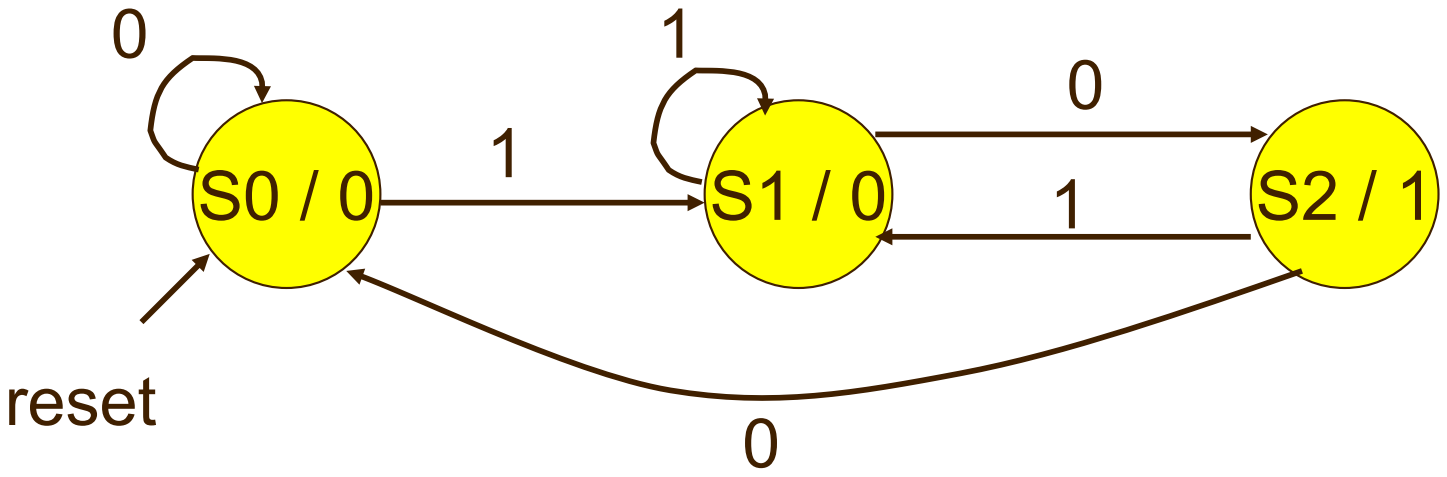


Mealy Machine



Moore FSM - Example 1

- Moore FSM that Recognizes Sequence “10”



Meaning of states:

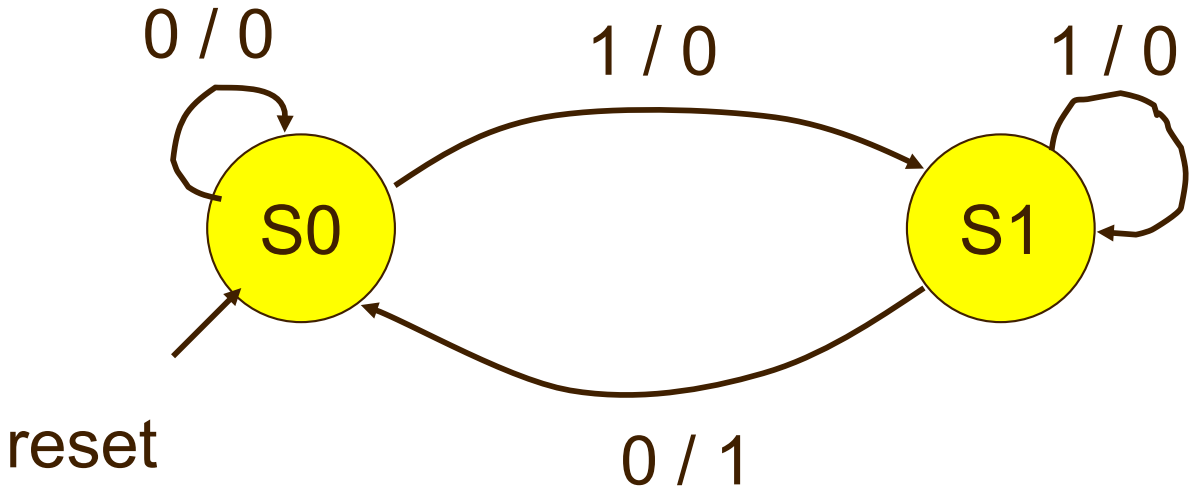
S0: No elements of the sequence observed

S1: “1” observed

S2: “10” observed

Mealy FSM - Example 1

- Mealy FSM that Recognizes Sequence "10"

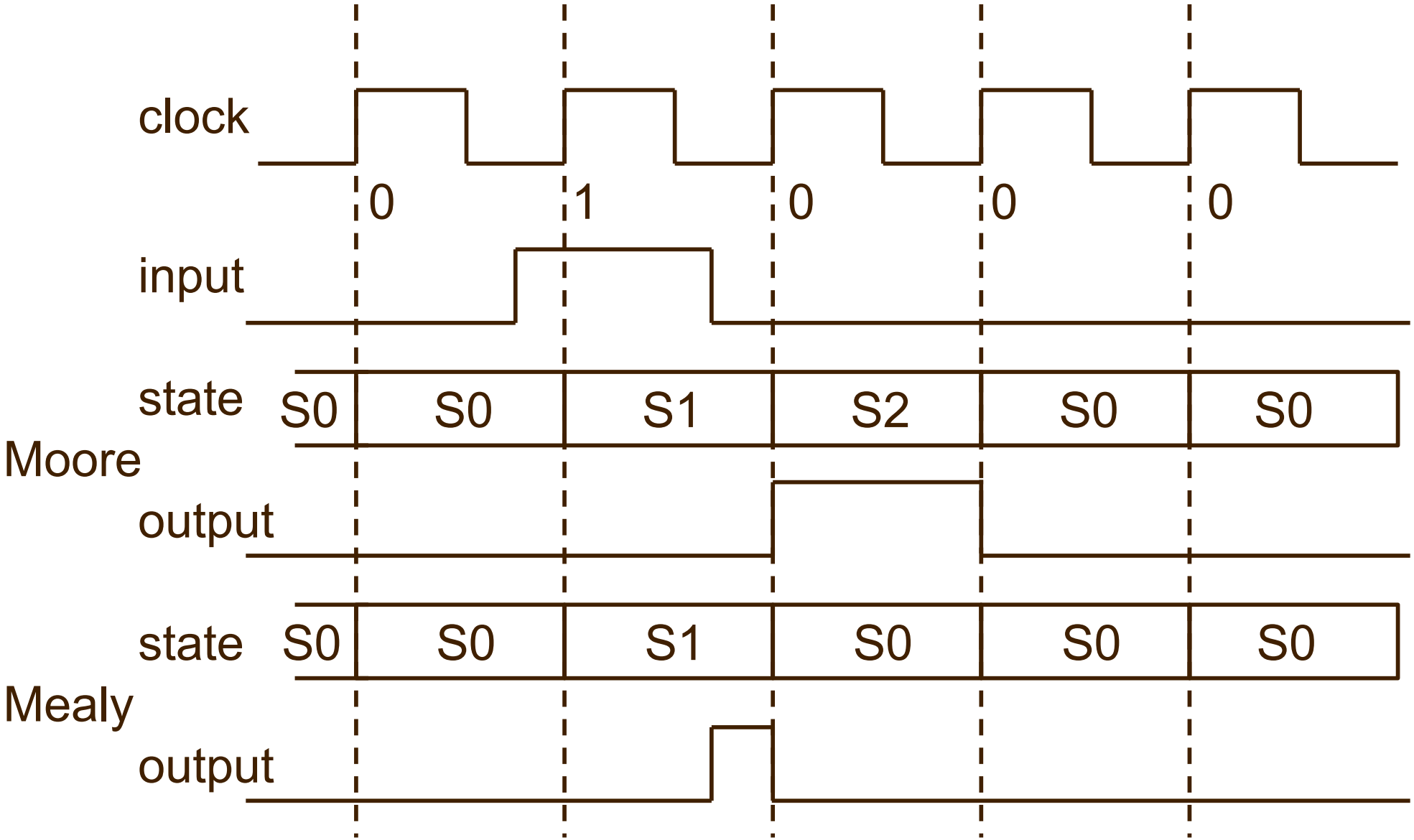


Meaning of states:

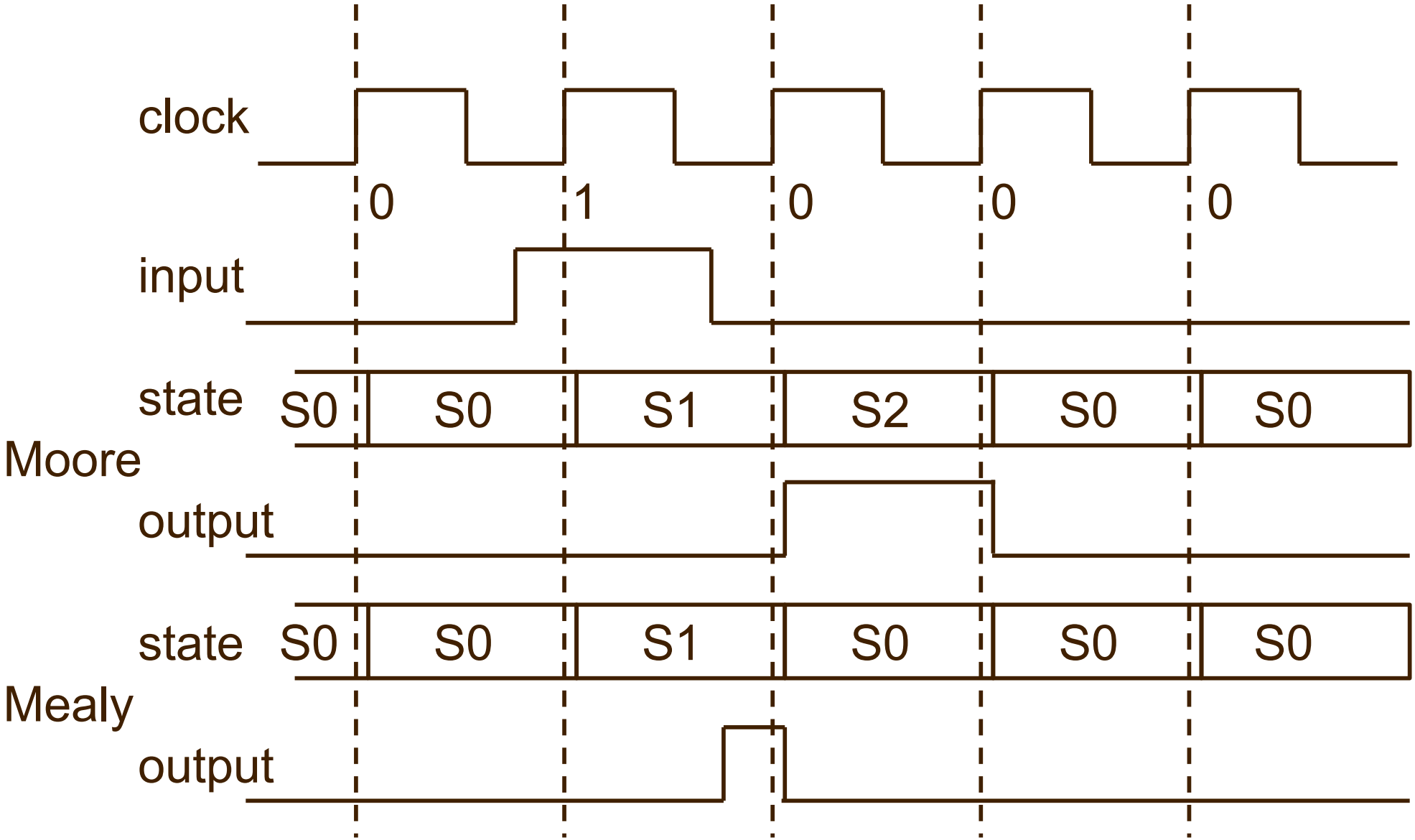
S0: No elements of the sequence observed

S1: "1" observed

Moore & Mealy FSMs without delays



Moore & Mealy FSMs with delays



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

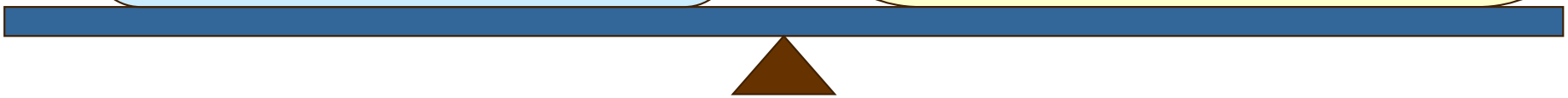
Fewer states

Lower Area

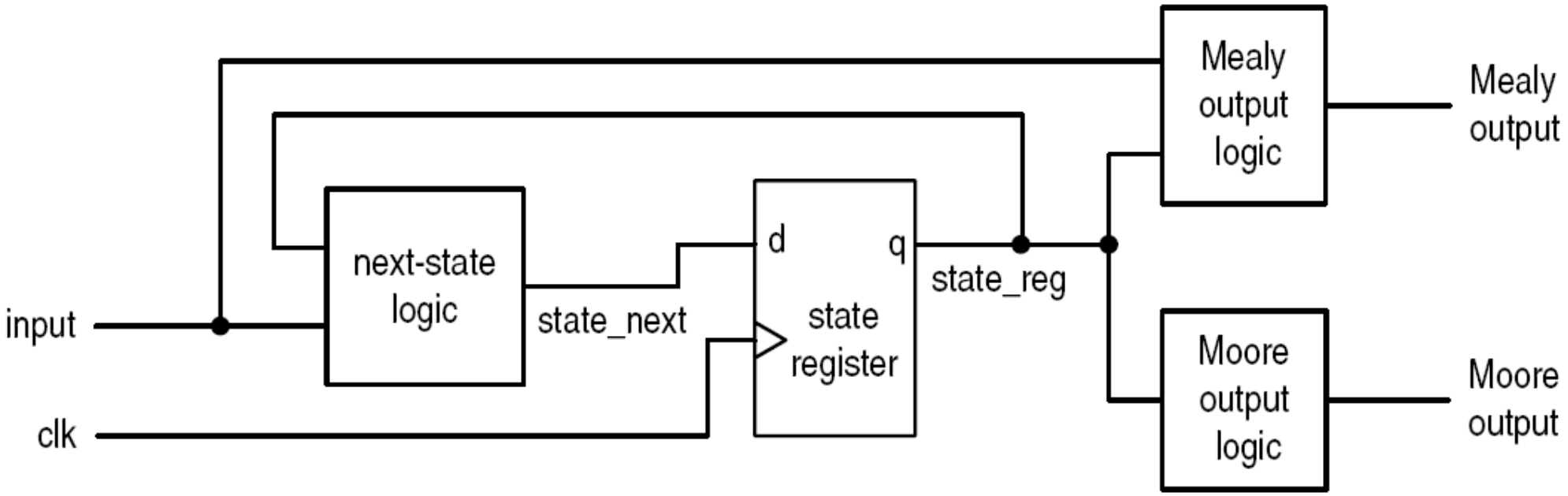
Responds one clock cycle earlier

Moore FSM

Safer.
Less likely to affect the critical path.



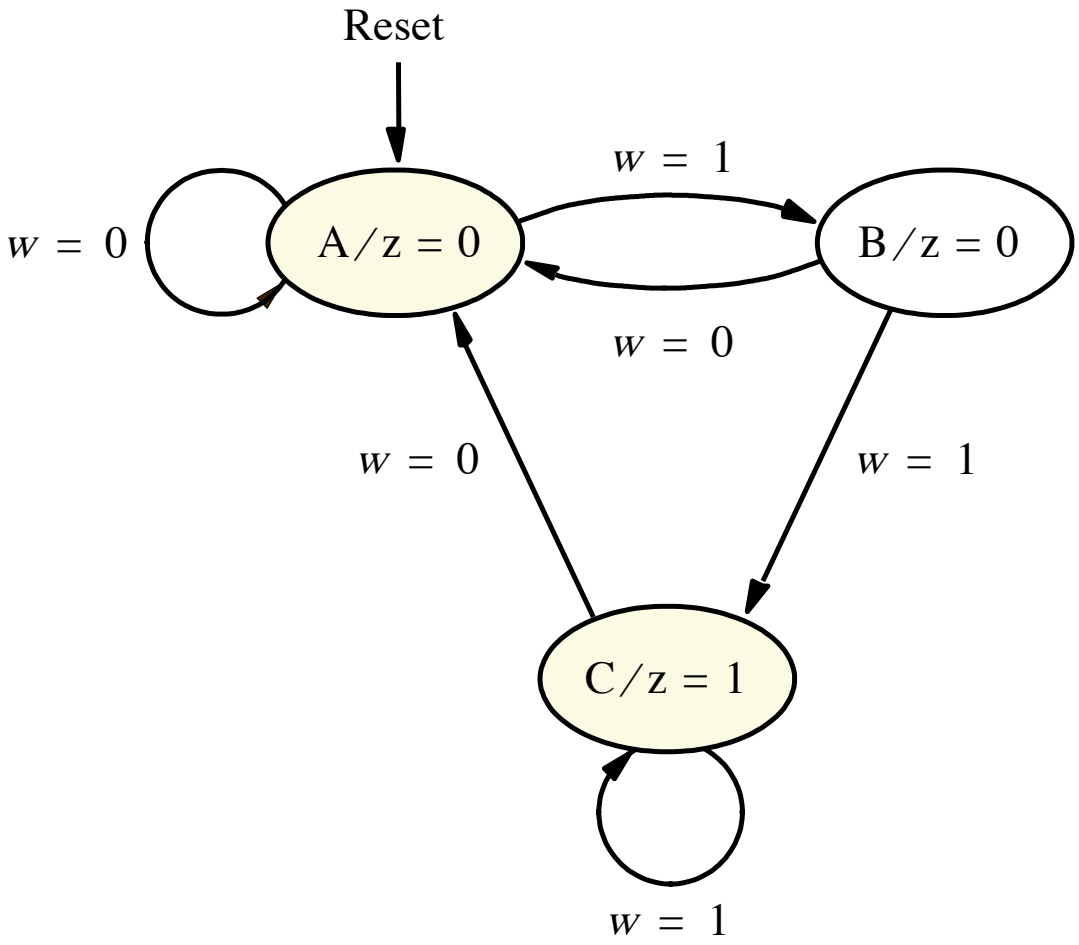
Generalized FSM



Problem 1

Determine a function implemented by
the following state diagram and state table

Moore FSM – Example 1: State diagram

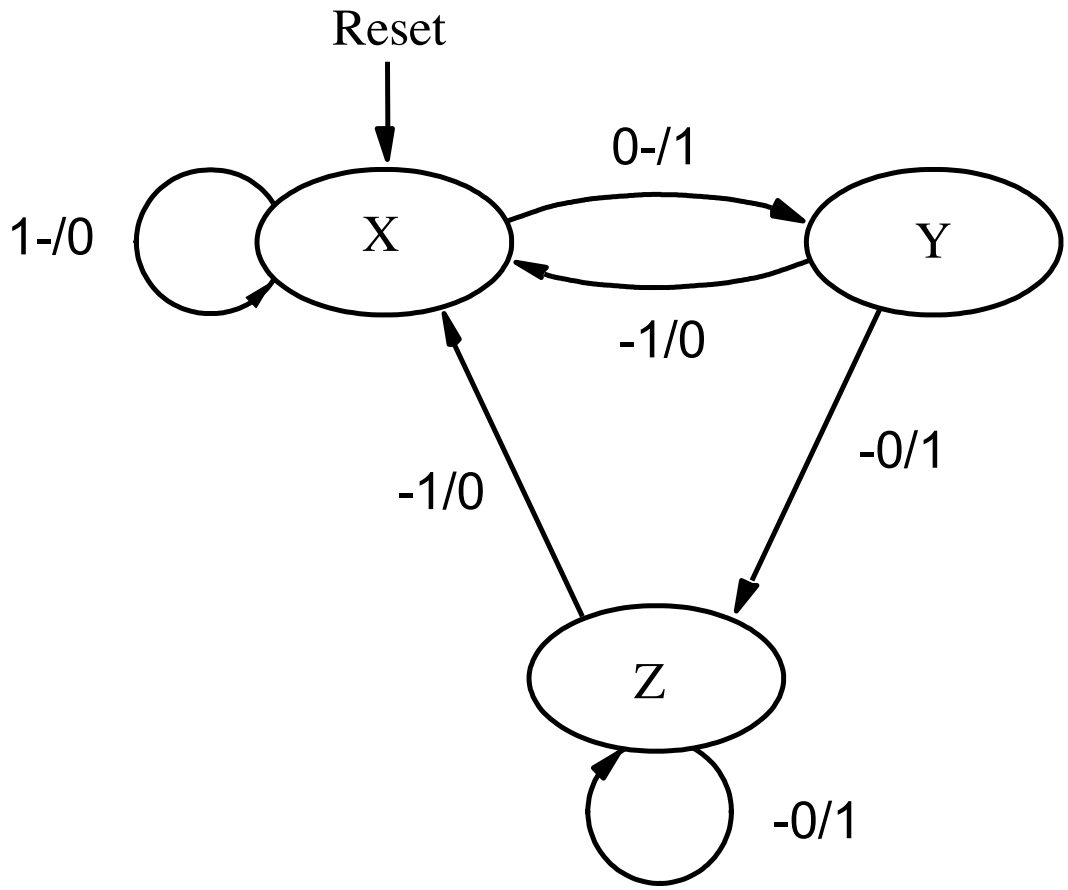


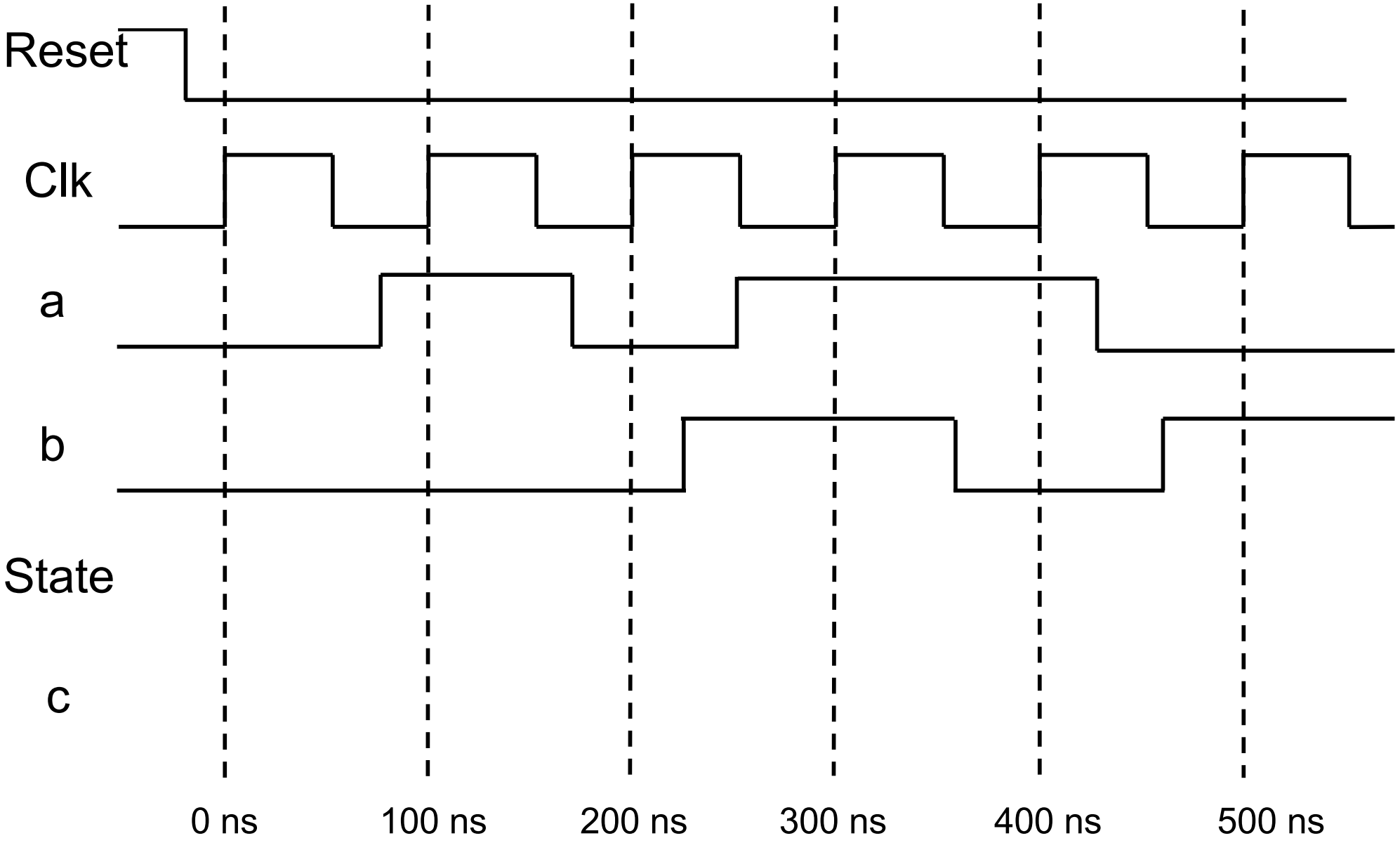
Moore FSM – Example 1: State table

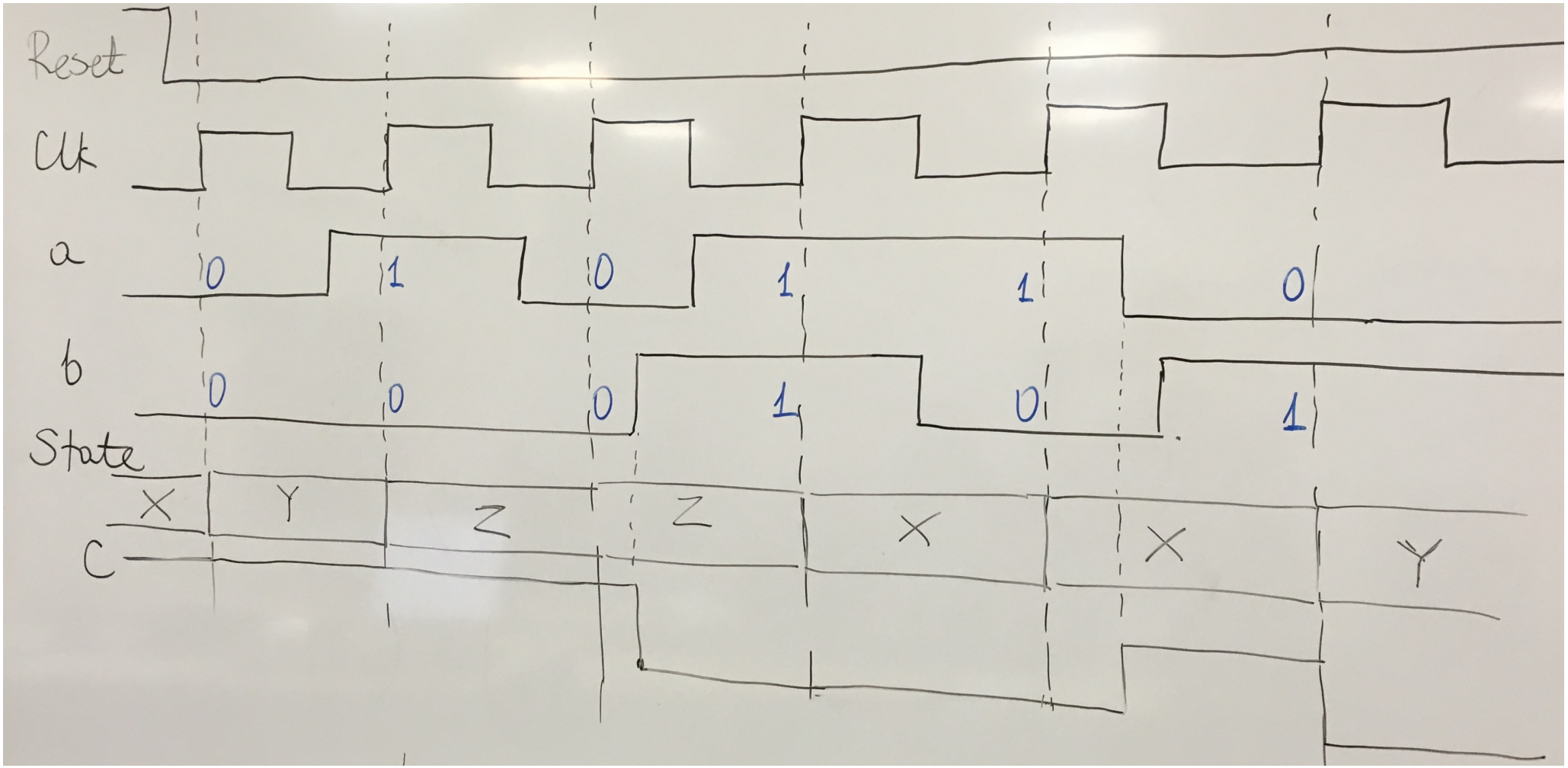
Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Problem 2

Assuming the 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.
Assume zero delays and the asynchronous reset.


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


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


Reset
Clk
a
0
1
0
1
1
0
b
0
0
0
1
0
1
State
X
Y
Z
Z
X
X
Y
C



Algorithmic State Machine (ASM) Charts

Algorithmic State Machine

Algorithmic State Machine –

representation of a Finite State Machine

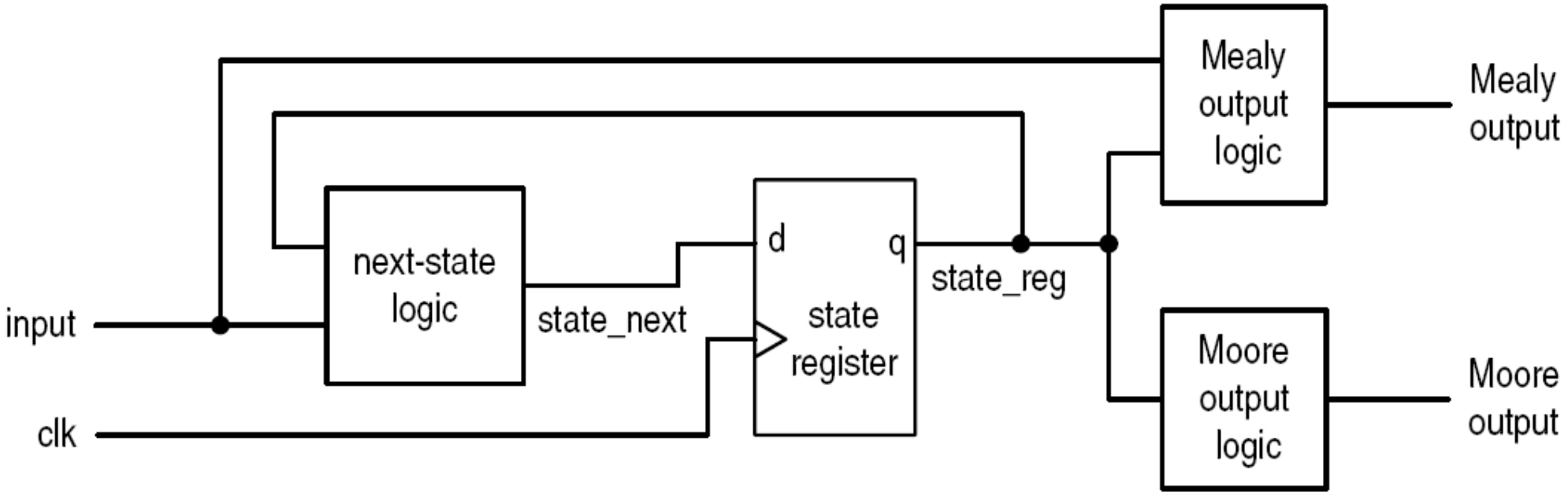
suitable for FSMs with a larger number of inputs and outputs compared to FSMs expressed using state diagrams and state tables.

ASM Chart

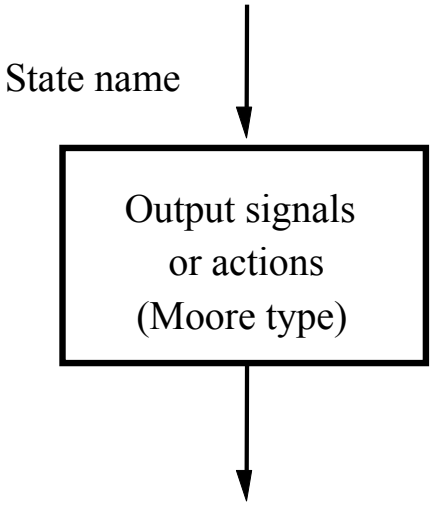
- Flowchart-like diagram
- Provides the same info as a state diagram
- More descriptive, better for complex description

ASM describing generalized FSM

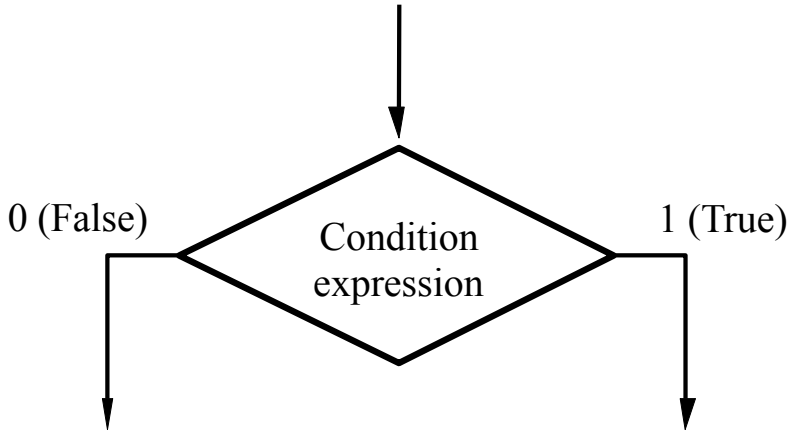
- Algorithmic state machines can model both Mealy and Moore Finite State Machines
- They can also model generalized machines that are of the mixed type



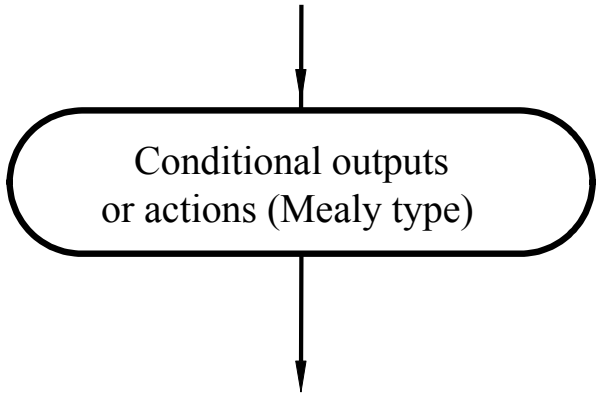
Elements used in ASM charts (1)



(a) State box



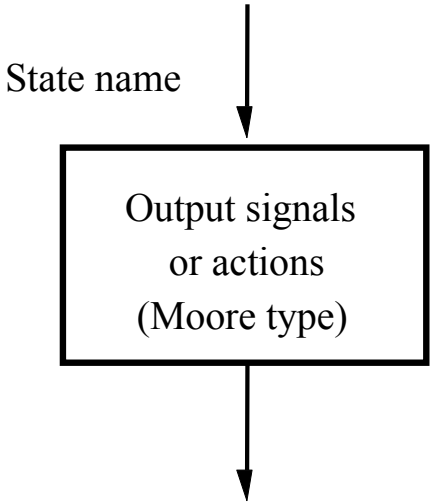
(b) Decision box



(c) Conditional output box

State Box

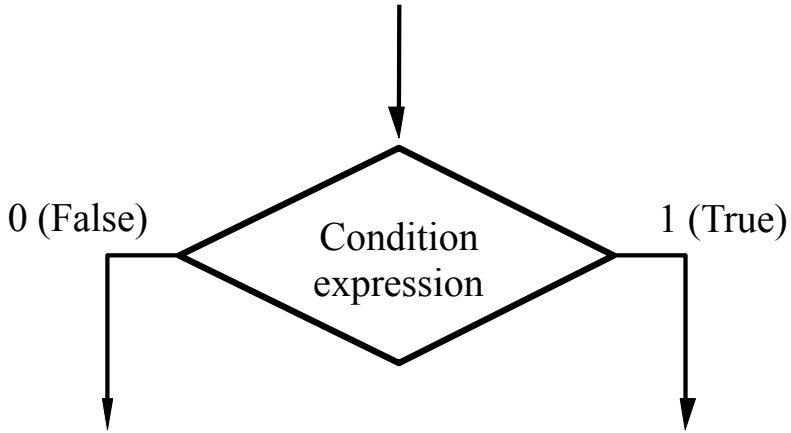
- **State box** – represents a state.
- Equivalent to a node in a state diagram or a row in a state table.
- Contains register transfer actions or output signals
- **Moore-type outputs are listed inside of the box.**
- It is customary to write only the name of the signal that has to be asserted in the 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

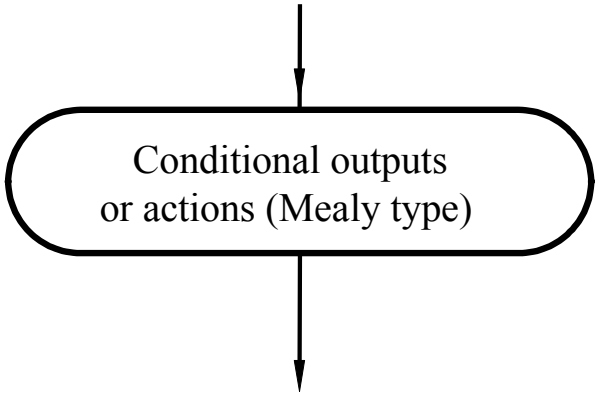
- **Decision box** – indicates that a given condition is to be tested and the exit path is to be chosen accordingly.

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

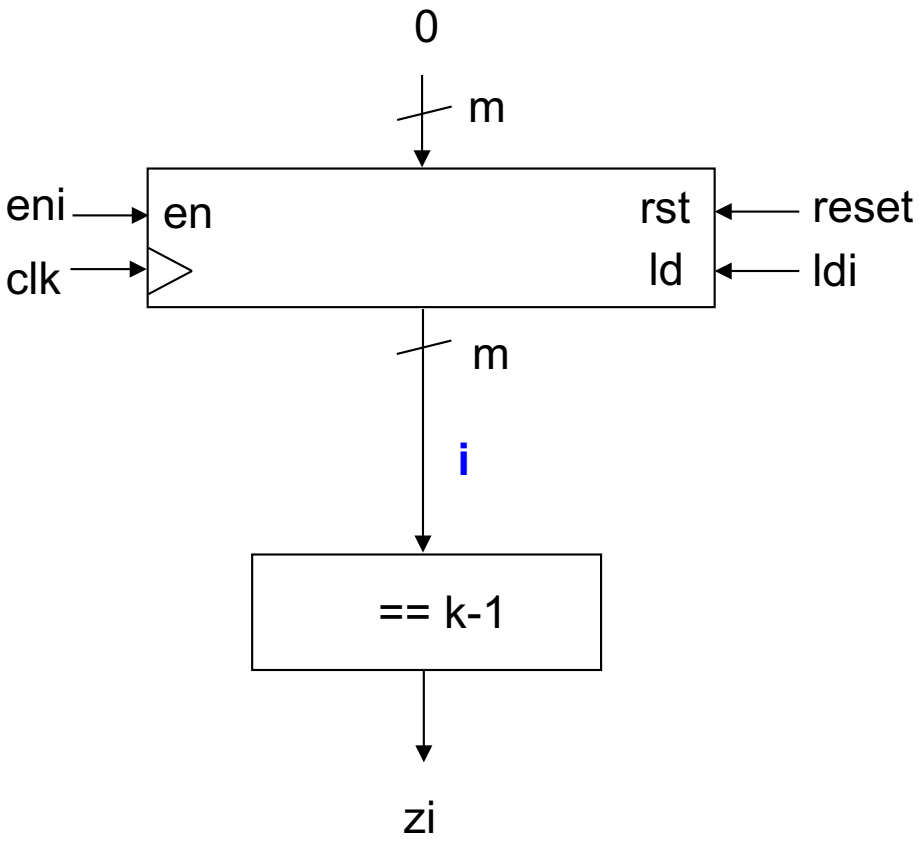


Conditional Output Box

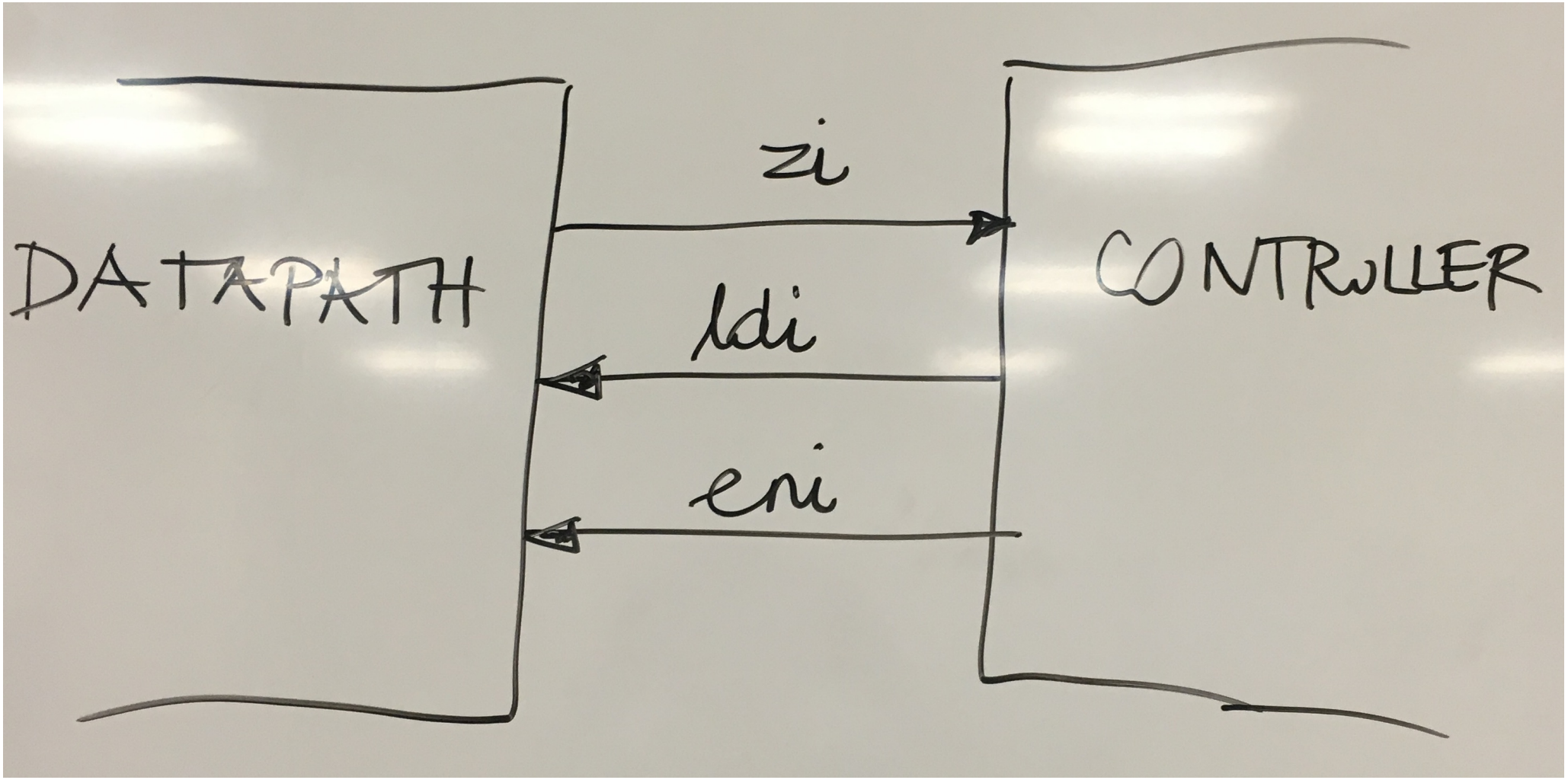
- **Conditional output box**
- **Denotes output signals that are of the Mealy type.**
- The condition that determines whether such outputs are generated is specified in the decision box.



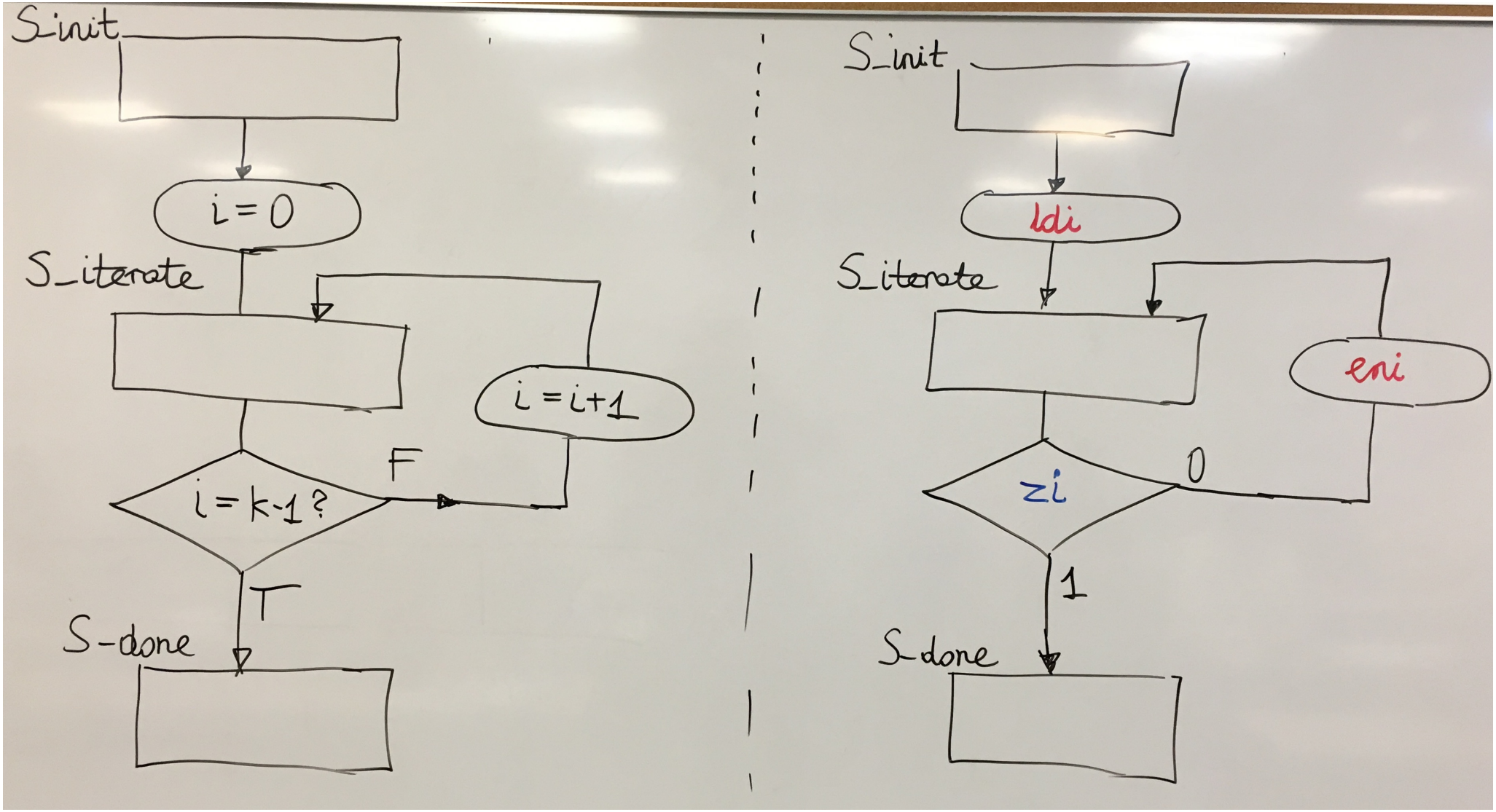
Example: Counter



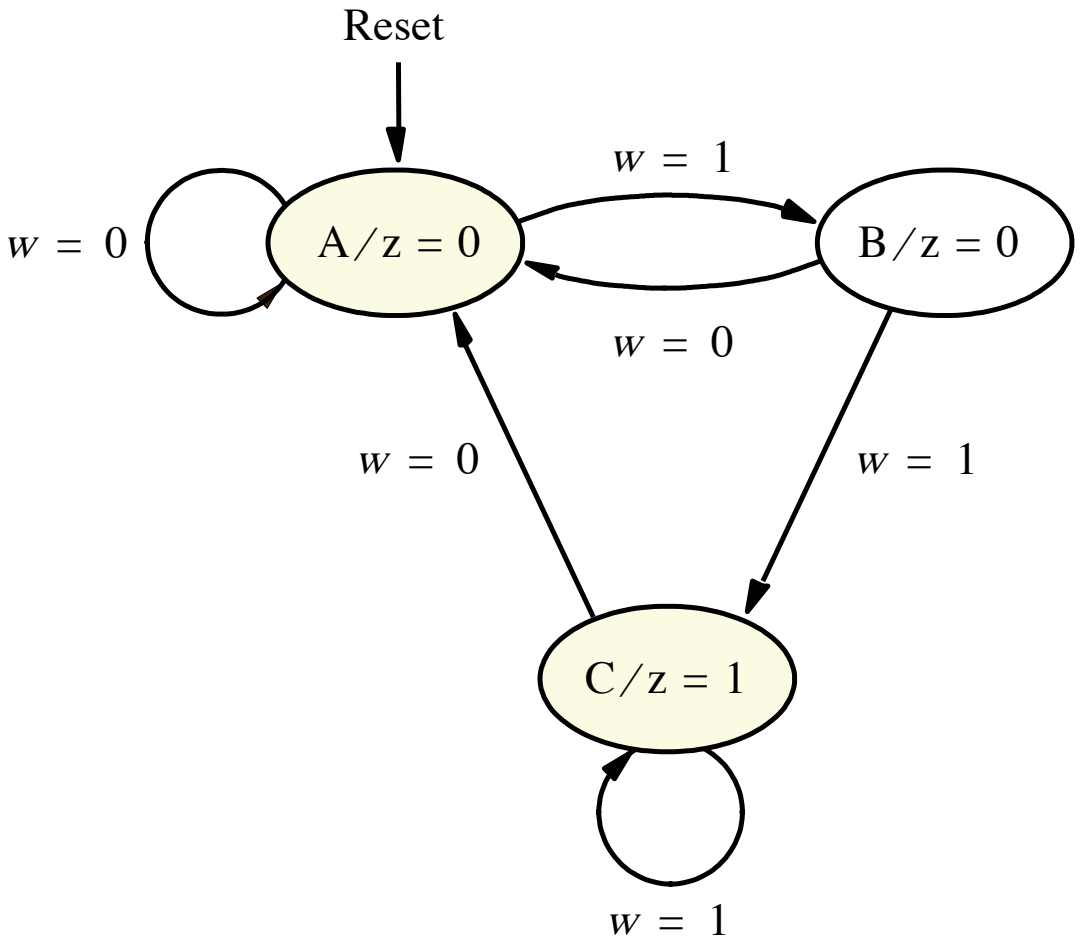
Counting: Datapath + Controller



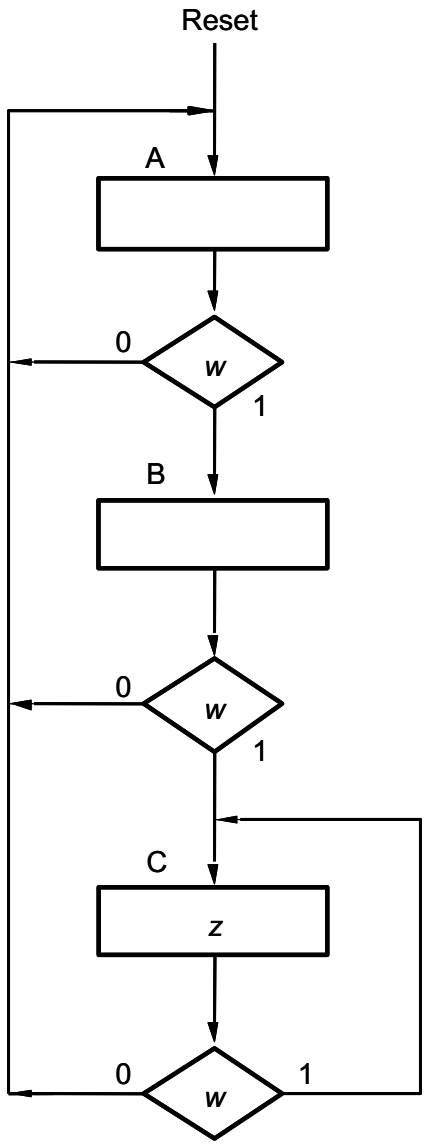
Counting: Block diagram + ASM



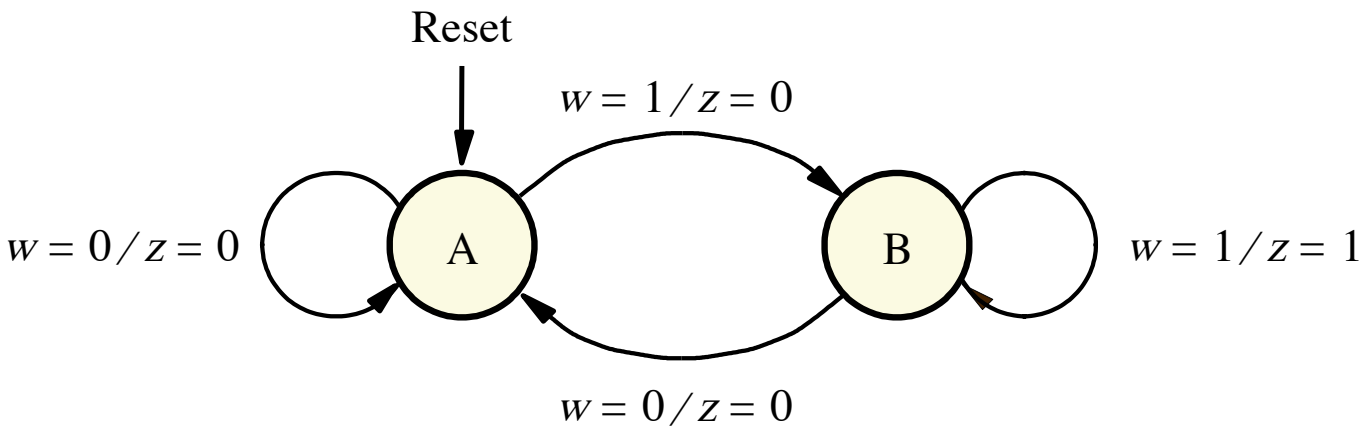
Example 1: State Diagram



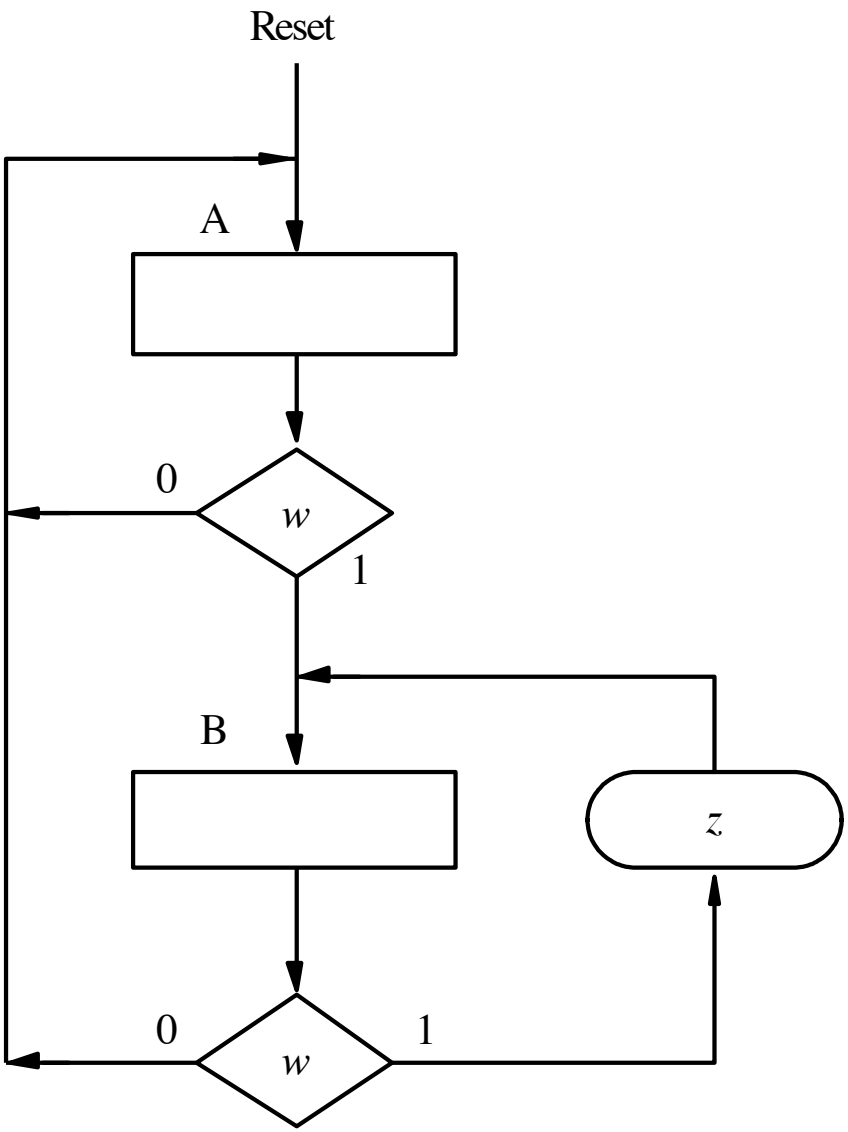
Example 1: ASM Chart



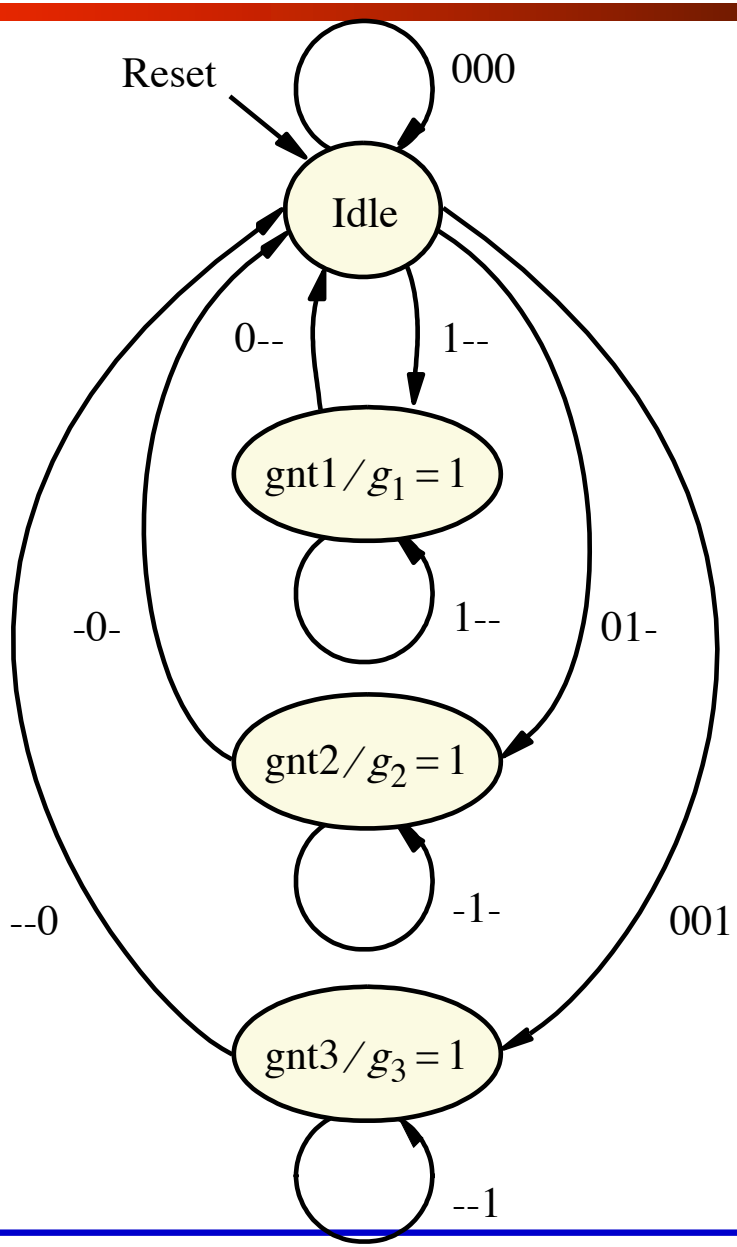
Example 2: State Diagram



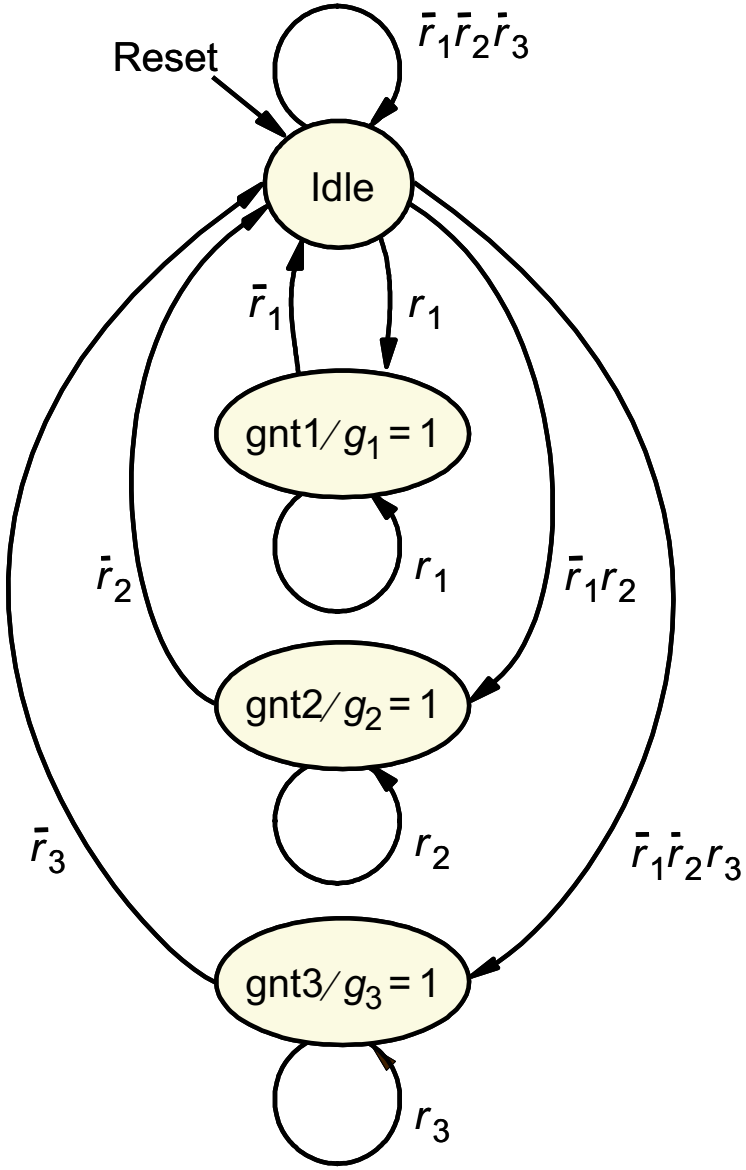
Example 2: ASM Chart



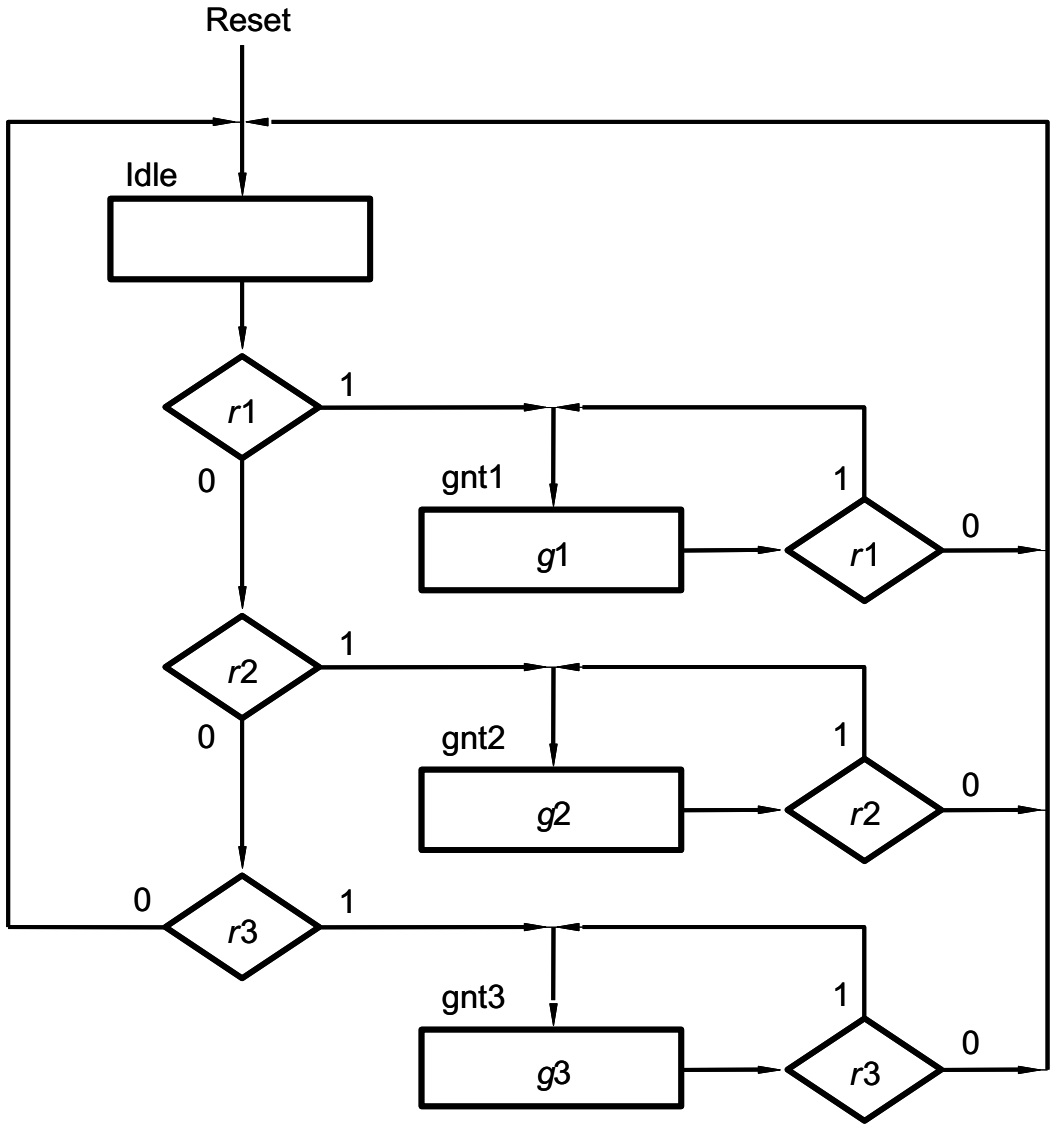
Example 3: State Diagram



Example 3: State Diagram



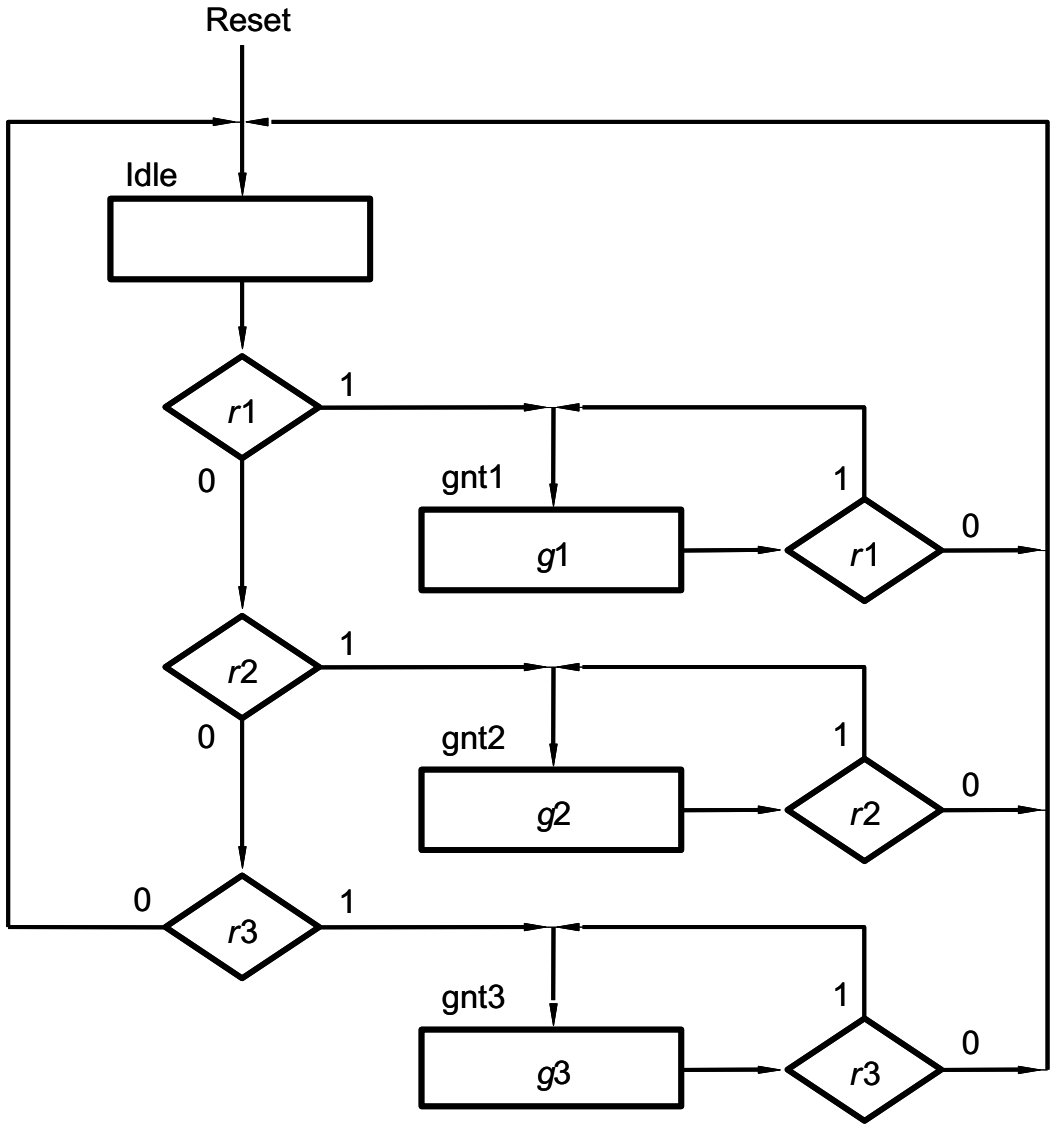
Example 3: State Diagram

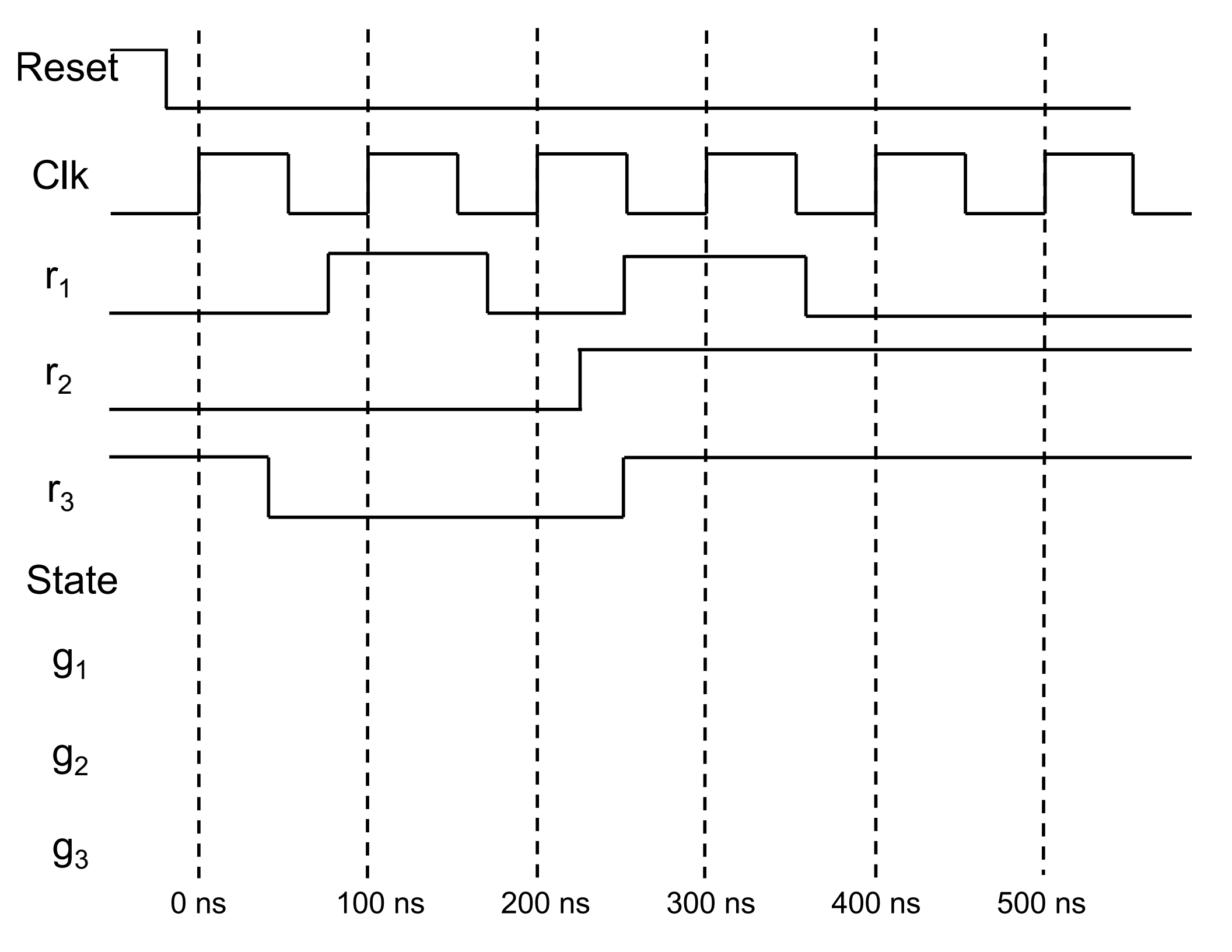


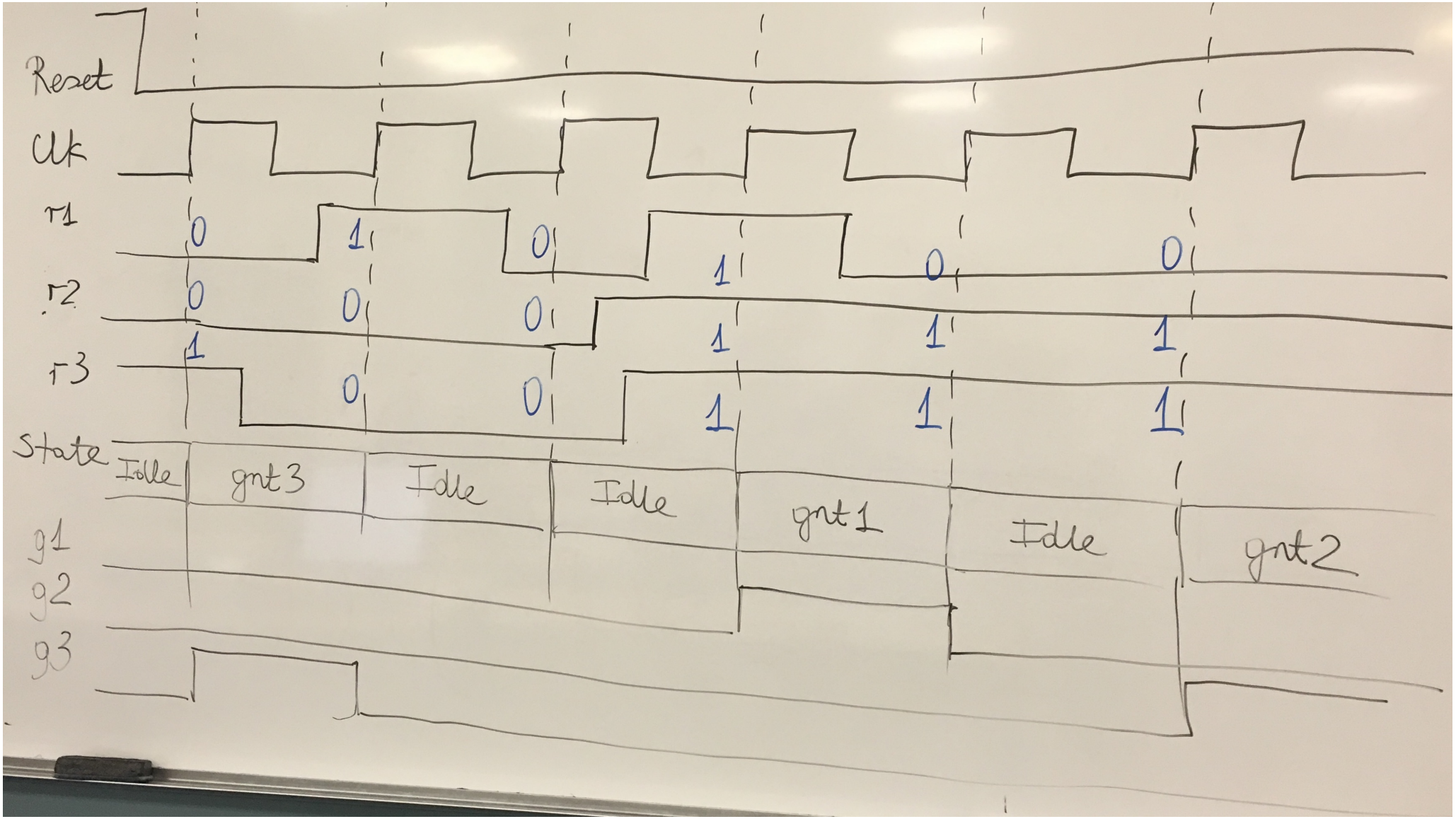
Exercise

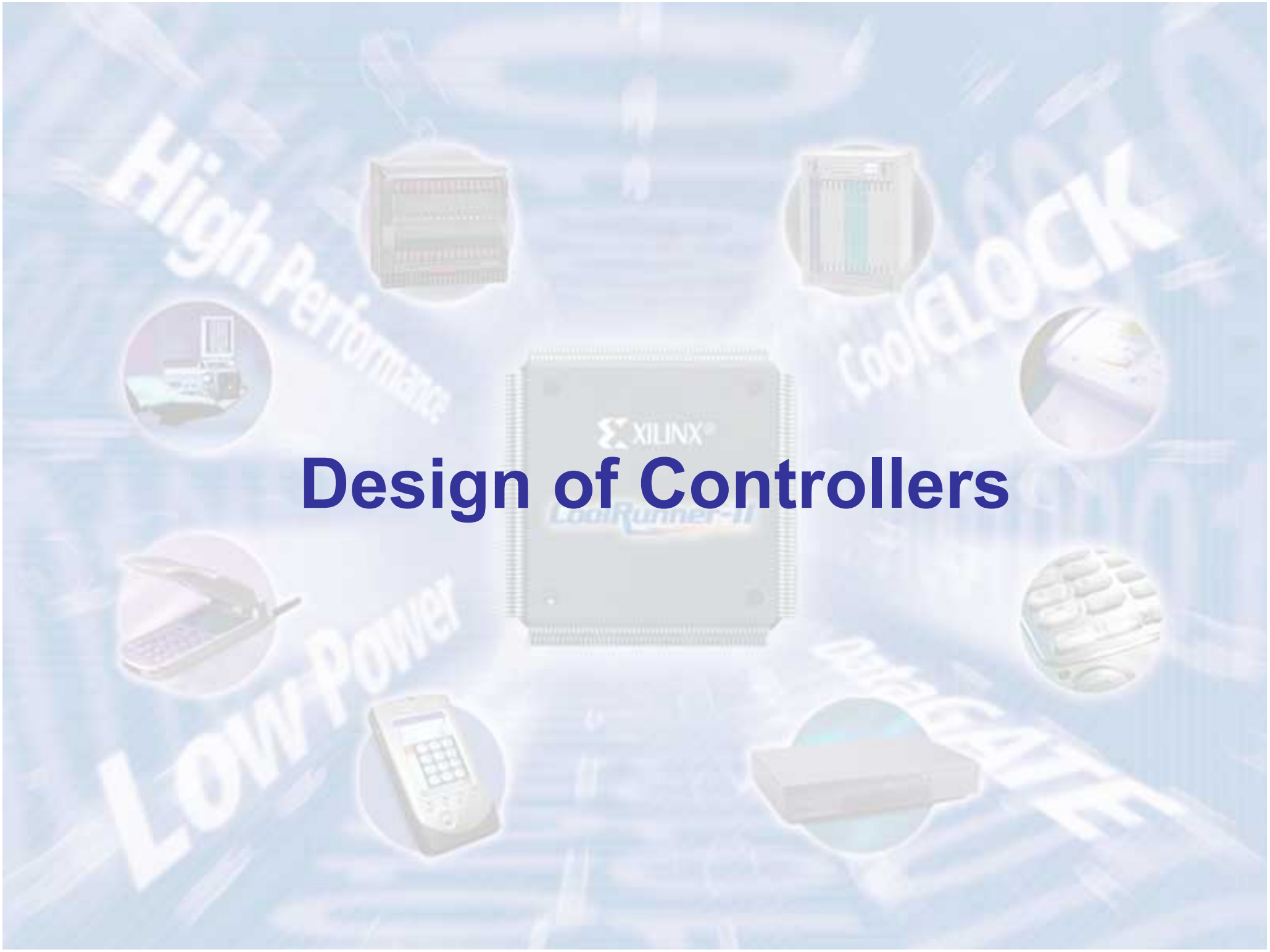
Assuming ASM chart given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals **State, g1, g2, g3**, in the interval from 0 to 575 ns. Assume zero delays.

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

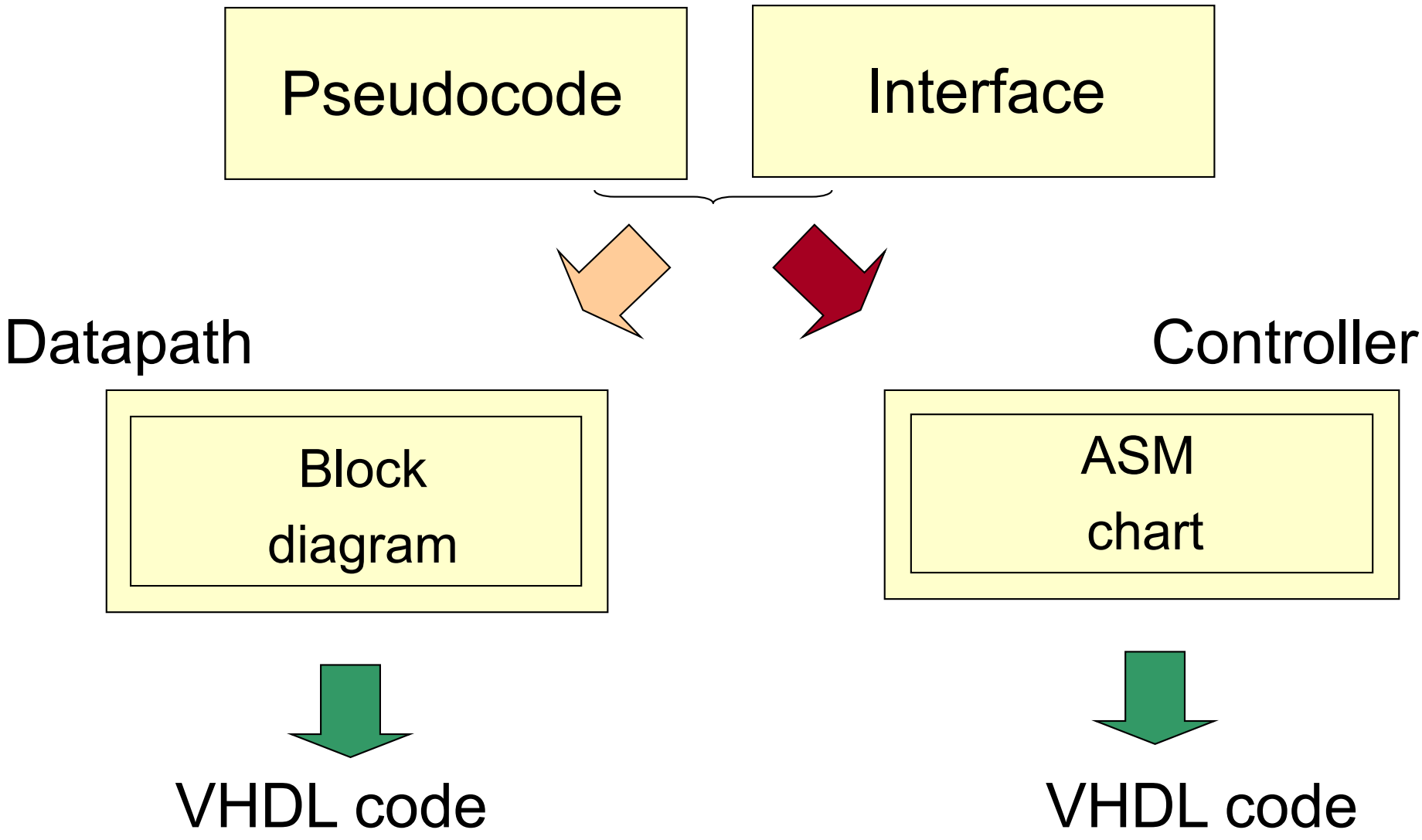

Reset
Clk
r1
r2
r3
State
g1
g2
g3
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns


Reset
Clk
r1
r2
r3
State
Idle
gnt3
Idle
Idle
gnt1
Idle
gnt2
g1
g2
g3



Design of Controllers

Hardware Design with RTL VHDL



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

ASM Block

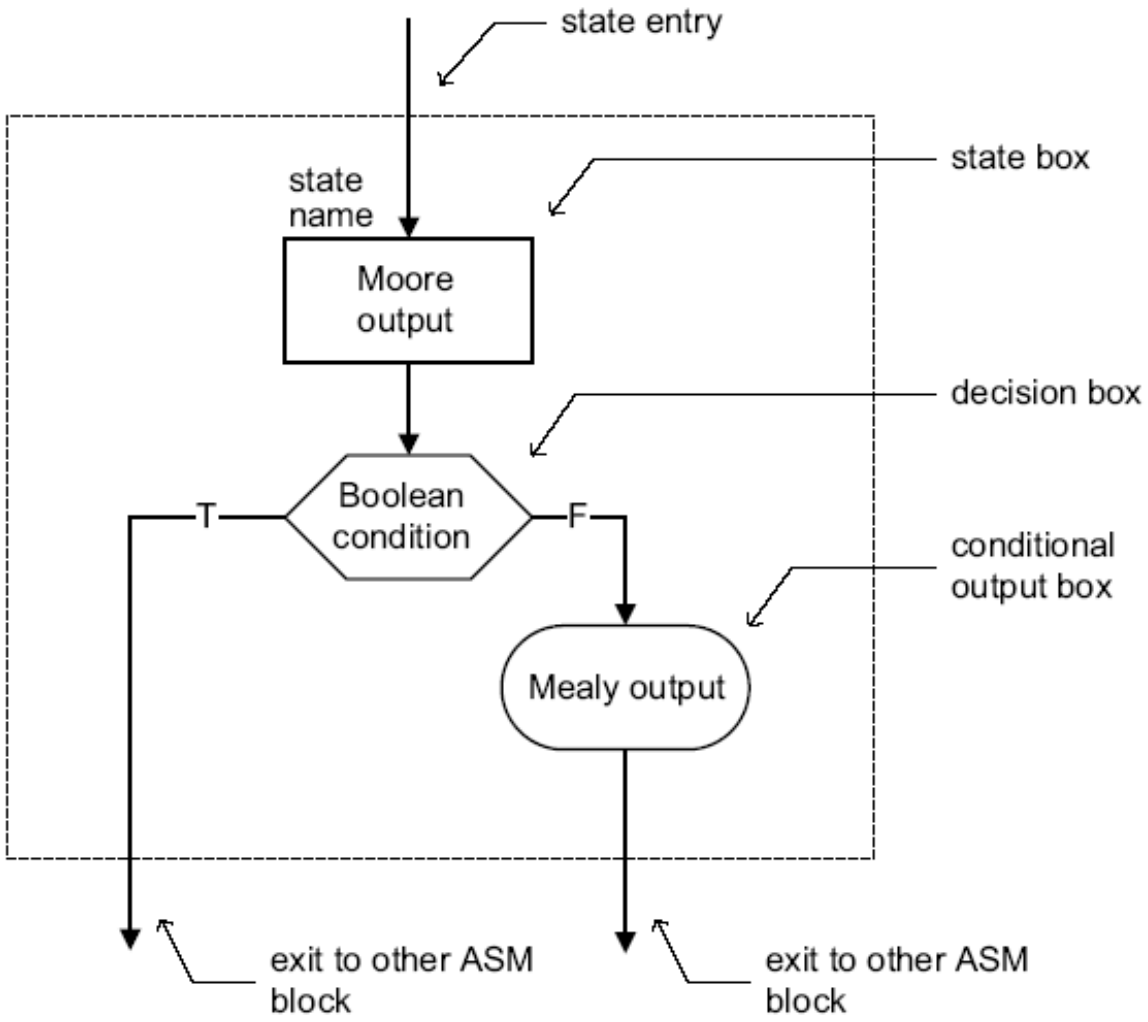


Figure 10.4 ASM block.

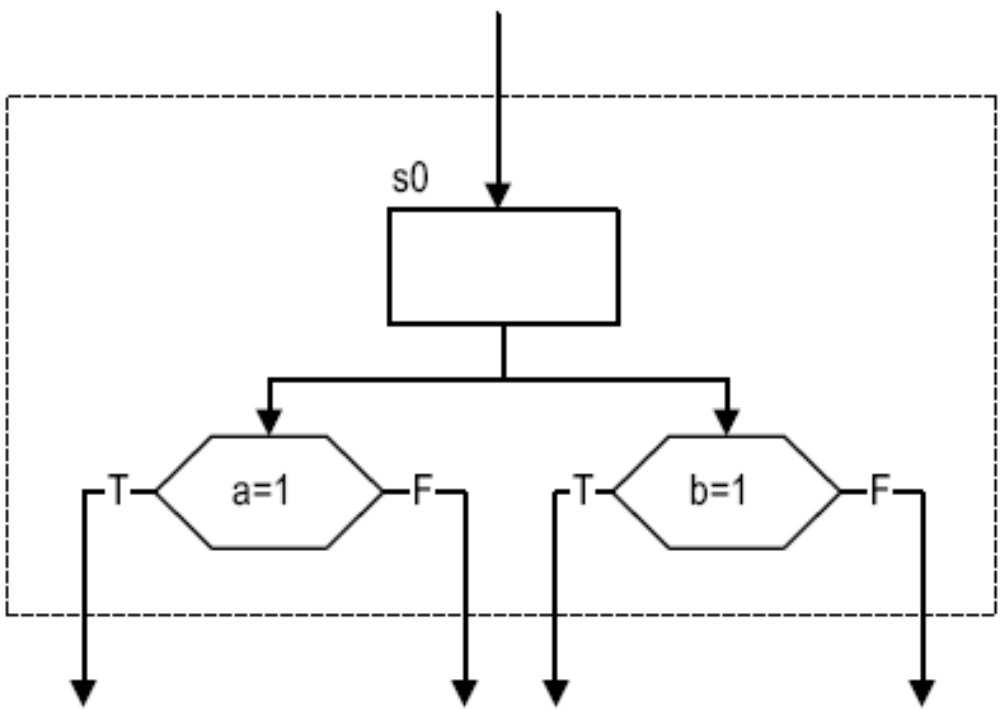
- One state box
- One or more (optional) decision boxes: with T (1) or F (0) exit paths
- One or more (optional) conditional output boxes: for Mealy outputs

ASM Chart Rules

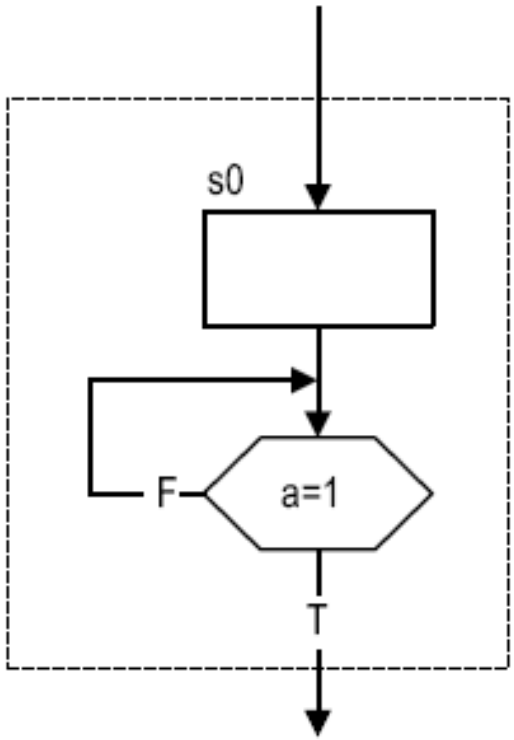
- Difference between a regular flowchart and an ASM chart:
  - Transition governed by clock
  - Transition occurs between ASM blocks
- Basic rules:
  - For a given input combination, there is one unique exit path from the current ASM block
  - Any closed loop in an ASM chart must include a state box

Based on RTL Hardware Design by P. Chu

Incorrect ASM Charts



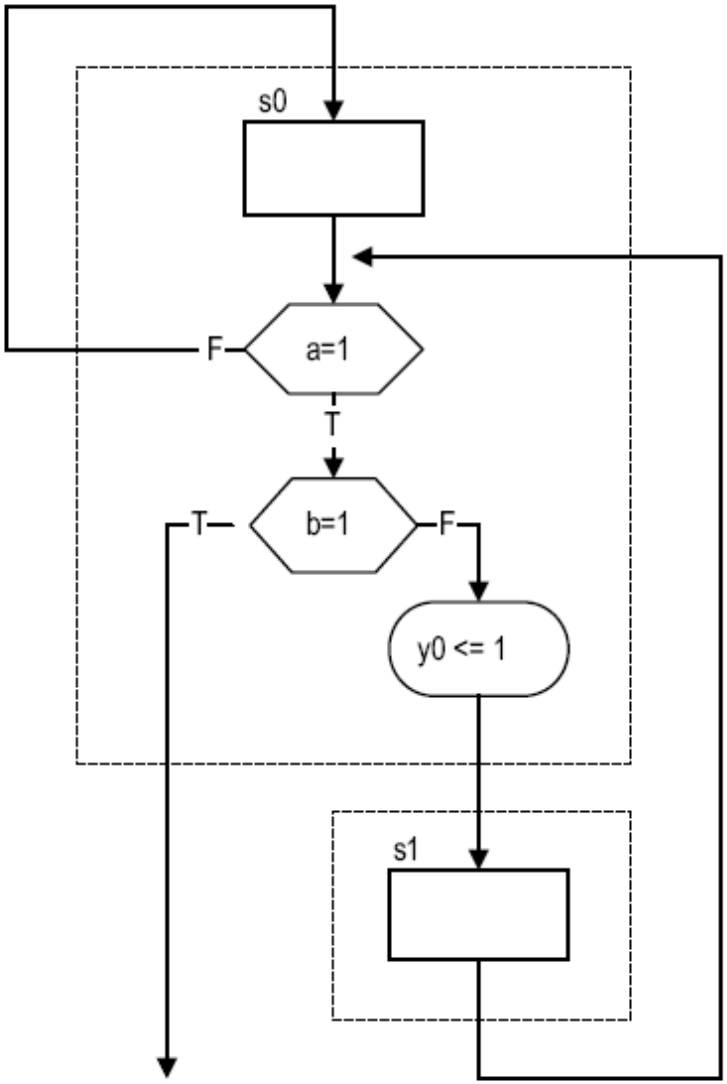
(a)



(b)

Based on RTL Hardware Design by P. Chu

Correct ASM Chart



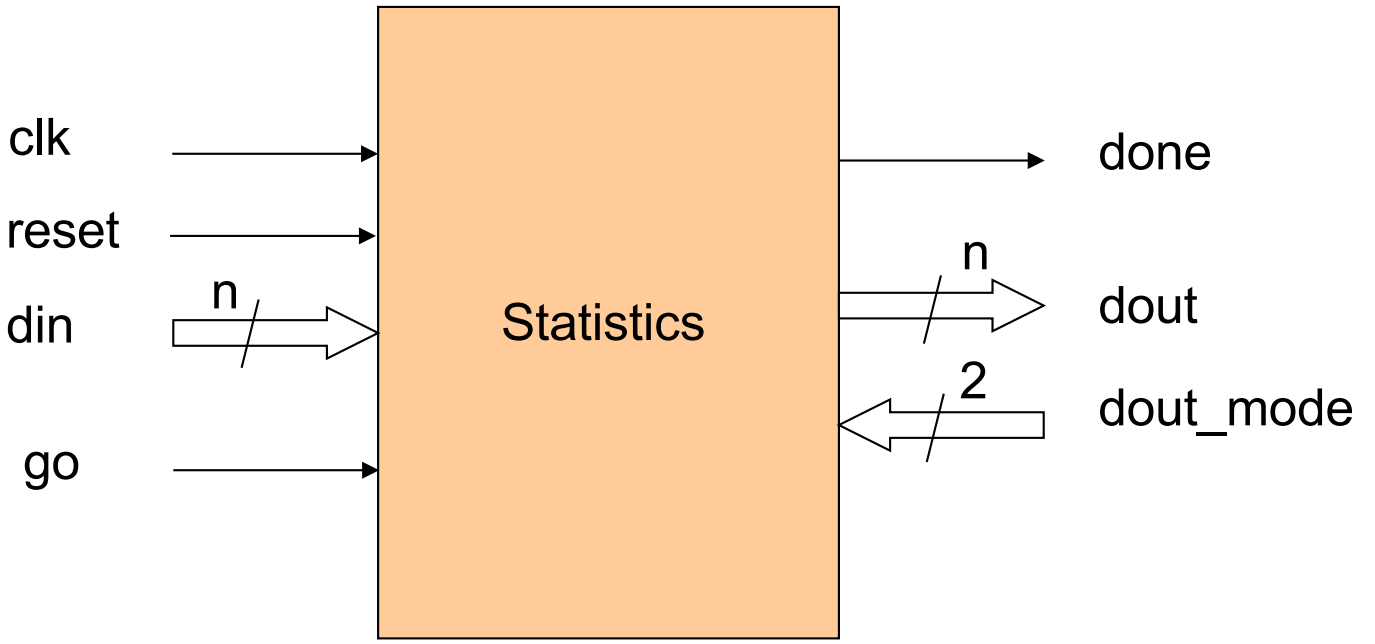


Class Exercise 1
STATISTICS

Pseudocode

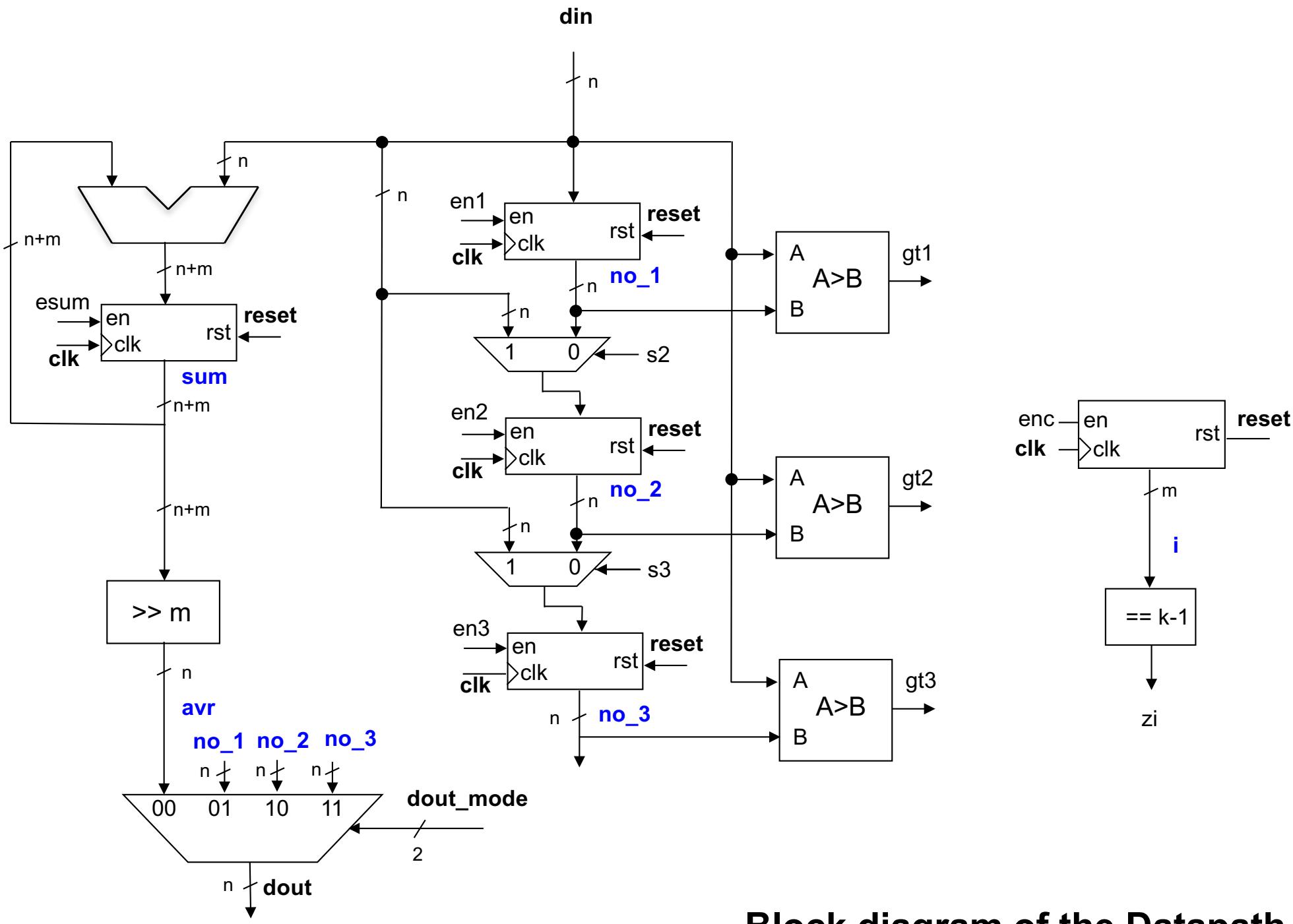
```
no_1 = no_2 = no_3 = sum = 0
for i=0 to k-1 do
    sum = sum + din
    if  din > no_1 then
            no_3 = no_2
            no_2 = no_1
            no_1 = din
    elseif (din > no_2) then
            no_3 = no_2
            no_2 = din
    elseif (din > no_3) then
            no_3 = din
    end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

Circuit Interface



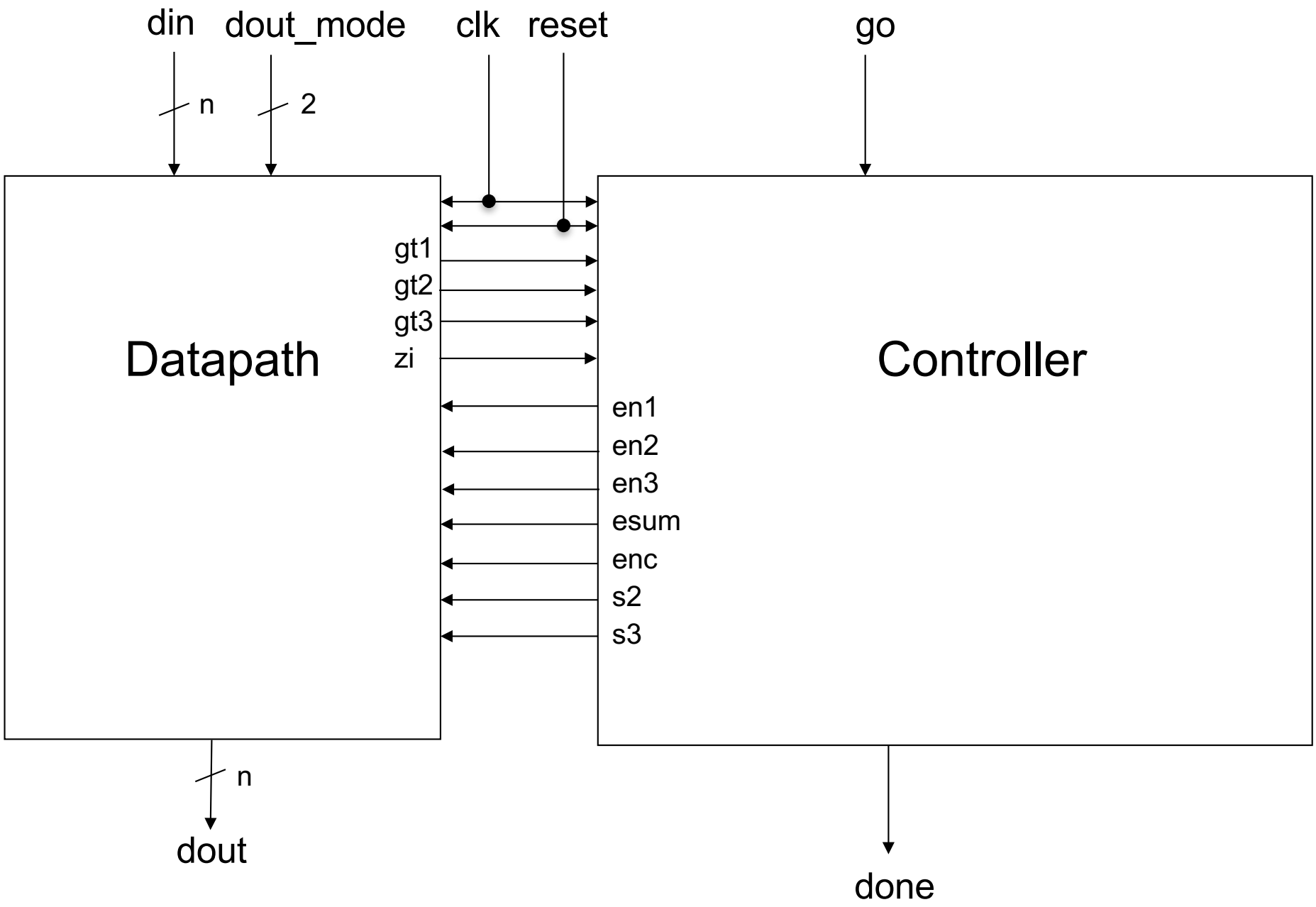
Interface Table

Port	Width	Meaning
clk	1	System clock.
reset	1	System reset.
din	n	Input Data.
go	1	Control signal indicating that the first input is ready. Active for one clock cycle.
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.

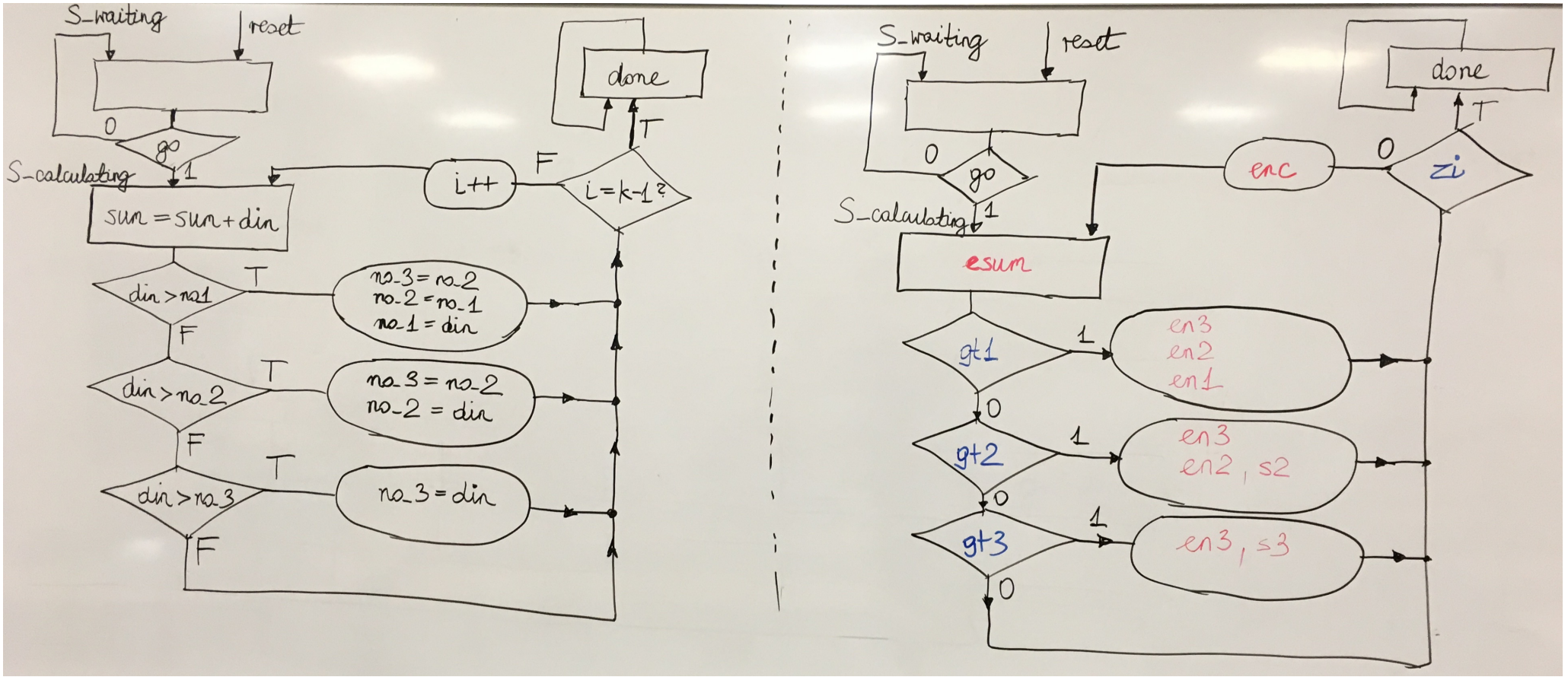


Block diagram of the Datapath

Interface with the division into the Datapath and Controller



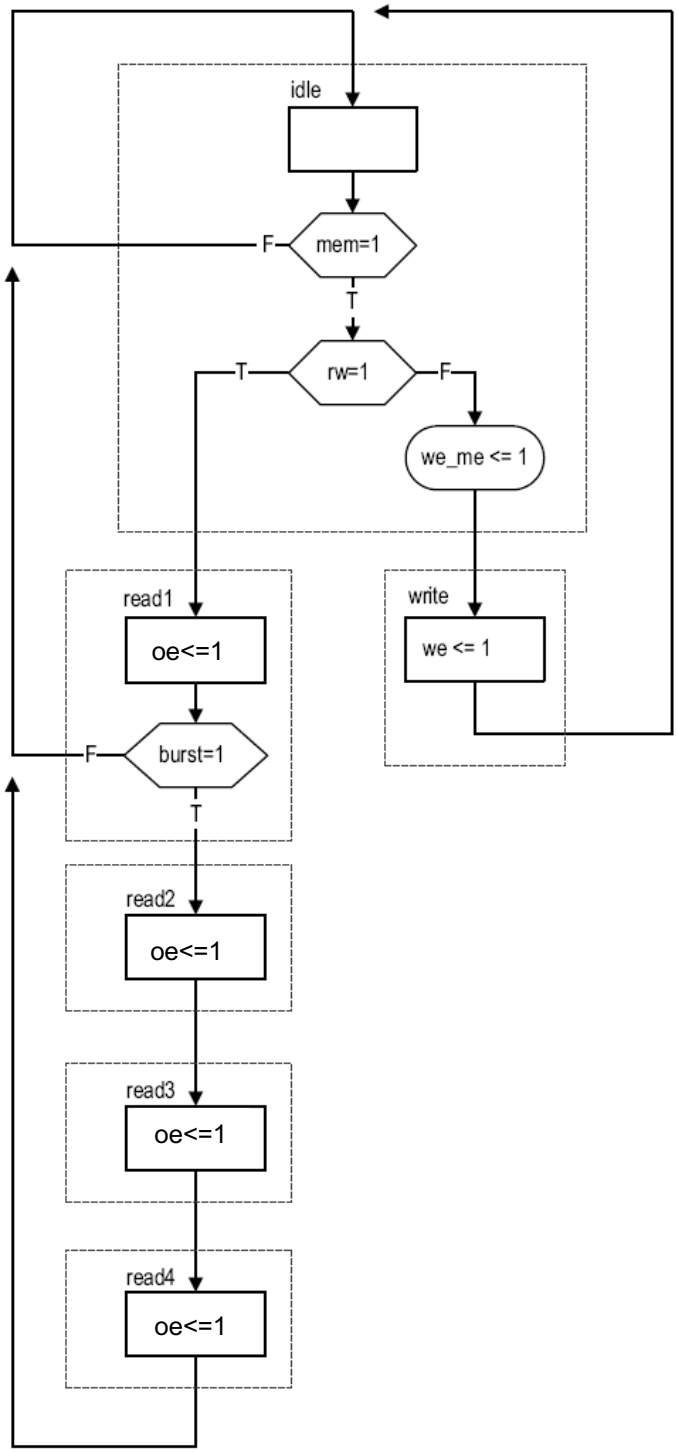
ASM Charts


S_waiting
reset
0
go
1
S_calculating
sum = sum + din
din > no1
T
F
no_3 = no_2
no_2 = no_1
no_1 = din
din > no_2
T
F
no_3 = no_2
no_2 = din
din > no_3
T
F
no_3 = din
i++
F
i = k-1 ?
T
done
S_waiting
reset
0
go
1
S_calculating
esum
gt1
1
0
gt2
1
0
gt3
1
0
en3
en2
en1
en3
en2 , s2
en3 , s3
enc
0
zi
T
done

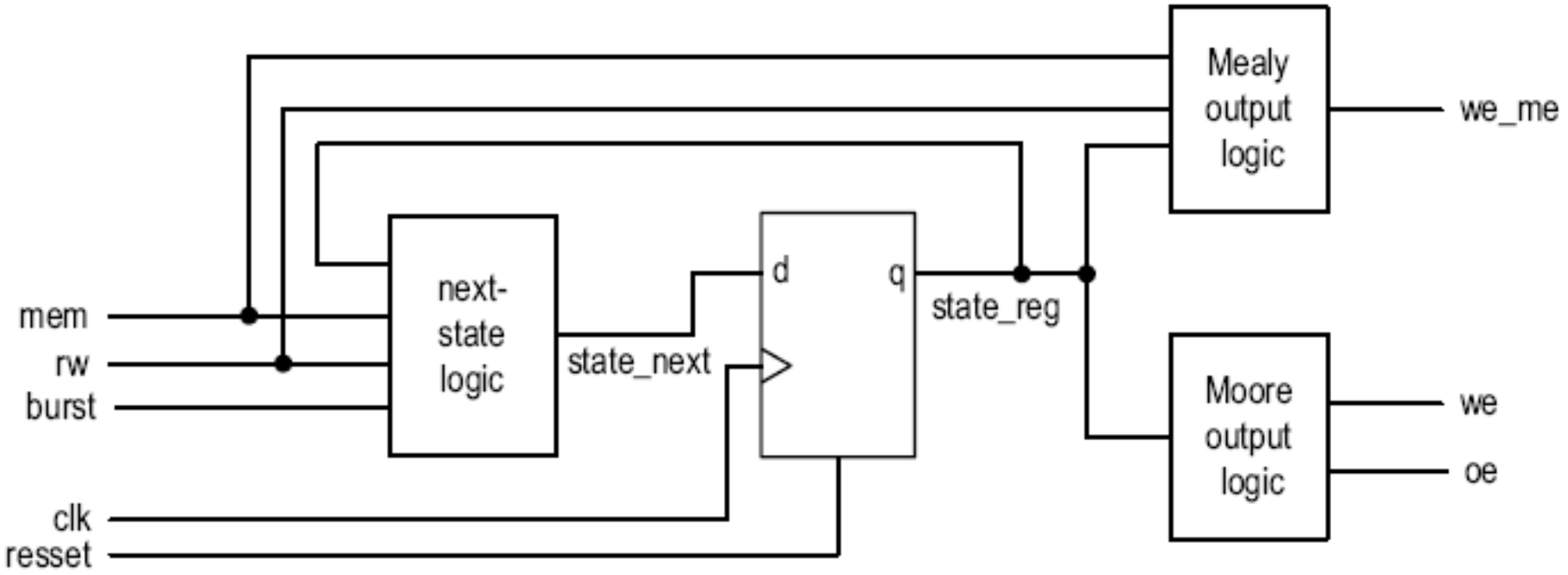


Alternative VHDL Coding Styles

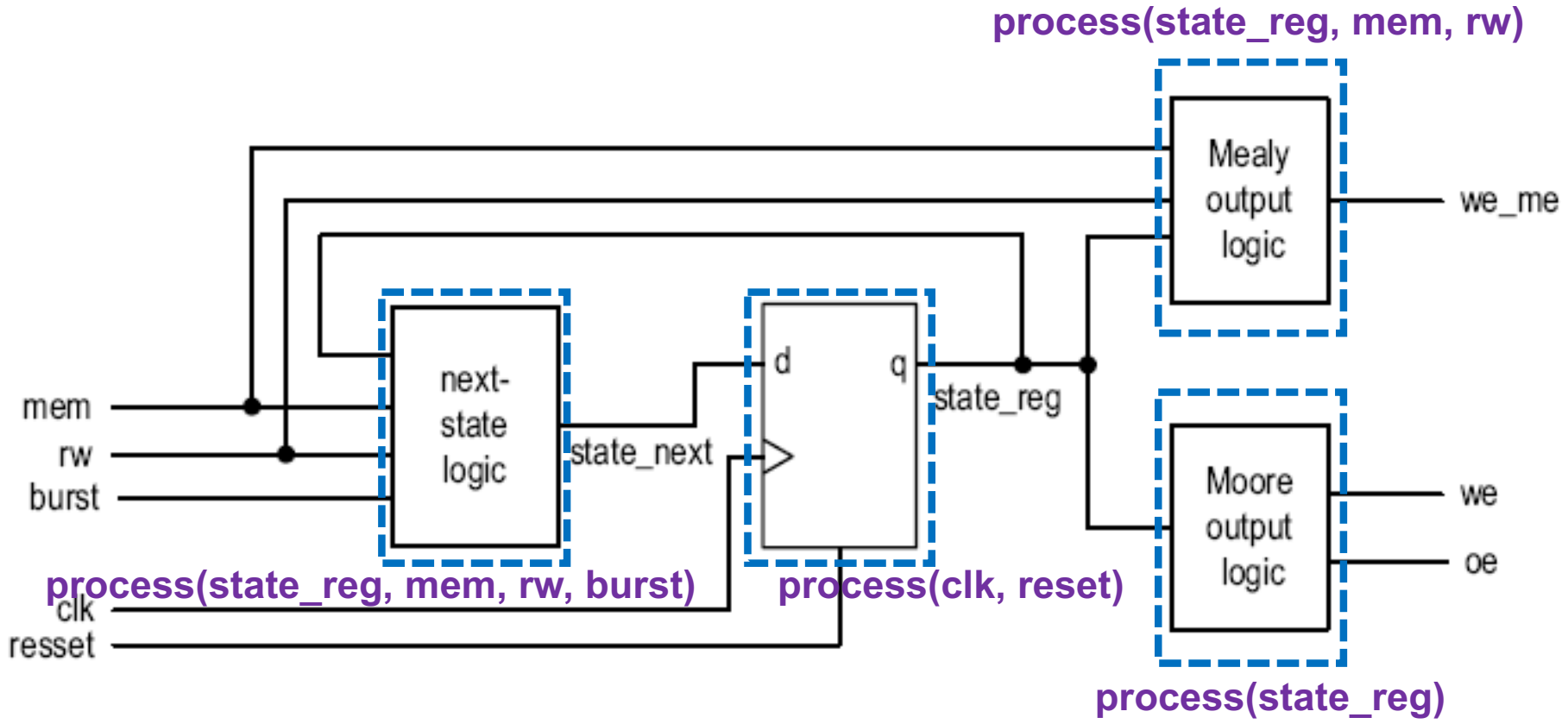
Memory Controller



Coding Style 1



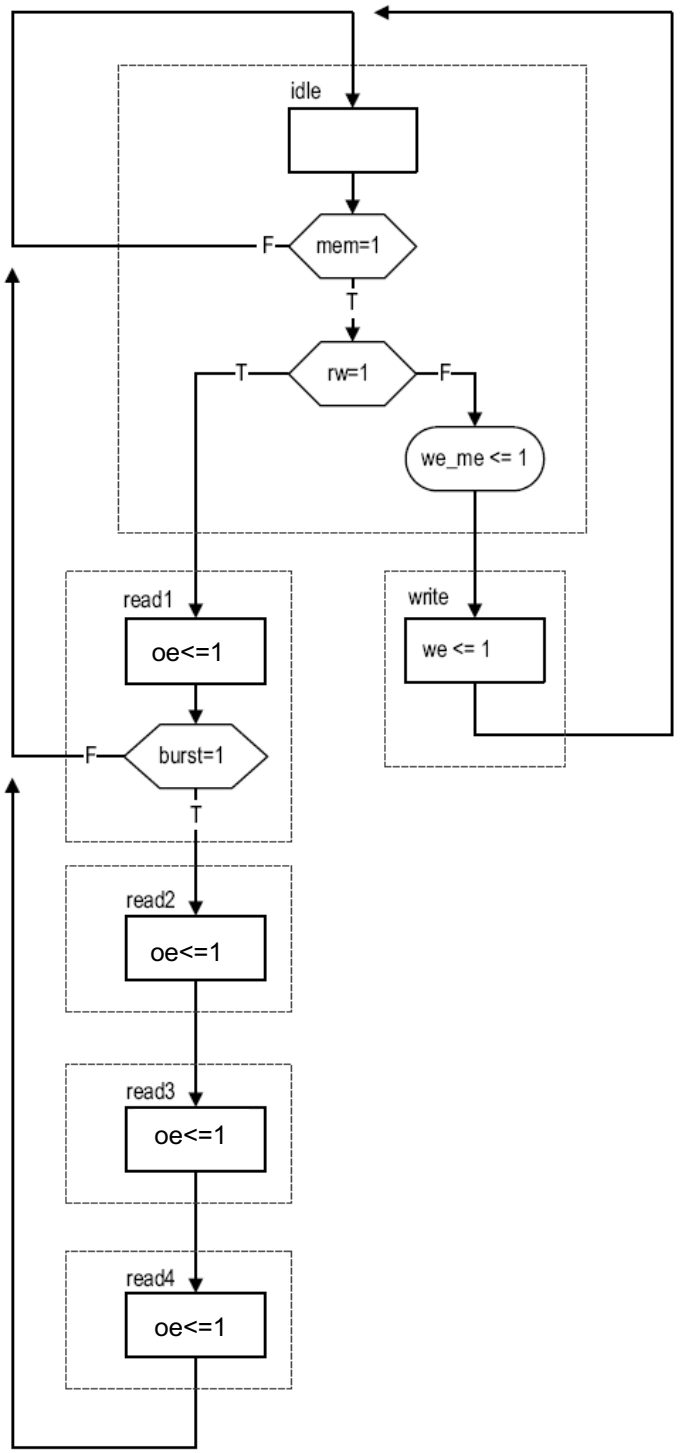
Coding Style 1: Choice of Description Styles



```
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;
```

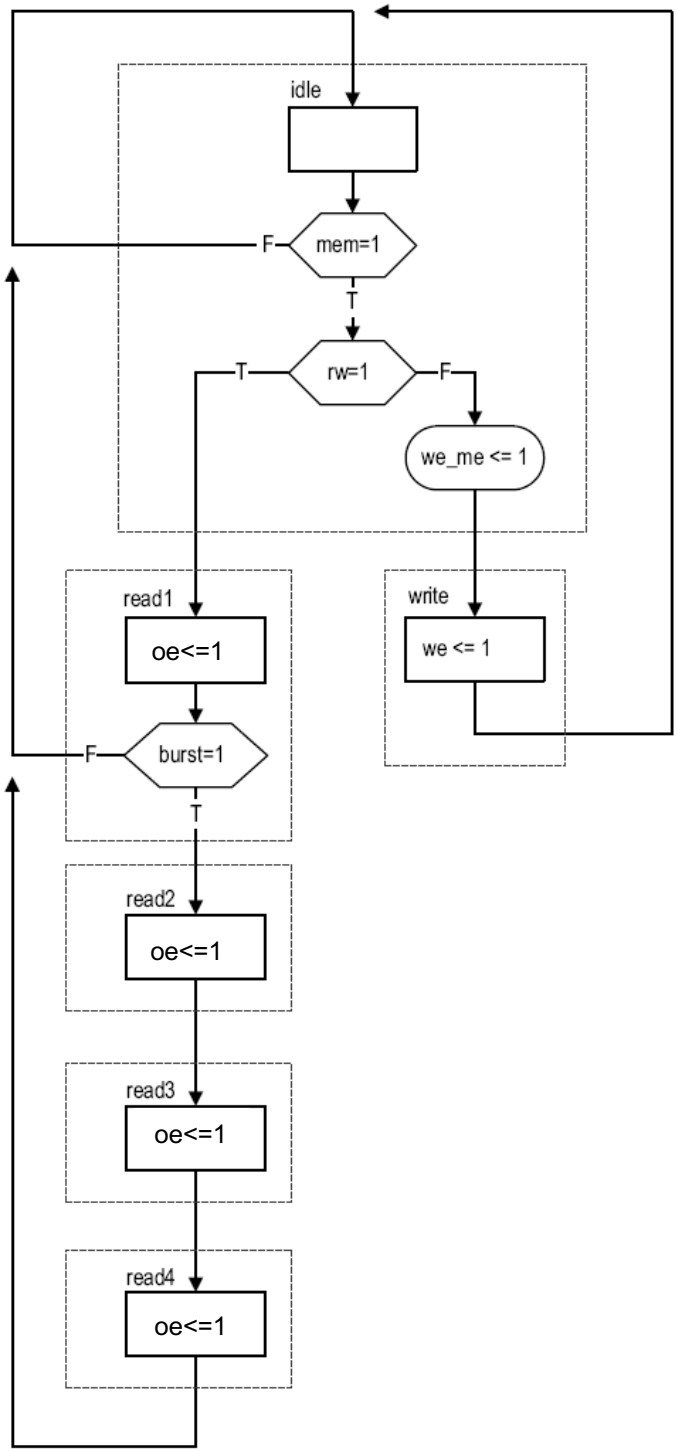
Memory Controller



```
-- 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;
```

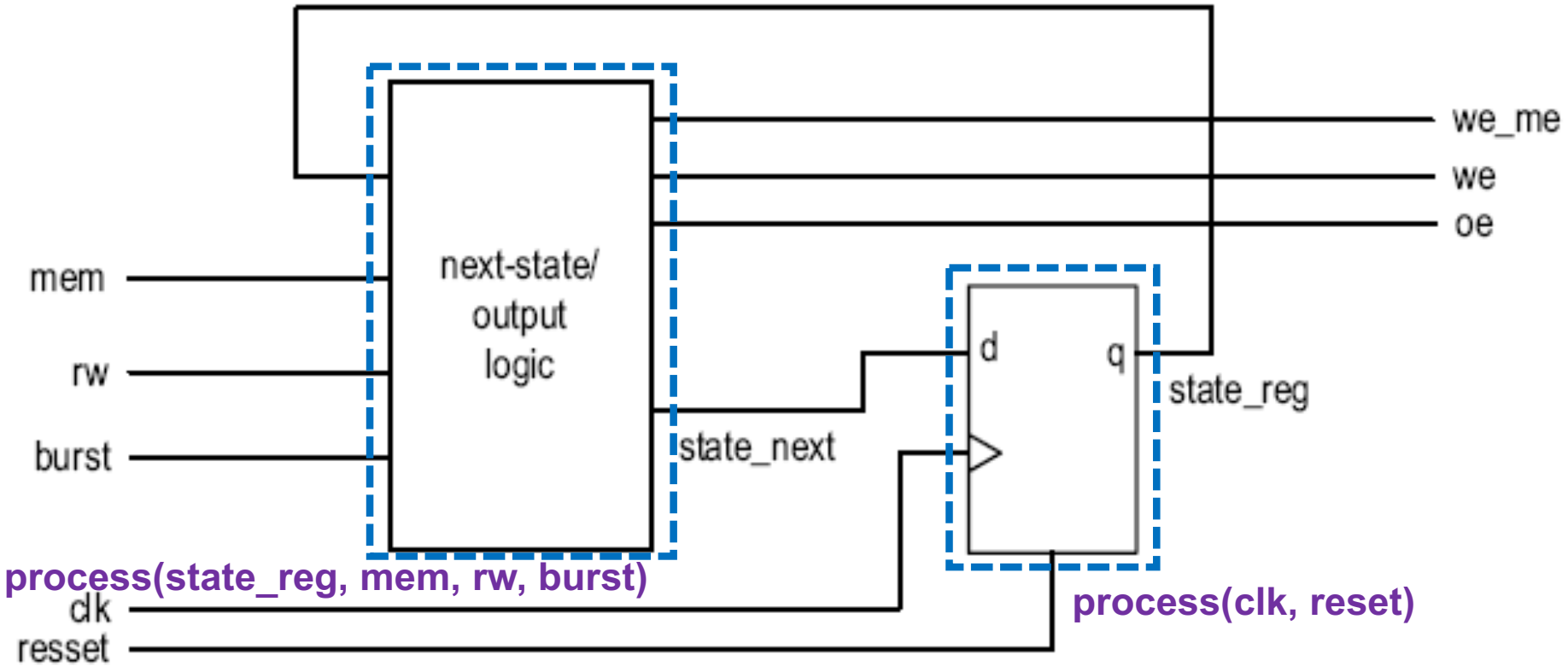
Memory Controller



```
-- 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;
```

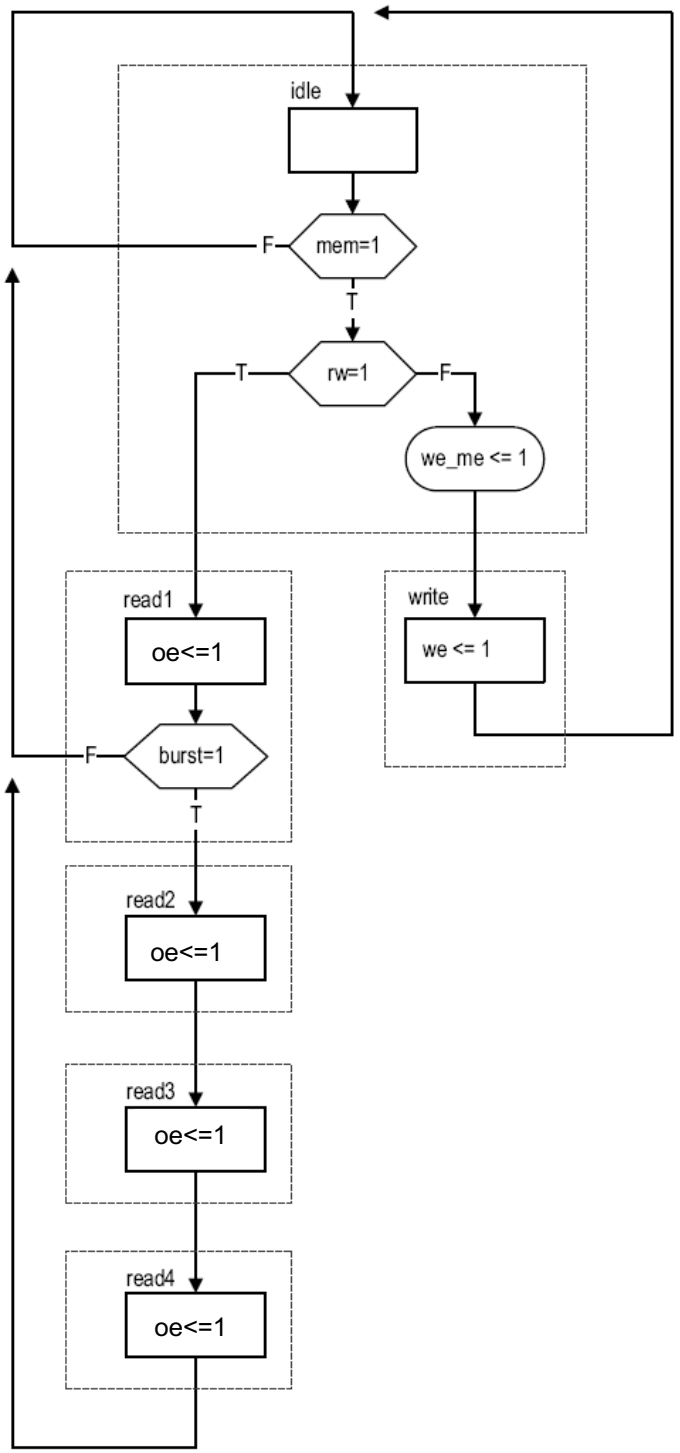
Coding Style 2 - Recommended



```
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 ;

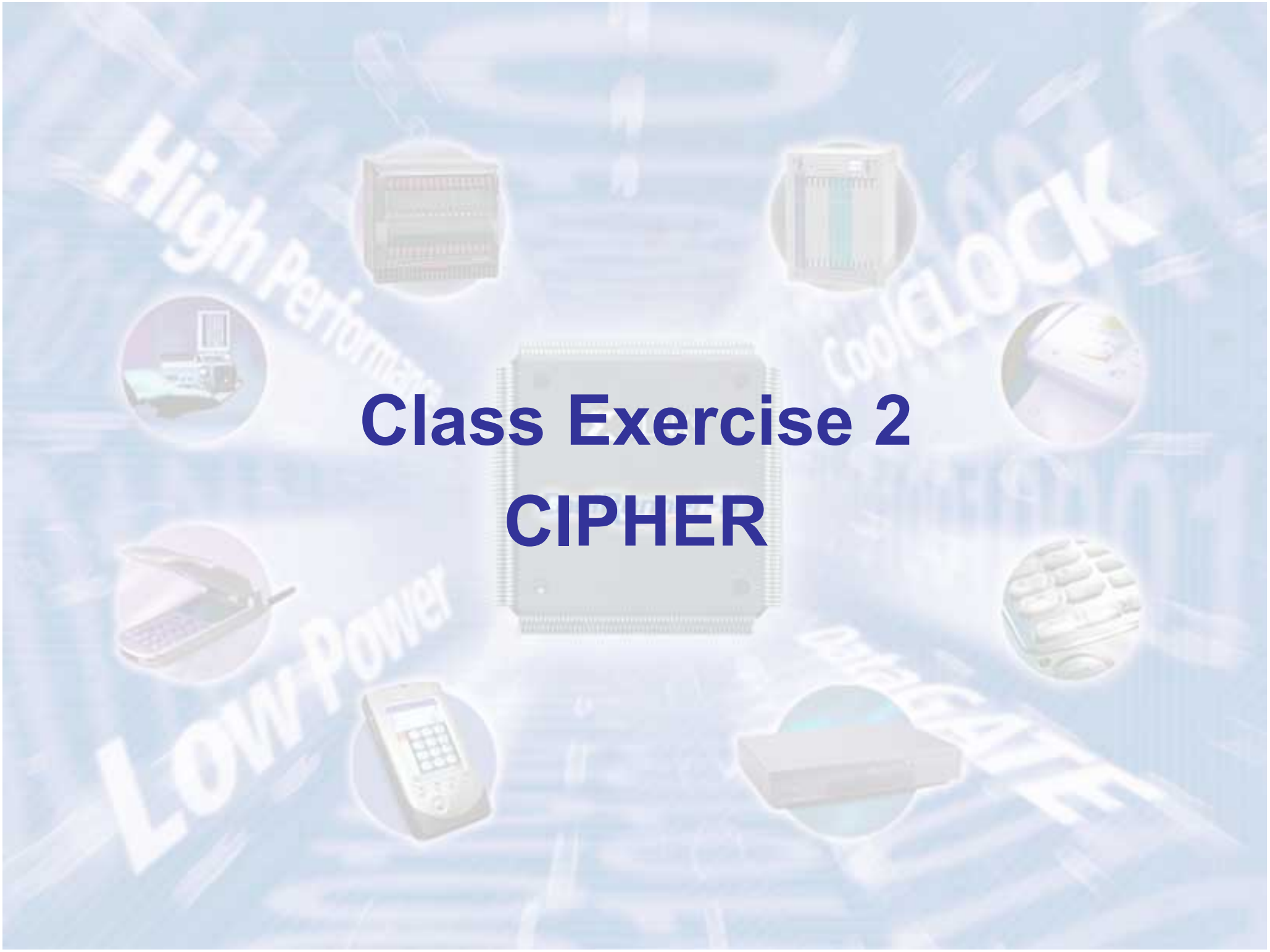
architecture mult_seg_arch of mem_ctrl is
   type mc_state_type is
      (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
begin
   -- state register
   process(clk, reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
```

Memory Controller



```
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;
```



Class Exercise 2
CIPHER

Pseudocode

Split input I into four words, I3, I2, I1, I0, of the size of w bits each

```
A = I3; B = I2; C = I1; D=I0
B = B + S[0]
D = D + S[1]
      for i = 1 to r do
      {
            T = (B*(2B + 1)) <<< k
            U = (D*(2D + 1)) <<< k
            A = ((A ⊕ T) <<< U) + S[2i]
            C = ((C ⊕ U) <<< T) + S[2i + 1]
            (A, B, C, D)  =  (B, C, D, A)
      }
A = A + S[2r + 2]
C = C + S[2r + 3]
O = (A, B, C, D)
```

Notation

w: word size, e.g., w=32 (constant)

k: $\log_2(w)$ (constant)

A, B, C, D, U, T: w-bit variables

I3, I2, I1, I0: Four w-bit words of the input I

r: number of rounds (constant)

O: output of the size of 4w bits

S[j] : 2r+4 round keys stored in two RAMs.

Each key is a w-bit word.

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

Operations

⊕ : XOR

+ : addition modulo 2^w

– : subtraction modulo 2^w

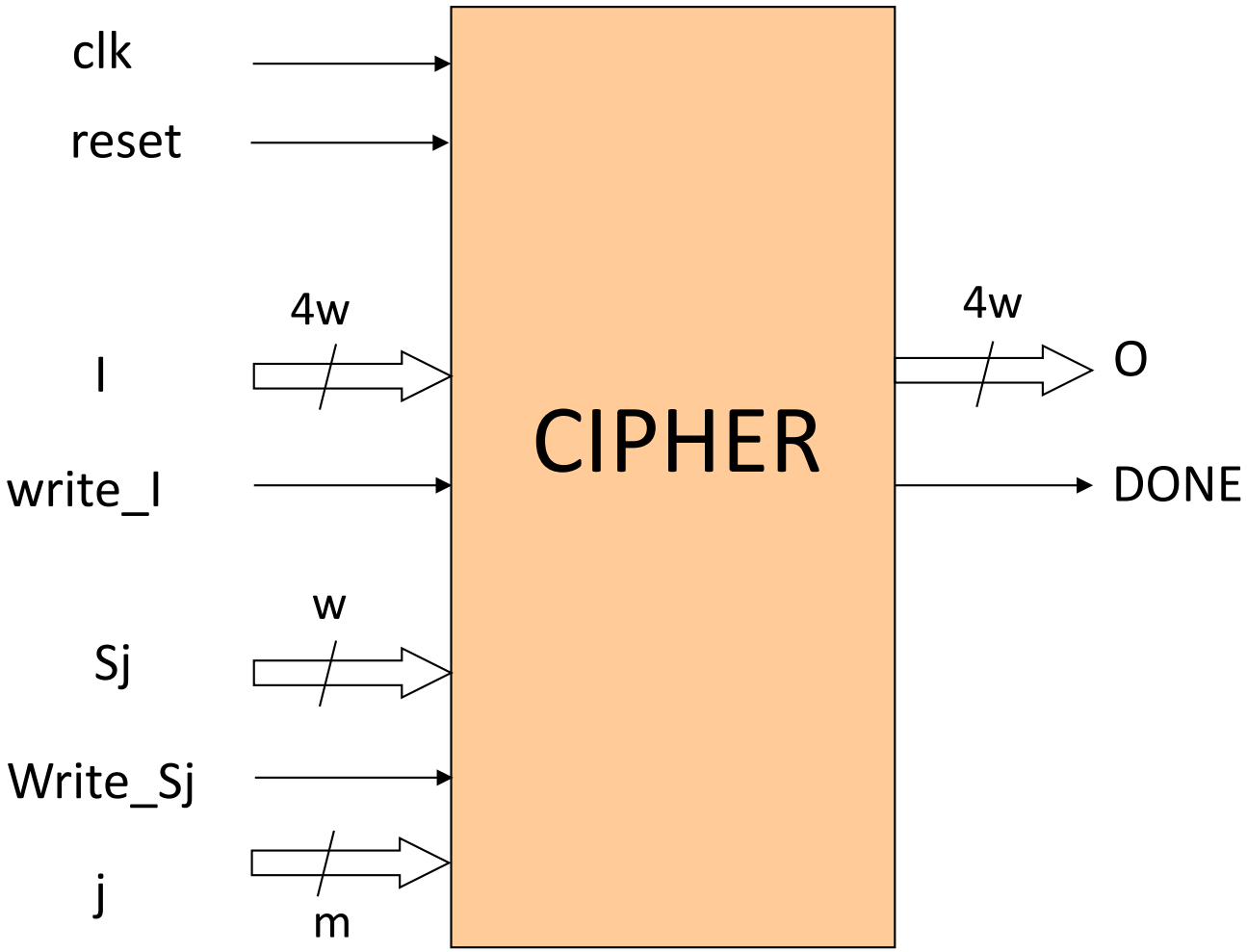
*** :** multiplication modulo 2^w

X <<< Y : rotation of X to the left by the number of positions given in Y

X >>> Y : rotation of X to the right by the number of positions given in Y

(A, B, C, D) : Concatenation of A, B, C, and D.

Circuit Interface



Interface Table

Port	Width	Meaning
clk	1	System clock.
reset	1	System reset – clears internal registers.
I	4w	Input block.
write_I	1	Synchronous write control signal for the input block I. After the block I is written to the CIPHER unit, the encryption of I starts automatically.
Sj	w	Round key S[j] loaded to one of the two internal memories.
write_Sj	1	Synchronous write control signal for the round key S[j].
j	m	Index of the round key S[j] loaded using input Sj.
O	4w	Output block.
DONE	1	Asserted for one clock cycle when the output O is ready.

Note:
m is a size of index j. It is a minimum integer, such that $2^m - 1 \geq 2r+3$.

Protocol (1)

An external circuit first loads all round keys
S[0], S[1], S[2], …, S[2r+2], [2r+3]
to the two internal memories of the CIPHER unit.

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

Loading round keys is performed using inputs: Sj, j, write_Sj, clk.

Then, the external circuits, loads an input block I to the CIPHER unit, using inputs: I, write_I, clk.

After the input block I is loaded to the CIPHER unit, the encryption starts automatically.

Protocol (2)

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

The output O 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.

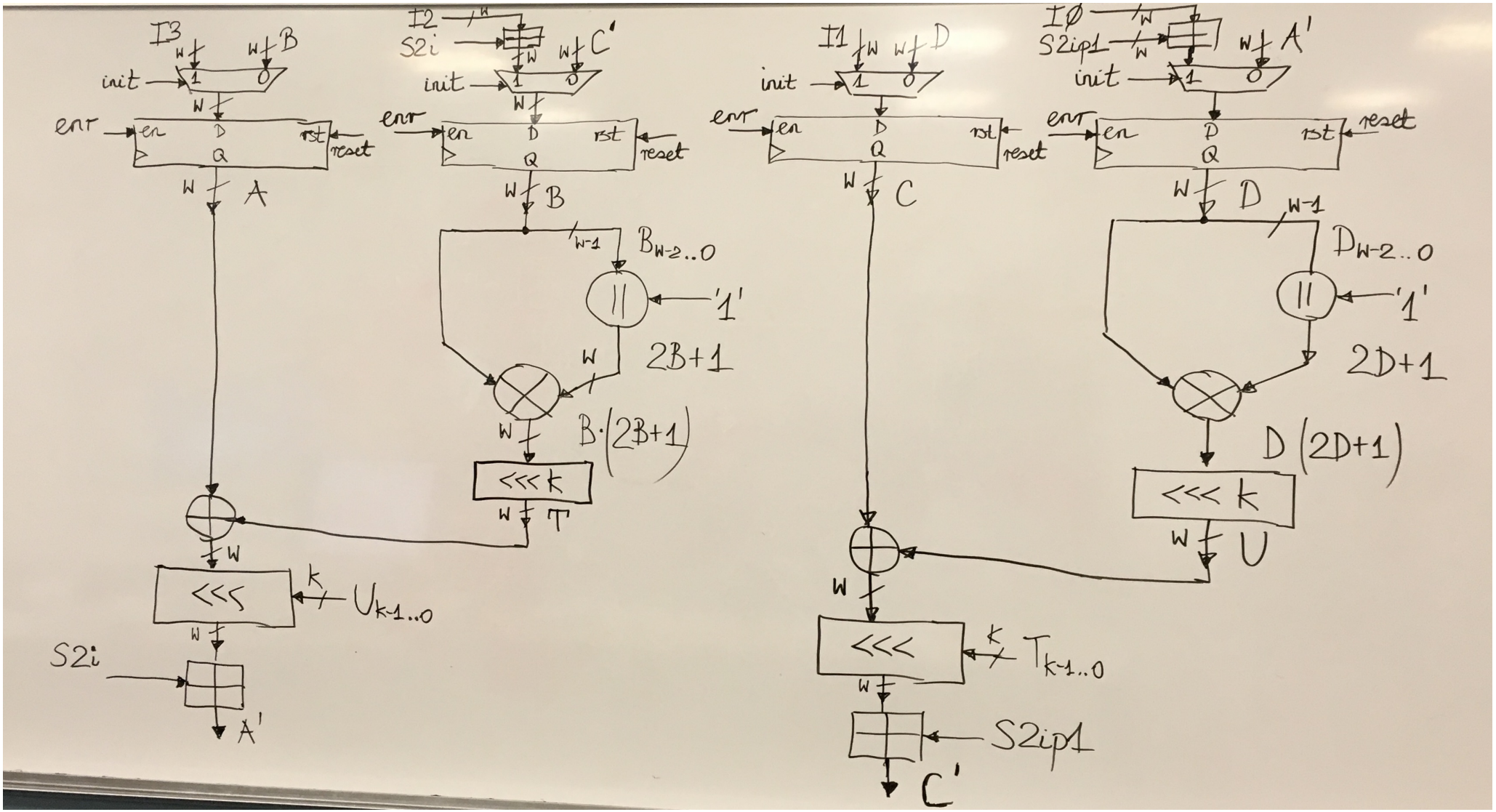
Assumptions

- 2r+4 clock cycles are used to load round keys to internal RAMs
- 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

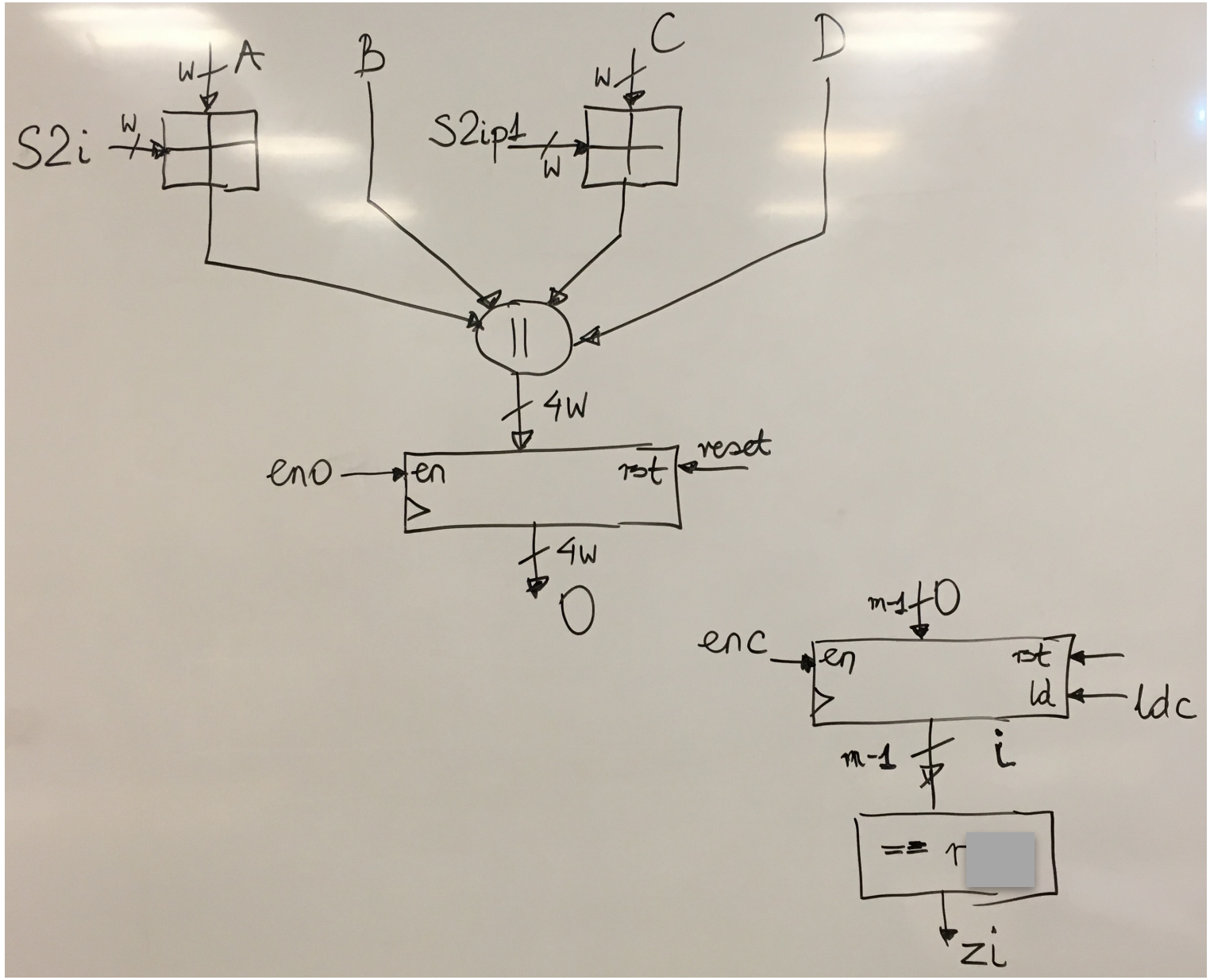
As a result, the encryption of a single input block I

should last r+2 clock cycles.

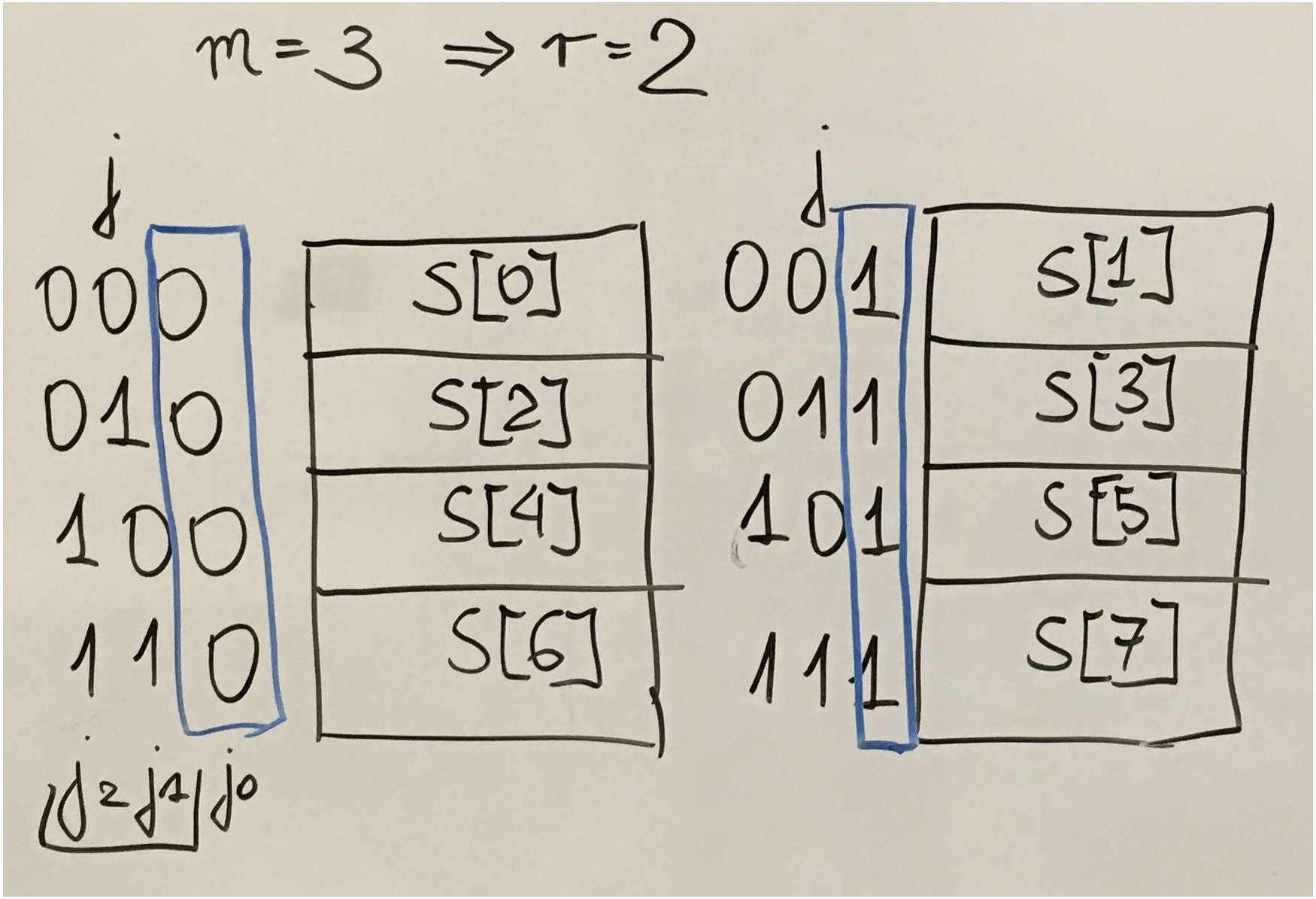
Datapath – Initialization & Main Loop



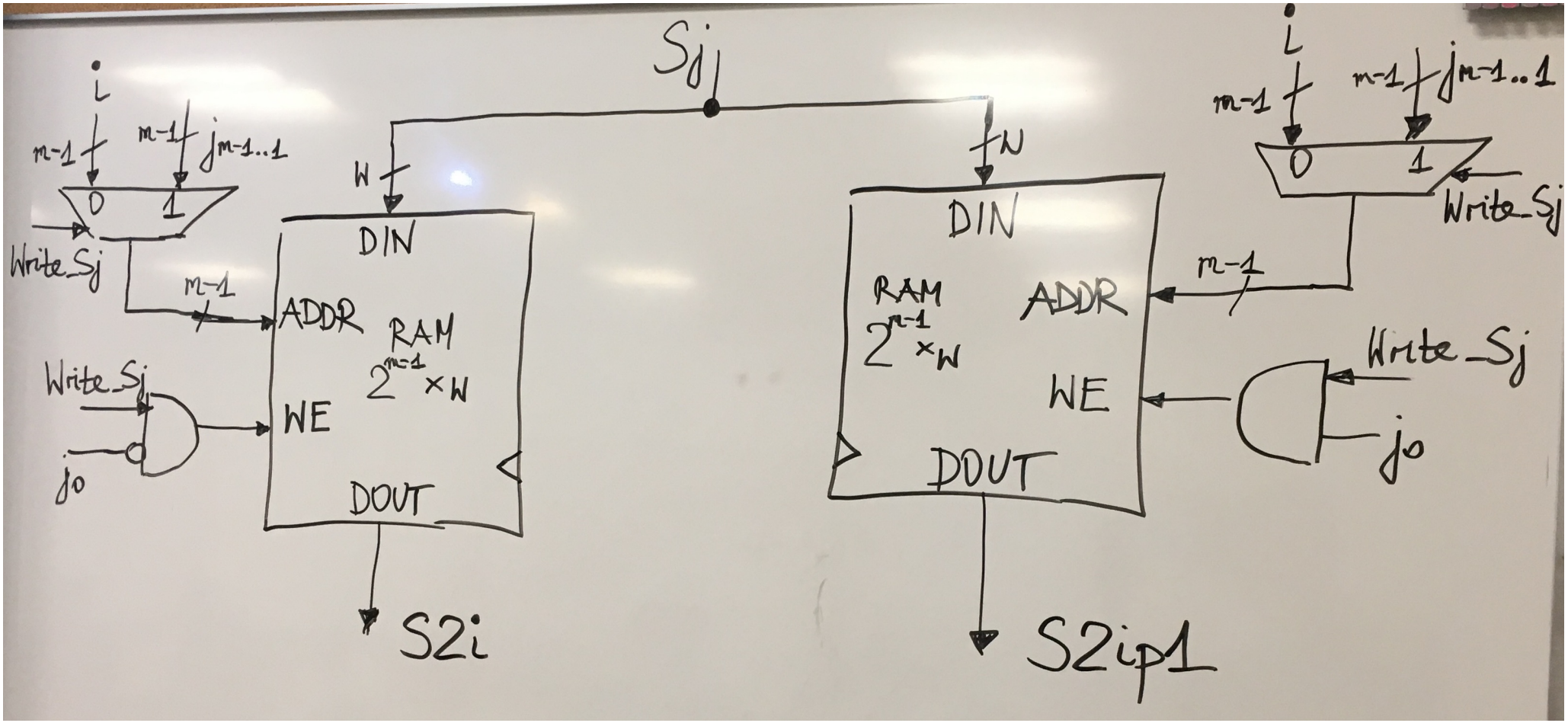
Datapath – PostProcessing & Loop Counter



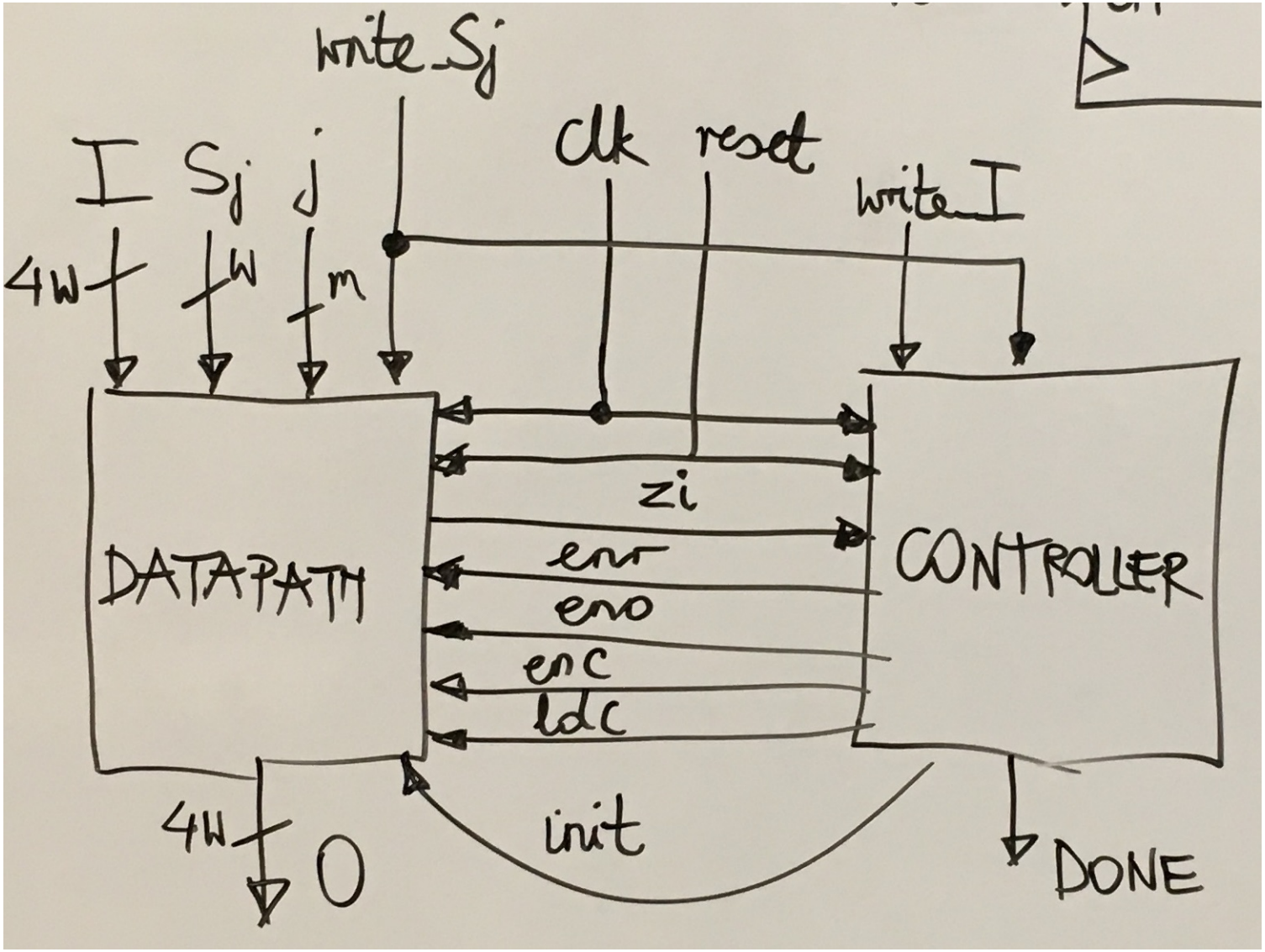
Memories of Round Keys – Contents
Case of m=3 and r=2



Datapath – Memories of Round Keys



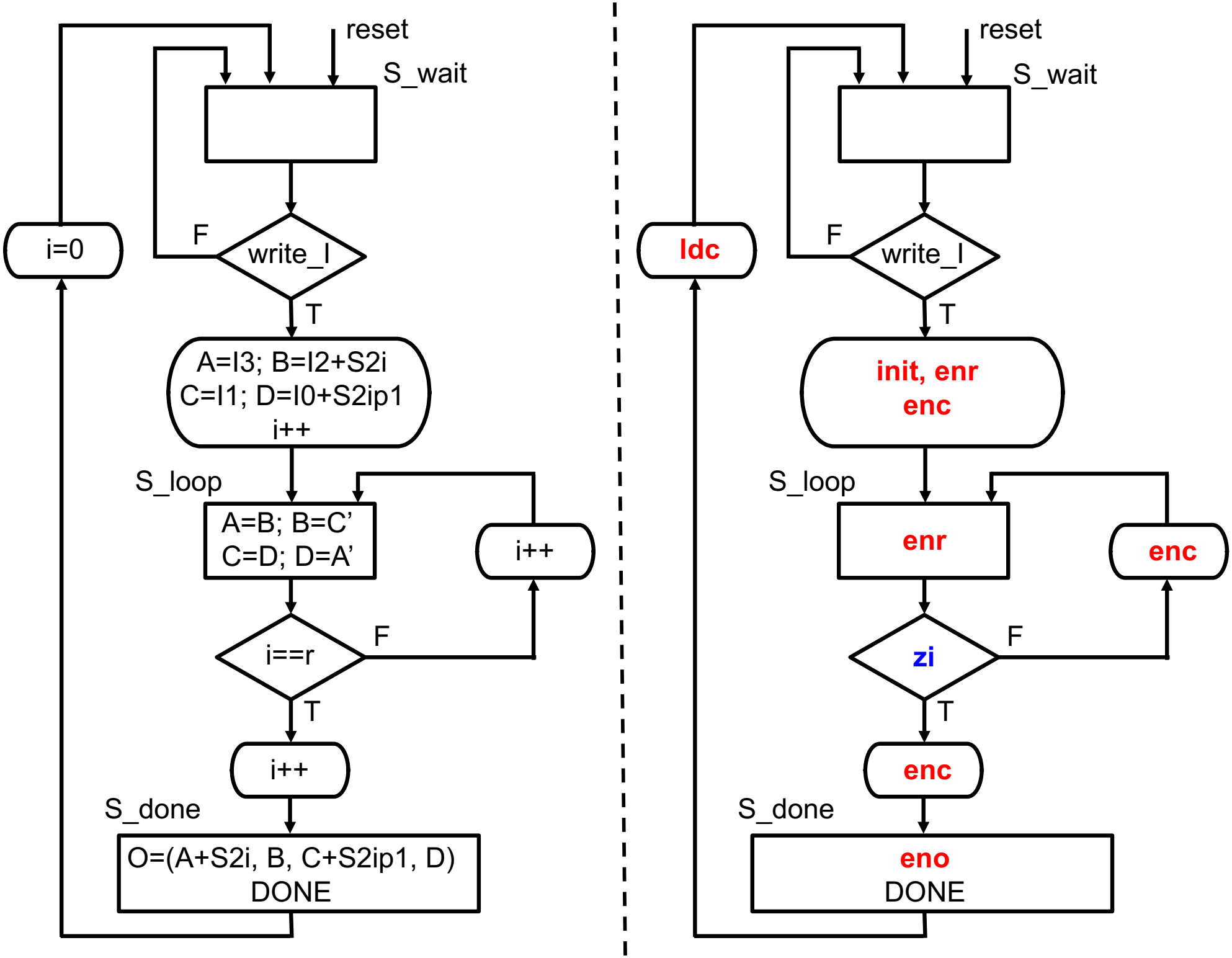
Interface with the Division into the Datapath and Controller



Pseudocode

Split input I into four words, I3, I2, I1, I0, of the size of w bits each

```
A = I3; B = I2; C = I1; D=I0
B = B + S[0]
D = D + S[1]
      for i = 1 to r do
      {
            T = (B*(2B + 1)) <<< k
            U = (D*(2D + 1)) <<< k
            A = ((A ⊕ T) <<< U) + S[2i]
            C = ((C ⊕ U) <<< T) + S[2i + 1]
            (A, B, C, D)  =  (B, C, D, A)
      }
A = A + S[2r + 2]
C = C + S[2r + 3]
O = (A, B, C, D)
```


reset
S_wait
F
write_I
T
i=0
A=I3; B=I2+S2i
C=I1; D=I0+S2ip1
i++
S_loop
A=B; B=C'
C=D; D=A'
i++
i==r
F
T
i++
S_done
O=(A+S2i, B, C+S2ip1, D)
DONE
reset
S_wait
F
write_I
T
ldc
init, enr
enc
S_loop
enr
enc
zi
F
T
enc
S_done
eno
DONE