COMP3222/9222 Digital Circuits & Systems

6. Synchronous Sequential Circuits

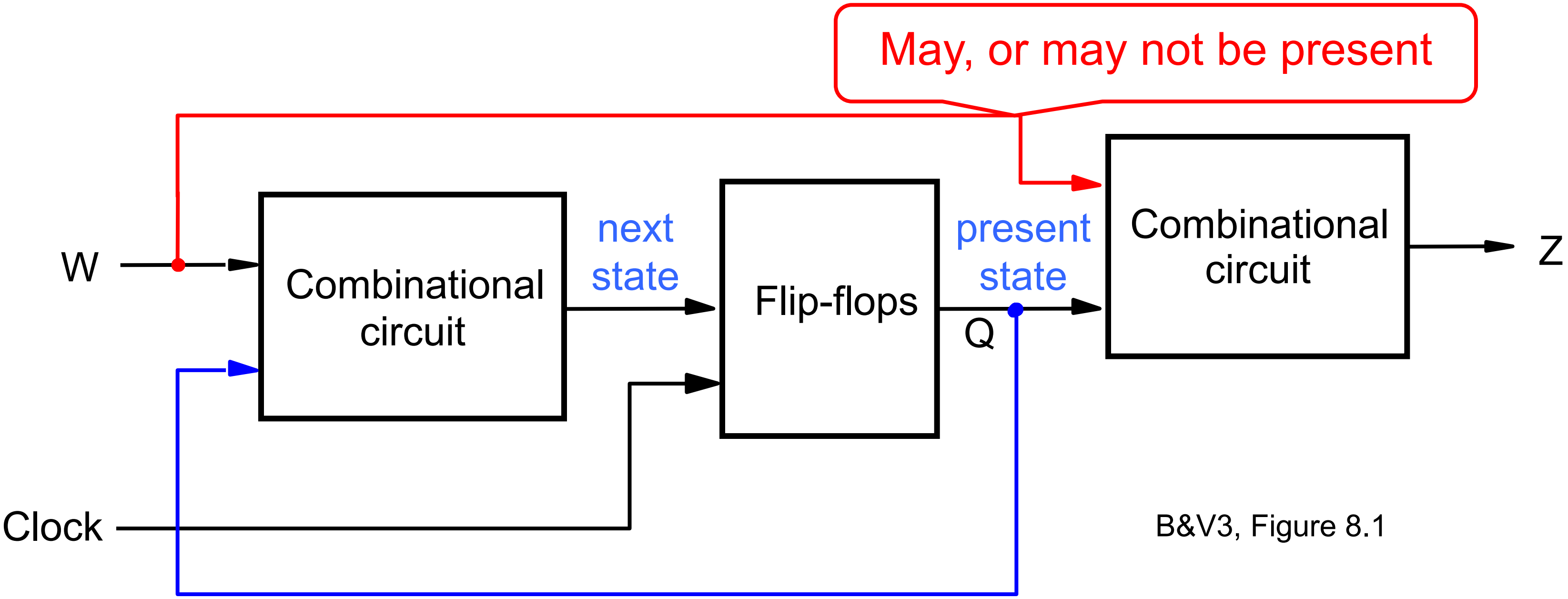
Objectives

- Learn design techniques for circuits that use flip-flops
- Understand the concept of states and their implementation with flip-flops
- Learn about the synchronous control of circuits using a clock signal
- Learn how to design synchronous sequential circuits
- Learn how to specify synchronous sequential circuits using VHDL
- Understand the techniques CAD tools use to synthesize synchronous sequential circuits

Synchronous sequential circuits

- So far…
  - Looked at *combinational circuits*, whose outputs are completely determined by their inputs, and
  - *Flip-flops*, whose outputs depend upon their current state
- Now:
  - Consider a general class of circuits, known as ***sequential circuits***, whose outputs depend upon *past inputs and state*, as well as *present input values*
  - A clock signal is commonly used to control the operation of a sequential circuit; these circuits are therefore known as ***synchronous*** sequential circuits; we won't consider the design of asynchronous sequential circuits in this course
  - Synchronous sequential circuits are designed using combinational logic together with one or more flip-flops

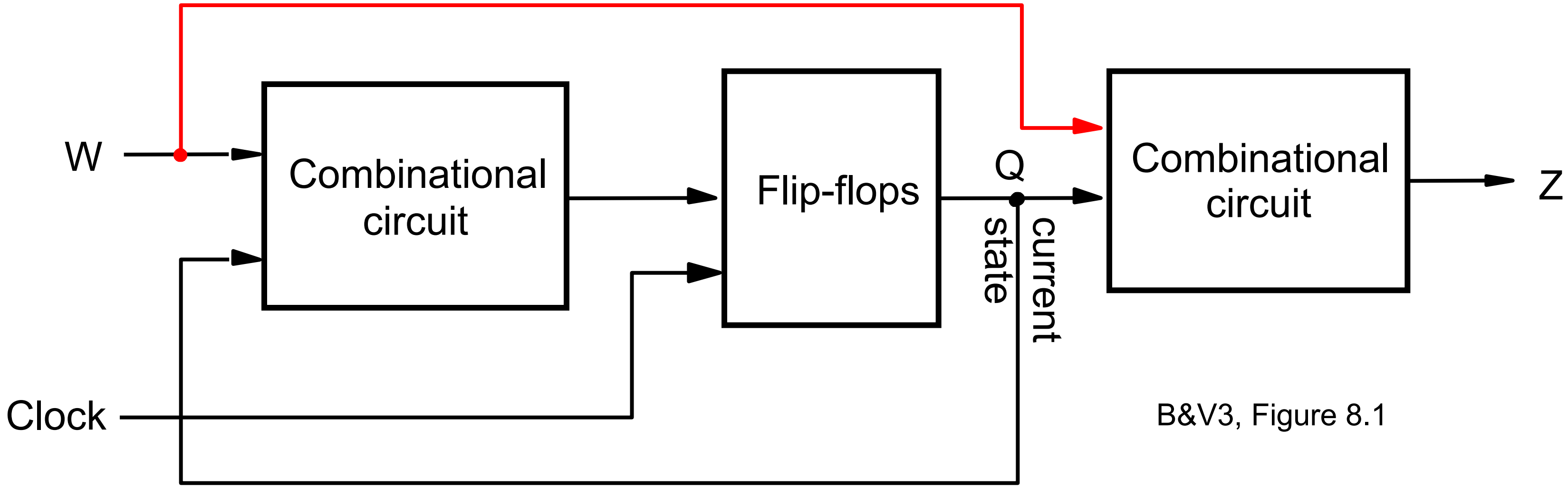
General form of a sequential circuit



B&V3, Figure 8.1

- Circuit has primary inputs W and primary outputs Z
- The outputs of the FFs are referred to as the *state*, Q, of the circuit.
  - In order to simplify analysis, the state should only change once per clock cycle; FFs should therefore be edge-triggered
  - Changes in state depend upon both the current inputs and the present (current) outputs of the FFs, Q
- Outputs of the circuit depend upon the current state, and *may* also depend upon the current inputs, though this is not required

General form of a sequential circuit



B&V3, Figure 8.1

- When the outputs Z only depend upon the current state Q, the circuit is said to be of ***Moore*** type
- Alternatively, when the outputs Z depend upon the current state, Q, and the inputs W, the circuit is said to be of ***Mealy*** type
- Mealy circuits may require less states for similar behaviour and are more responsive to changes in the inputs
- Because the functional behaviour of the circuit can be represented using a finite number of states, sequential circuits are also called ***finite state machines***

Basic design steps

- Consider the design of a simple circuit meeting the following specifications:
  1. The circuit has one input, *w*, and one output, *z*
  2. All changes in the circuit occur on the positive edge of a clock signal
  3. $z = 1$ if during two immediately preceding clock cycles/periods the input *w* was equal to 1. Otherwise, $z = 0$

- Input/output behaviour of the circuit:

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

B&V3, Figure 8.2

Input/output behaviour of the circuit

- Consider the design of a simple circuit meeting the following specifications:
  1. The circuit has one input, *w*, and one output, *z*
  2. All changes in the circuit occur on the positive edge of a clock signal
  3. *z = 1* if during two immediately preceding clock cycles the input *w* was equal to 1. Otherwise, *z = 0*

- The circuit detects two or more consecutive 1s. Circuits that detect the occurrence of a particular input pattern are referred to as ***sequence detectors***
- Clearly, the output doesn't only depend on the present value of *w*…easily seen if we consider the desired input/output behaviour of the circuit

Input/output behaviour of the circuit

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

B&V3, Figure 8.2

- The different outputs during cycles t_4 and t_8 or t_2 and t_5 illustrate that the output must be determined by some state of the circuit rather than by the current input value

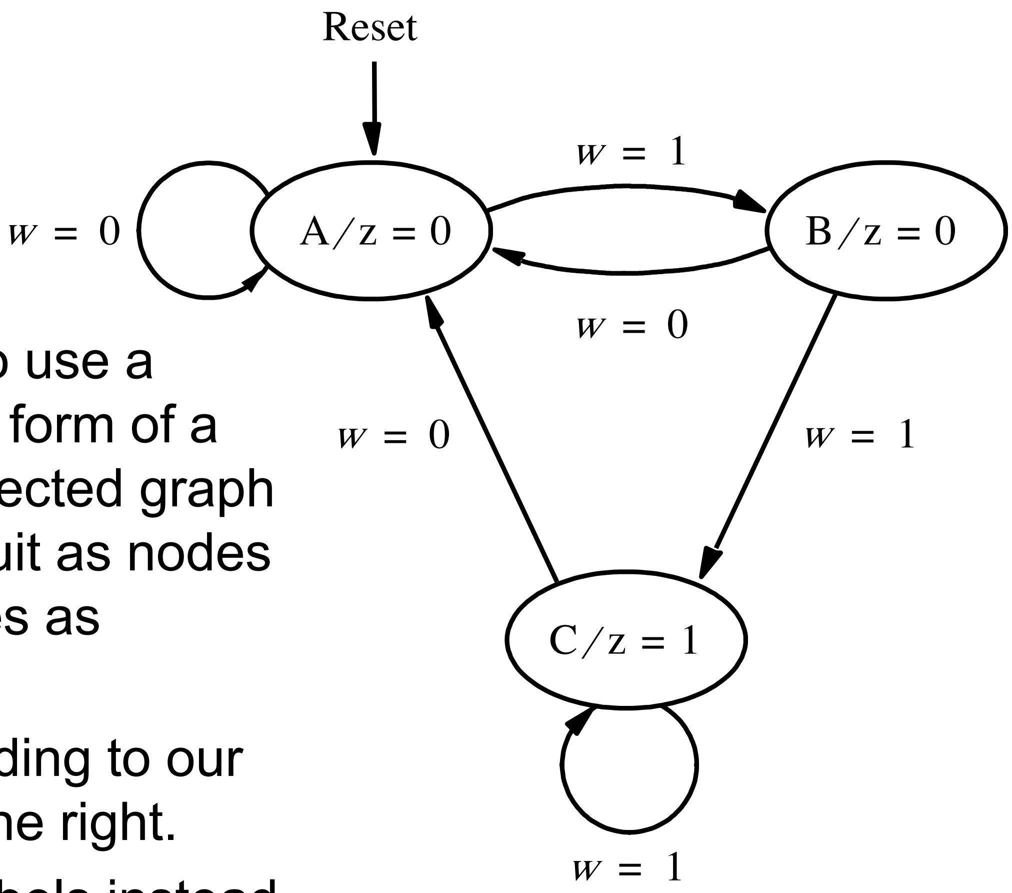
- The first step in designing a finite state machine is to determine how many states are needed and which "transitions" are possible from one state to another

Behaviour of state machine

- There is no set procedure for determining the number of states
- In our example:
  - A Select a starting state that the circuit should enter when first powered on or when a *reset* signal is applied; call it state *A*
  - While *w* = *0*, the circuit need not do anything, and so each active clock edge results in the circuit remaining in state *A*
  - B When *w* = *1*, the machine should recognize this and move to a new state, *B*, say. The transition takes place on the next active clock edge after *w* = *1.*
  - In both states *A* and *B*, the output *z* = *0* as the machine has not yet seen *w* = *1* for 2 consecutive clock cycles.
  - C When in state *B*, if w = 0 at the next active clock edge, the circuit should return to state *A*. However, if *w* = *1* is seen in state *B*, the circuit should change to a third state called *C* and generate an output *z* = *1*.
  - The circuit should remain in state *C* and output *z* = *1* as long as *w* = *1*. When *w* becomes 0, the machine should return to state *A*.
- As all possible values for *w* have been considered in all possible states, we can conclude that 3 states are enough.

State diagram

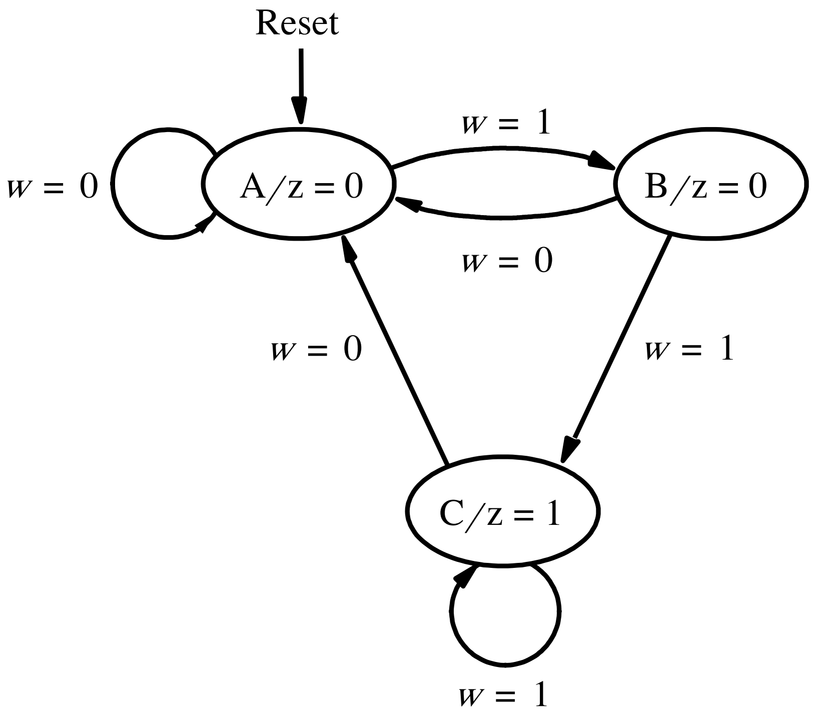
- The behaviour of a sequential circuit can be described in several ways.
- The conceptually easiest is to use a pictorial representation in the form of a ***state diagram***, which is a directed graph that depicts states of the circuit as nodes and transitions between states as directed edges.
- The state diagram corresponding to our specification is as shown to the right.
- It should be noted that any labels instead of letters could be used for the states, and that the transition that is taken is the one associated with the input present prior to the active clock edge arriving.



B&V3, Figure 8.3

State table

- While a state diagram is easy to understand, for implementation it is more convenient to translate the diagram into tabular form
- A ***state table*** indicates all transitions from each *present state* to the *next state* for different input signal values
- For our design, the state table is as shown:
  - Note that here the output is listed with respect to the present state only



State table:

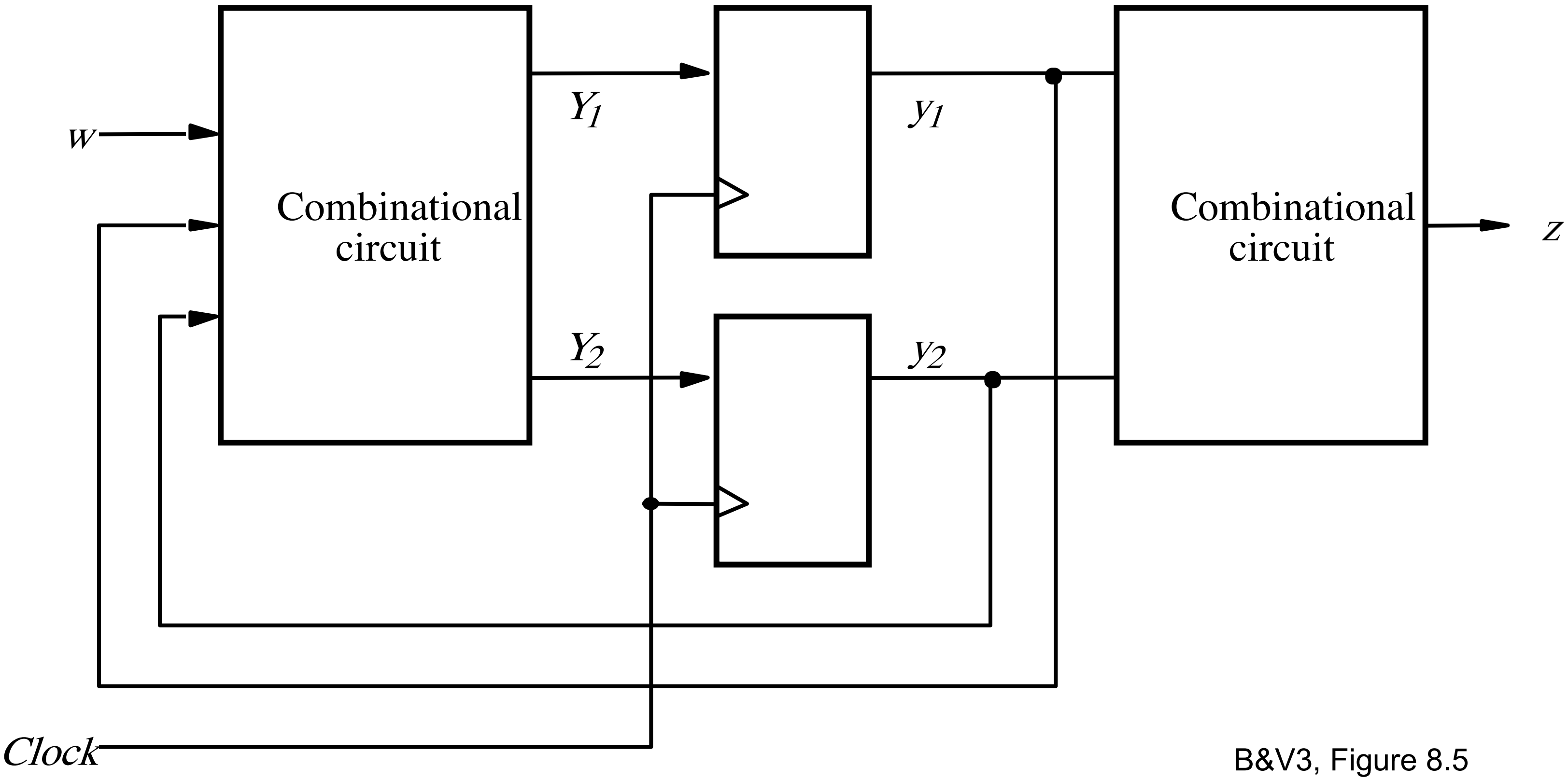
Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

B&V3, Figure 8.4

State assignment

- The state table of the previous slide defines 3 states in terms of letters *A*, *B* and *C*
- When implemented in a logic circuit, each state is represented by a particular valuation of ***state variables***
- Each state variable is implemented in the form of a flip-flop
  - In our example, with three states to represent, at least two state variables are required

Moore sequential circuit with two state flip-flops



B&V3, Figure 8.5

- Upper case Y_1 and Y_2 are called the ***next-state variables***
- Lower case y_1 and y_2 are called the ***present-state variables***
- Next, we need to determine what type of flip-flop to use and design the combinational circuit blocks

State-assigned table

- We therefore need to produce a truth table that defines the function of the combinational circuits.
- More importantly, this requires us to assign a specific valuation of the state variables to each state, resulting in a so-called ***state-assigned table*** for the circuit

State table:

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

State assigned table:

	Present state	Next state		Output
		$w = 0$	$w = 1$	z
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

B&V3, Figure 8.6

Derivation of logic expressions

- Depends on flip-flop type chosen for the implementation
  - D-type is most straightforward

State assigned table:

Present state	Next state		Output
	$w = 0$	$w = 1$	
$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
00	00	01	0
01	00	10	0
10	00	10	1
11	*dd*	*dd*	*d*

Ignoring don't cares

$$Y_1 = w\bar{y}_1\bar{y}_2$$

$$Y_2 = wy_1\bar{y}_2 + w\bar{y}_1 y_2$$

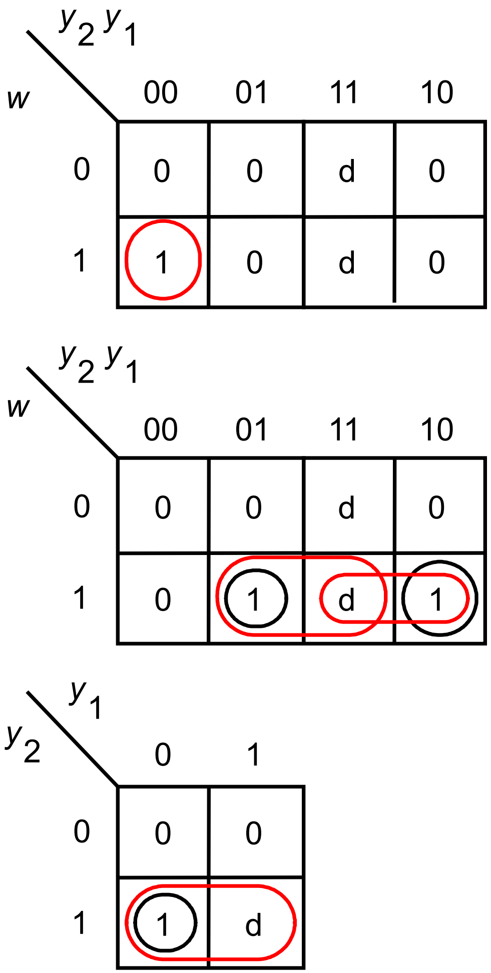
$$z = \bar{y}_1 y_2$$

Using don't cares

$$Y_1 = w\bar{y}_1\bar{y}_2$$

$$Y_2 = wy_1 + wy_2$$
$$= w(y_1 + y_2)$$

$$z = y_2$$



$w \backslash y_2 y_1$	00	01	11	10
0	0	0	d	0
1	1	0	d	0

$w \backslash y_2 y_1$	00	01	11	10
0	0	0	d	0
1	0	1	d	1

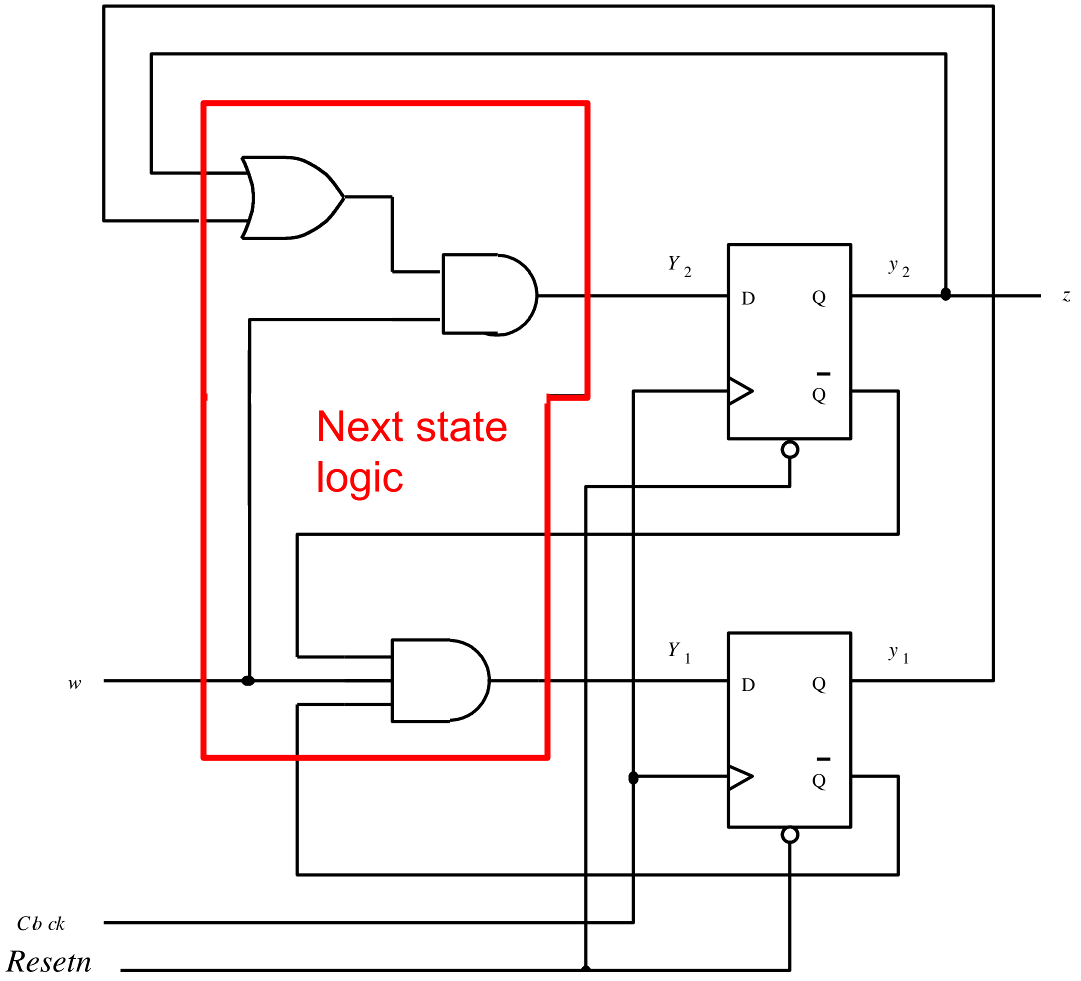
$y_2 \backslash y_1$	0	1
0	0	0
1	1	d

B&V3, Figure 8.7

Final implementation

$$Y_2 = wy_1 + wy_2$$
$$= w(y_1 + y_2)$$

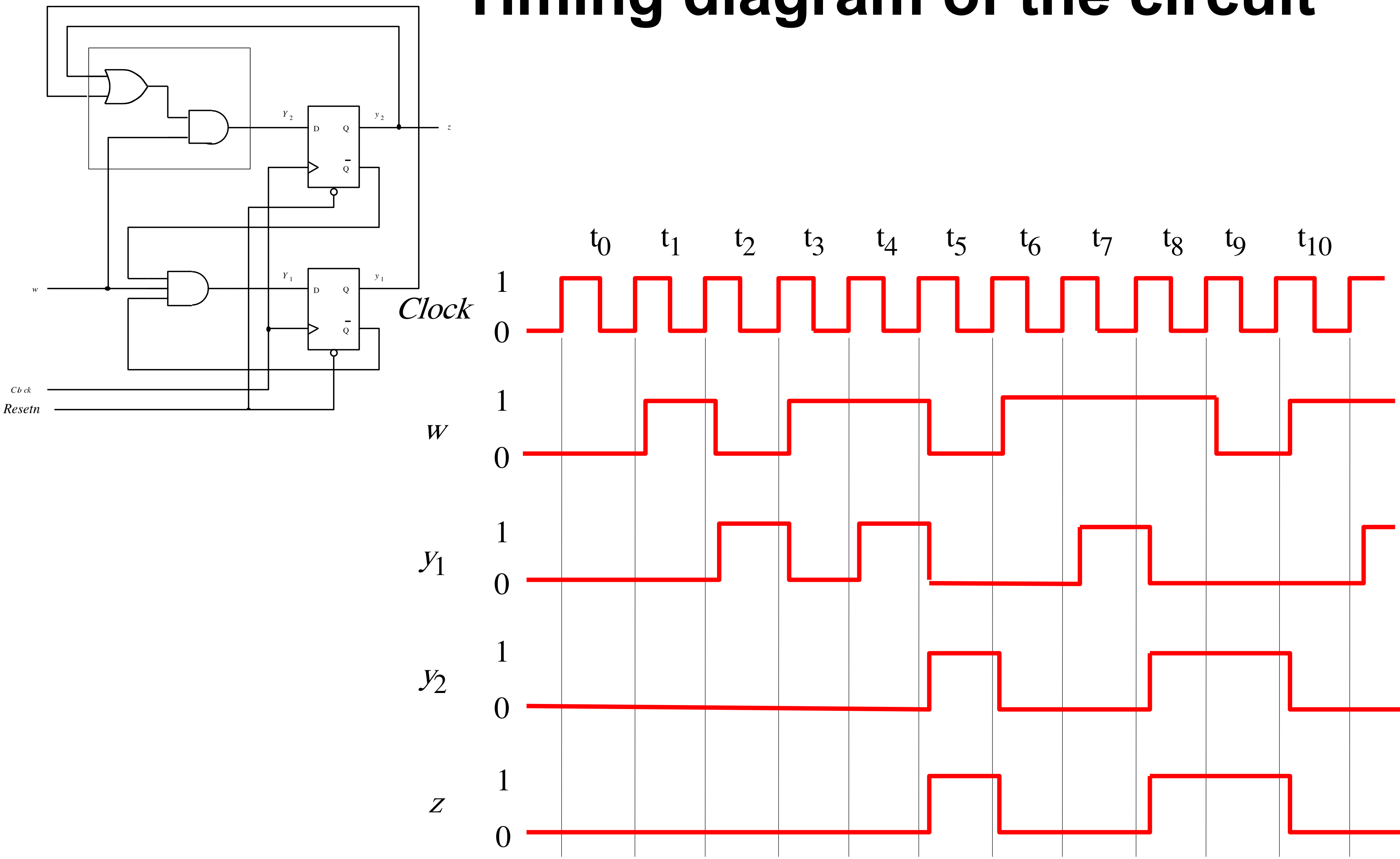
$$Y_1 = w\bar{y}_1\bar{y}_2$$



$$z = y_2$$

B&V3, Figure 8.8

Timing diagram of the circuit



B&V3, Figure 8.9

Summary of design steps

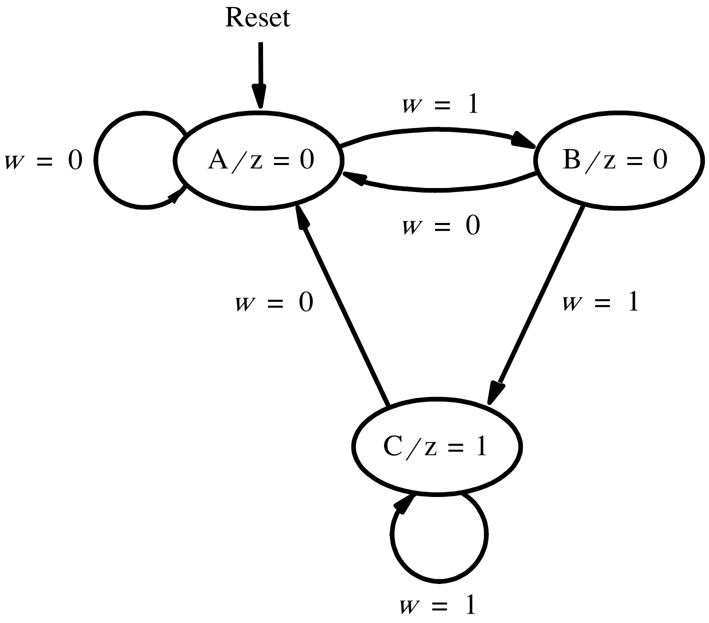
1. Obtain the specification of the desired circuit
2. Derive the states for the machine. **Create a state diagram**. Given a starting state, consider the behaviour in response to all possible inputs and identify new states as required. Repeat for all added states until all possible inputs have been considered for all states. When finished, the state diagram shows all states and the conditions under which the circuit moves from one state to another.
3. **Create a state table** from the state diagram.
4. Determine the number of state variables required to represent all the states and **perform a state assignment**.
5. Given the type of flip-flops to be used, **derive the next-state logic** expressions to control the FF inputs **and** to **produce the desired output**.
6. Implement the circuit.

State-assignment problem

- In the example we’ve just considered, we have seen straightforward implementations following from the state assignment we chose
- Is it possible to obtain better implementations for different assignments?

YES, it is!

Improved state assignment for example 1



	Present state	Next state		Output
		$w = 0$	$w = 1$	
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

B&V3, Figure 8.16

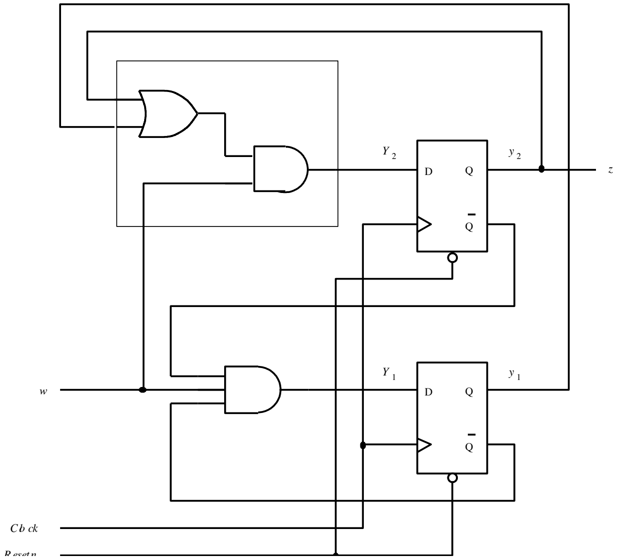
- Choosing *C = 11* rather than *C = 10*, as we previously did, and choosing to implement the circuit using D-type flip-flops results in the next-state and output expressions:

$$Y_1 = D_1 = w$$

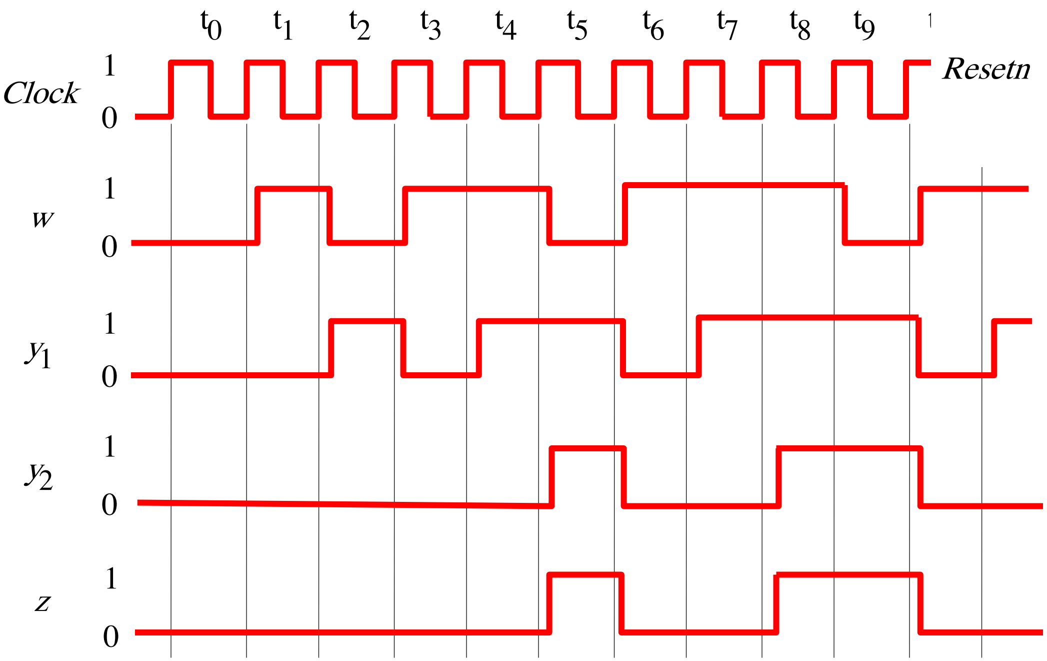
$$Y_2 = D_2 = wy_1$$

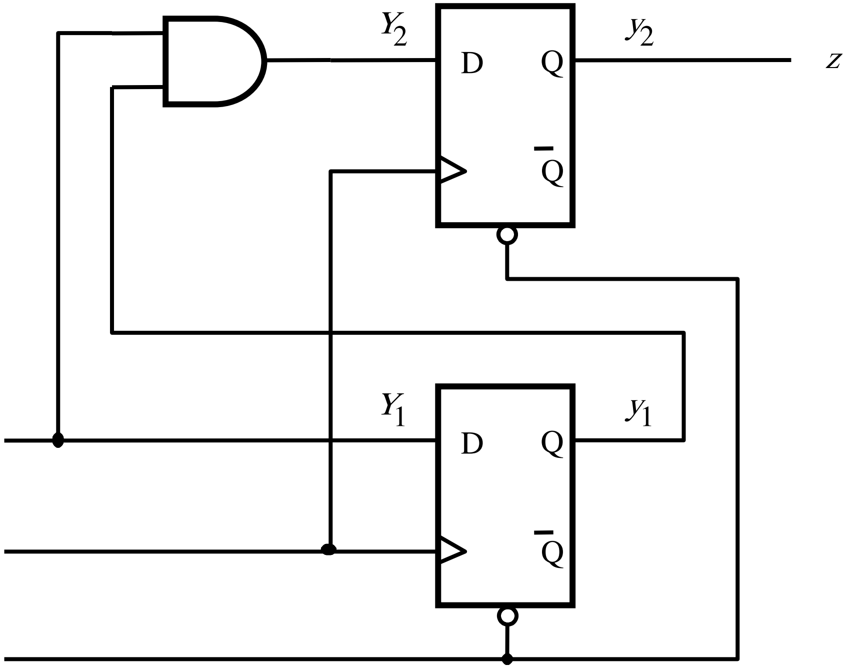
$$z = y_2$$

Circuit for improved state assignment



Previous circuit





B&V3, Figure 8.17

Mealy state machines

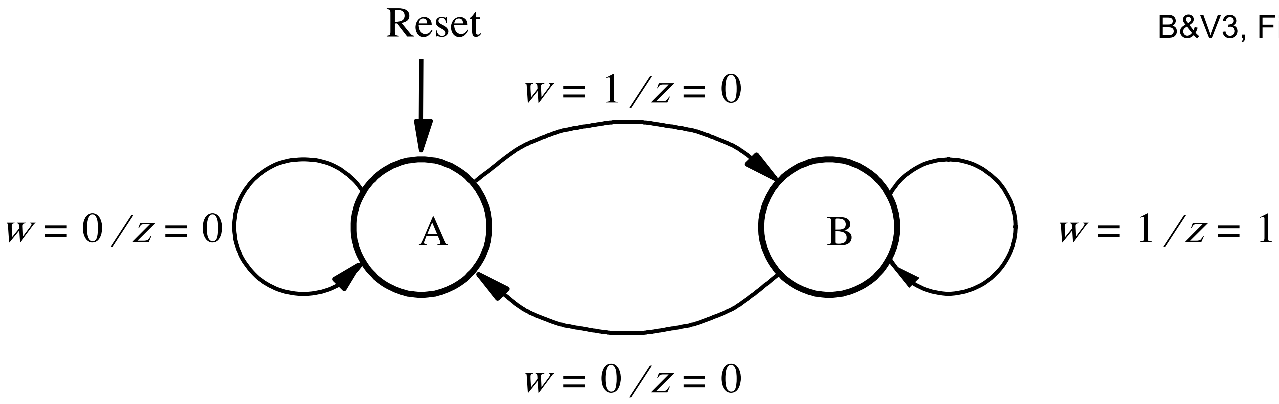
- In contrast to **Moore state machines**, in which the output is purely a function of the present state of the circuit, in **Mealy machines**, the output is also a function of the circuit's current inputs
- Mealy machines thereby provide additional flexibility and responsiveness in the design of sequential circuits

- In our first example, the output was required to become 1 in the cycle after two consecutive 1s on the input had been detected.
- Suppose instead that the output should become 1 in the clock cycle during which a second or further consecutive 1 is detected.
- The input/output sequence should then look as follows:

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

B&V3, Figure 8.22

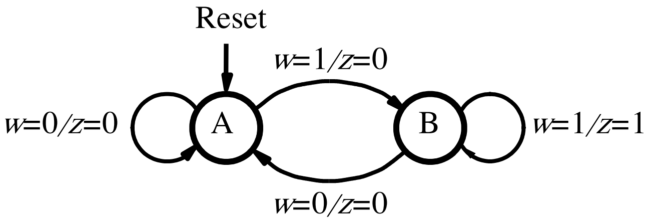
State diagram for revised example 1



B&V3, Figure 8.23

- Note that now only two states are needed because we allow the output value to depend upon the present value of the input as well as the present state of the machine
- The FSM is implemented by following the design steps previously outlined

State table for the revised example 1



Present state	Next state		Output *z*	
	w = 0	*w* = 1	*w* = 0	*w* = 1
A	A	B	0	0
B	A	B	0	1

B&V3, Figure 8.24

- Note that the output value is now dependent upon the present state as well as the input value

State-assigned table for revised example 1

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

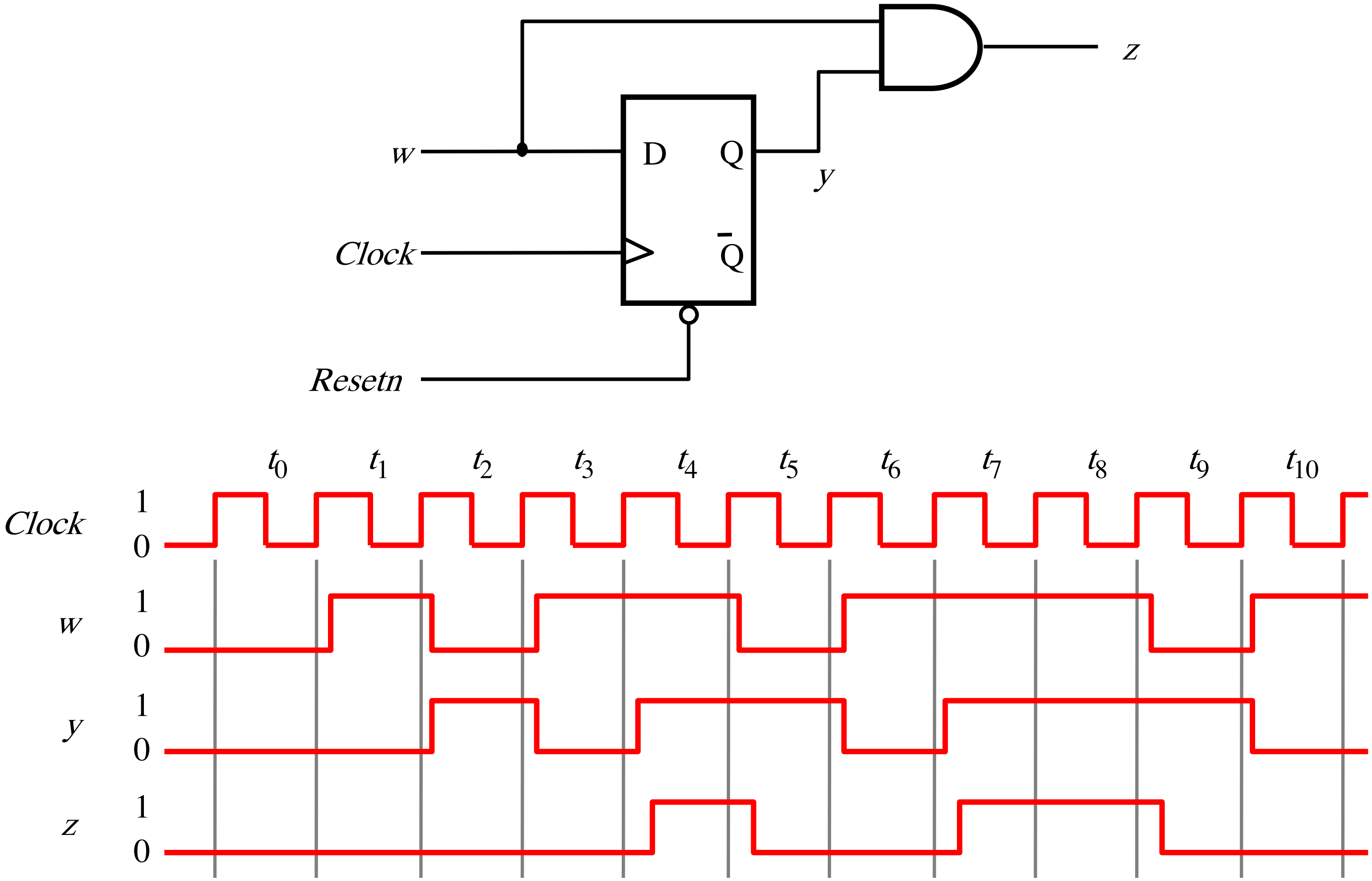
B&V3, Figure 8.25

- Assuming D-type flip-flops are selected to be used in the implementation of the machine, the next-state and output expressions are:

$$Y = D = w$$

$$z = wy$$

Implementation of revised example 1

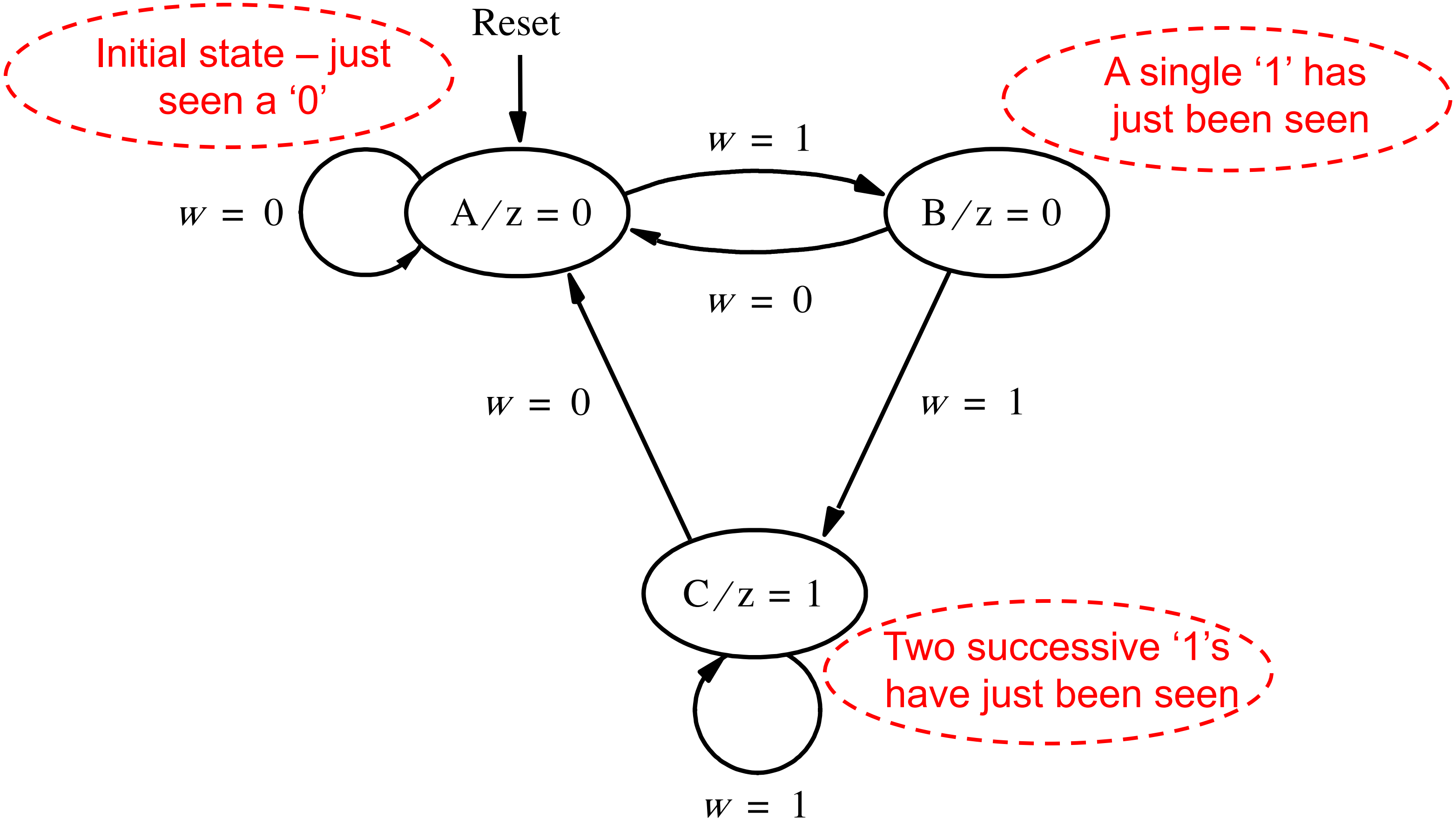


B&V3, Figure 8.26

Using CAD tools to design FSMs

- Manually designing FSMs is tedious and error-prone
  - One could use structural VHDL to input a manually derived design before simulation and implementation
  - But CAD tools offer a better alternative, namely, to enter the state diagram and to derive the design automatically
    - Graphical tools exist for this purpose
    - More commonly, behavioural HDL is used to capture the diagram
- Let’s take a look at this approach with our “two-1s” recognizer

Reminder: FSM of Example 1


Initial state – just seen a '0'
Reset
A single '1' has just been seen
w = 1
w = 0
A/z = 0
B/z = 0
w = 0
w = 0
w = 1
C/z = 1
Two successive '1's have just been seen
w = 1

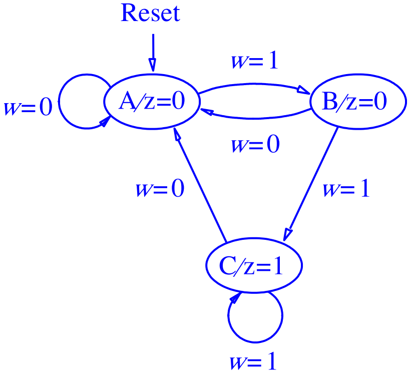
Observe that state diagrams for Moore machines associate output values with the state nodes and only list the inputs that lead to each state transition.

VHDL code for the Moore-type FSM of Example 1

- There is no standard way of describing FSMs
- Using VHDL syntax, there are a few different ways of describing FSMs
  - Note user-defined signal type [an enumerated type] (line 8)
  - The compiler chooses the number of state flip-flops and the state assignment
  - Changes in state occur on positive clock edges

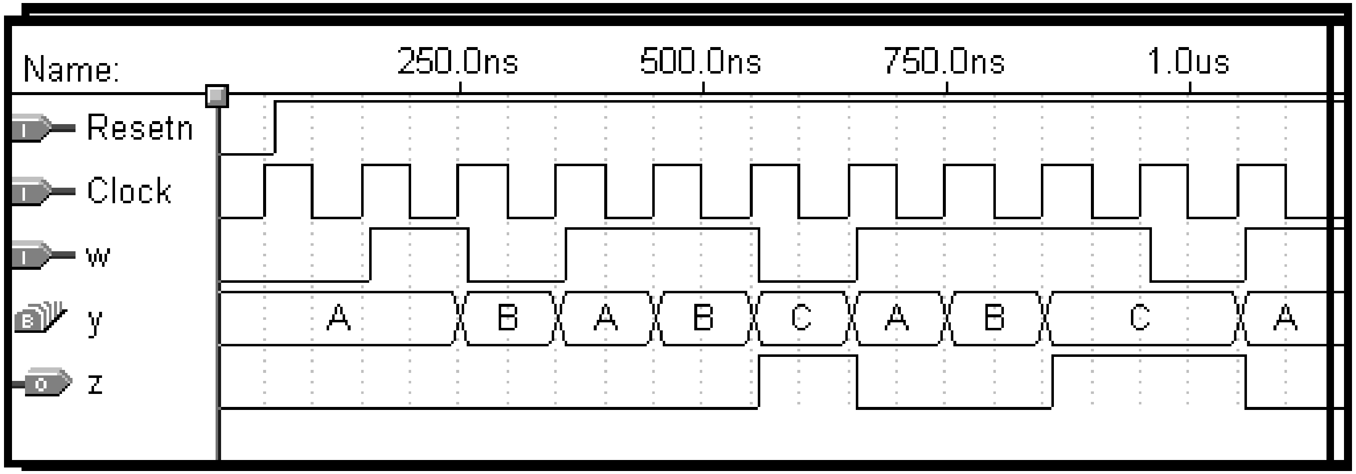
```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY simple IS
4       PORT (Clock, Resetn, w   : IN   STD_LOGIC ;
              z                  : OUT STD_LOGIC ) ;
5   END simple ;

7   ARCHITECTURE Behavior OF simple IS
8       TYPE State_type IS (A, B, C) ;
9       SIGNAL y : State_type ;
```

```
10  BEGIN
11      PROCESS ( Resetn, Clock )
12      BEGIN
13          IF Resetn = '0' THEN
14              y <= A ;
15          ELSIF (Clock'EVENT AND Clock = '1') THEN
16              CASE y IS
17                  WHEN A =>  -- each state needs a WHEN
18                      IF w = '0' THEN  -- input determines
                            y <= A ;     -- next state
                        ELSE
                            y <= B ;
                        END IF ;
                    WHEN B =>  -- when in state B
                        IF w = '0' THEN
                            y <= A ;
                        ELSE
                            y <= C ;
                        END IF ;
29                  WHEN C =>  -- state C
30                      IF w = '0' THEN
31                          y <= A ;
32                      ELSE
33                          y <= C ;
34                      END IF ;
35              END CASE ;
36          END IF ;
37      END PROCESS ;
38      z <= '1' WHEN y = C ELSE '0' ;  -- output depends on state
39  END Behavior ;
```



B&V3, Figure 8.29

Simulation results for the implemented circuit



B&V3, Figure 8.32

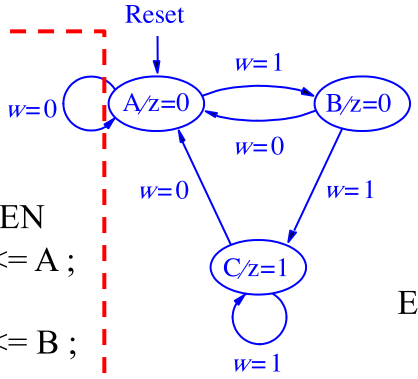
- For this simple FSM it is easy to check its correctness
- For more complex FSMs, there may be a large number of possible states and inputs, so the designer needs to plan sequences of input patterns and corresponding acceptance tests carefully

Common alternate style of VHDL code for the Moore-type FSM of Example 1

```
ARCHITECTURE Behavior OF simple IS
        TYPE State_type IS (A, B, C) ;
        SIGNAL y_present, y_next : State_type ;
BEGIN
```

Next-state logic

```
        PROCESS ( w, y_present )
        BEGIN
                CASE y_present IS
                        WHEN A =>
                                IF w = '0' THEN
                                        y_next <= A ;
                                ELSE
                                        y_next <= B ;
                                END IF ;
                        WHEN B =>
                                IF w = '0' THEN
                                        y_next <= A ;
                                ELSE
                                        y_next <= C ;
                                END IF ;
                        WHEN C =>
                                IF w = '0' THEN
                                        y_next <= A ;
                                ELSE
                                        y_next <= C ;
                        END IF ;
                END CASE ;
        END PROCESS ;
```



State flip-flops

```
        PROCESS (Clock, Resetn)
        BEGIN
                IF Resetn = '0' THEN
                        y_present <= A ;
                ELSIF (Clock'EVENT AND Clock = '1') THEN
                        y_present <= y_next ;
                END IF ;
        END PROCESS ;
```

Output logic

```
        z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

B&V3, Figure 8.33

- Note the explicit reference to the present and next state here
- First process implements the combinational logic to the left of the FFs, which determines the next state in L06/S13
- Second process implements the state FFs by giving effect to the state transition

User-defined state assignment

- It is possible for the user to manually specify a desired state assignment, but there is no standardized approach for doing so
- In *Quartus*, this is done as follows:

```
ARCHITECTURE Behavior OF simple IS
        TYPE State_TYPE IS (A, B, C) ;
        ATTRIBUTE ENUM_ENCODING                   : STRING ;
        ATTRIBUTE ENUM_ENCODING OF State_type   : TYPE IS "00 01 11" ;
        SIGNAL y_present, y_next : State_type ;
BEGIN
    .
    .
                                        .
```

B&V3, Figure 8.34

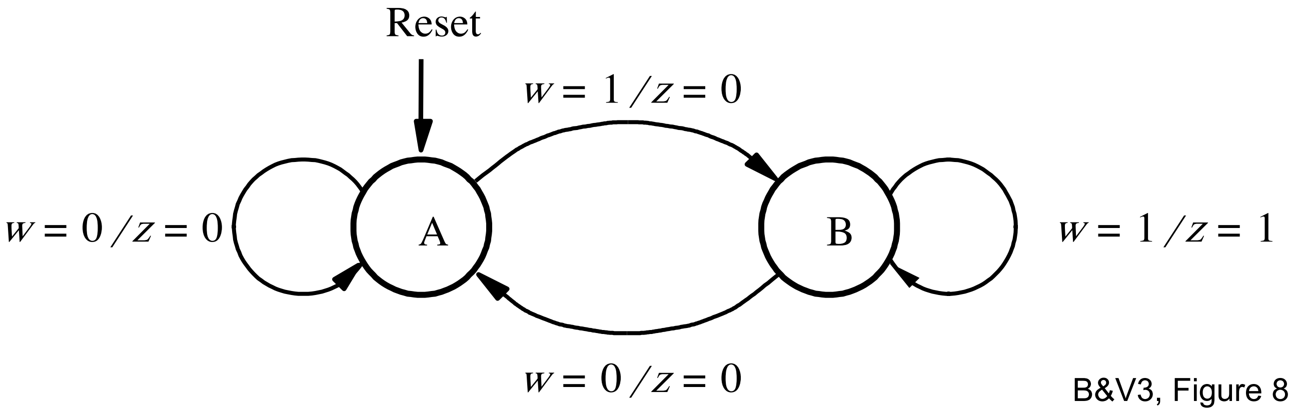
- ***But note that this should not normally be necessary***

Using constants for manual state assignment – works with all VHDL compilers

- Note the need for a WHEN OTHERS clause in the next_state logic as there is no enumerated state_type; *y_present* is simply a STD_LOGIC_VECTOR

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

ENTITY simple IS
    PORT ( Clock, Resetn, w : IN       STD_LOGIC ;
                z                 : OUT    STD_LOGIC ) ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
    CONSTANT A :  STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
    CONSTANT B :  STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
    CONSTANT C :  STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN y_next <= A ;
                ELSE y_next <= C ;
                END IF ;
            WHEN OTHERS =>
                y_next <= A ;
        END CASE ;
    END PROCESS ;
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Reminder: Mealy-type FSM for Example 1

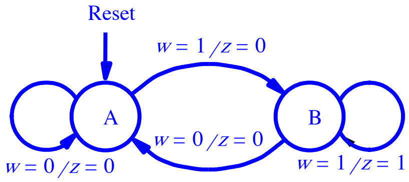


B&V3, Figure 8.23

Observe that state diagrams for Mealy machines list the output values together with the input that leads to each state transition.

VHDL code for the Mealy-type FSM of Example 1

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY mealy IS
    PORT ( Clock, Resetn, 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
            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 ;
        END IF ;
    END PROCESS ;
```



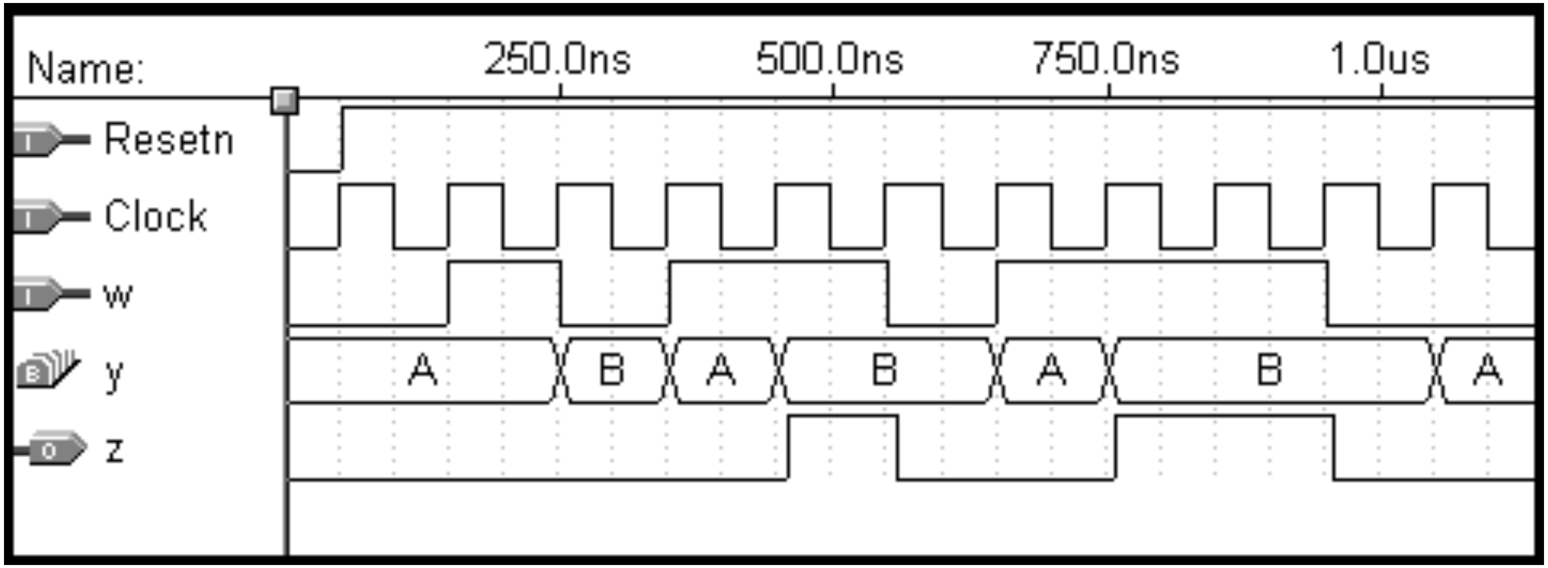
```
    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Output logic

B&V3, Figure 8.36

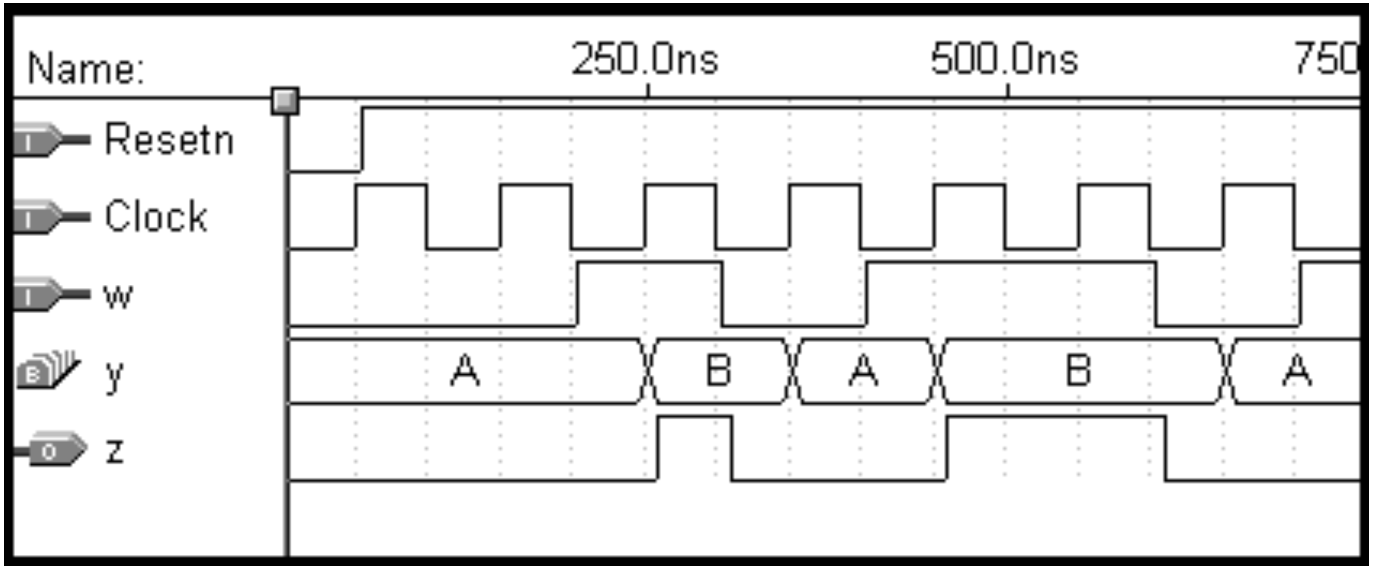
- Note use of second process to determine output independently of the state transition logic
- It is also common to separate the next-state logic from the state transition logic, as we saw in slide L06/S31, in which case, there would be 3 processes for the Mealy machine

Simulation results for the Mealy machine



B&V3, Figure 8.37

Potential problem with asynchronous inputs to Mealy machine

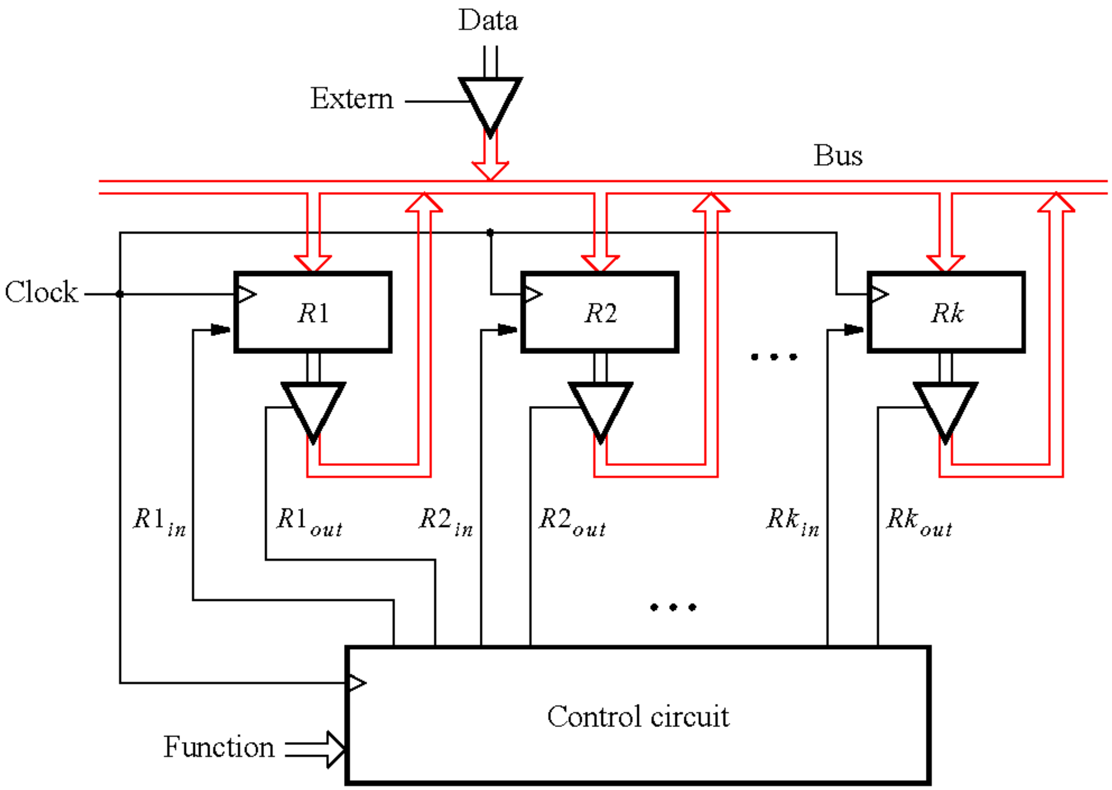


B&V3, Figure 8.38

- Here changes in *w* occur after negative clock edges
- *z* should not be asserted until after *w* is asserted for 1 clock period
  - If *z* is input to another circuit that is not controlled by the same clock, we could get big problems (downstream errors)
  - On the other hand, a downstream circuit controlled by the same clock should ignore the erroneous pulse

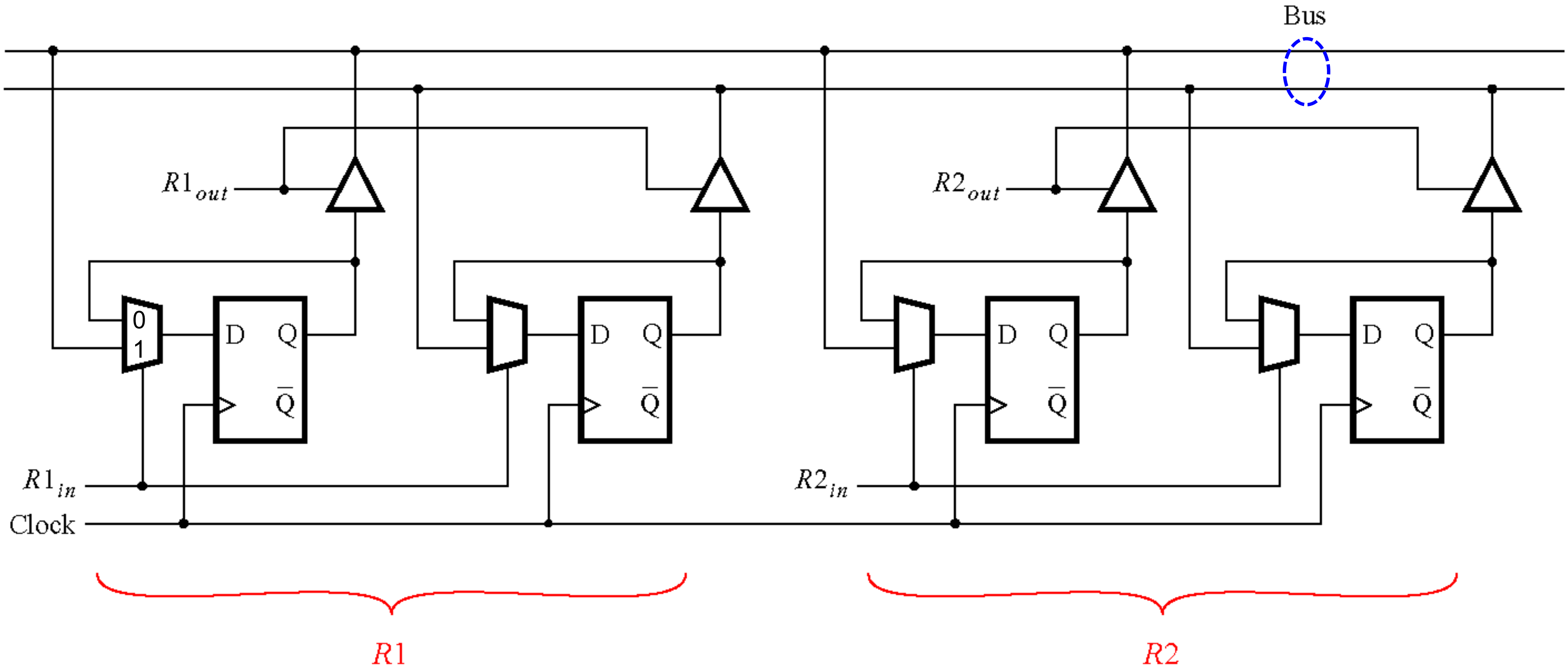
Example 2: Design a control circuit for a bus-based register swap

- Consider the control required to swap the contents of R1 and R2 via a bus using R3 for temporary storage

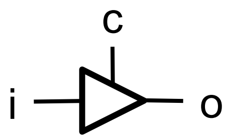


Details for connecting registers to a bus

- Consider two 2-bit registers
  - 3-state buffers used to avoid “tying” outputs together



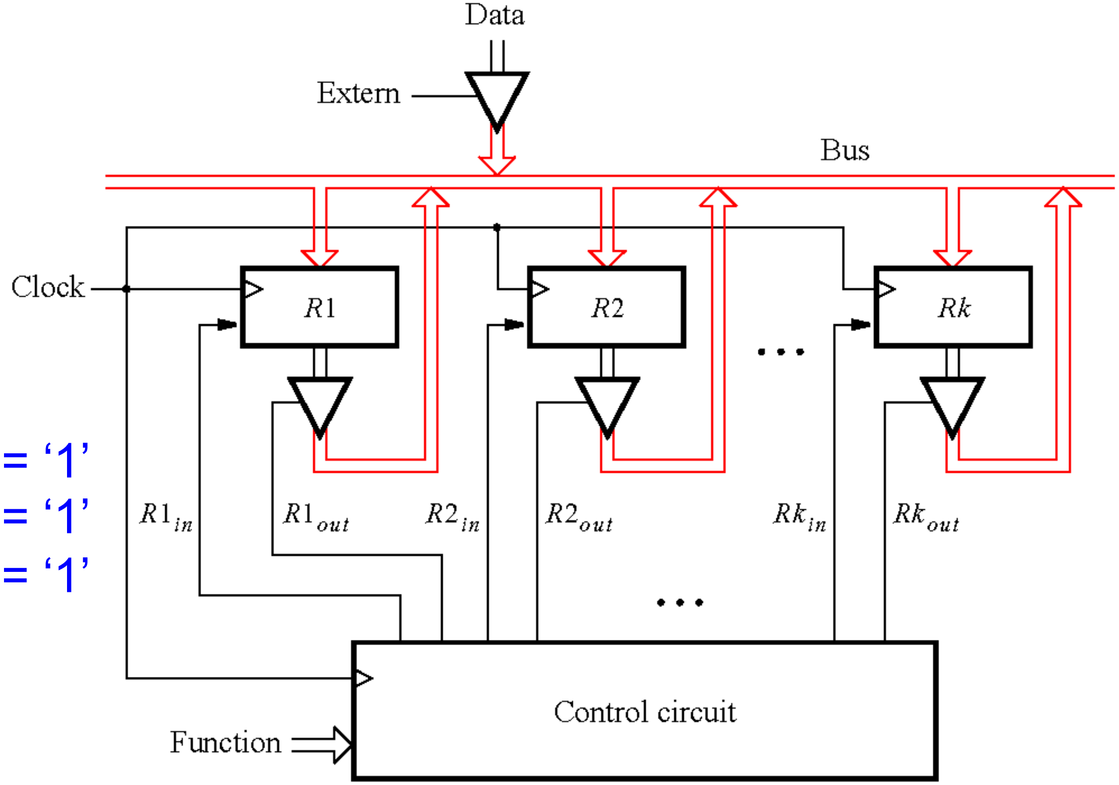
3-state buffer:
Tri-state® buffer:



c	o
0	Z = high impedance (open circuit)
1	i

Control circuit design

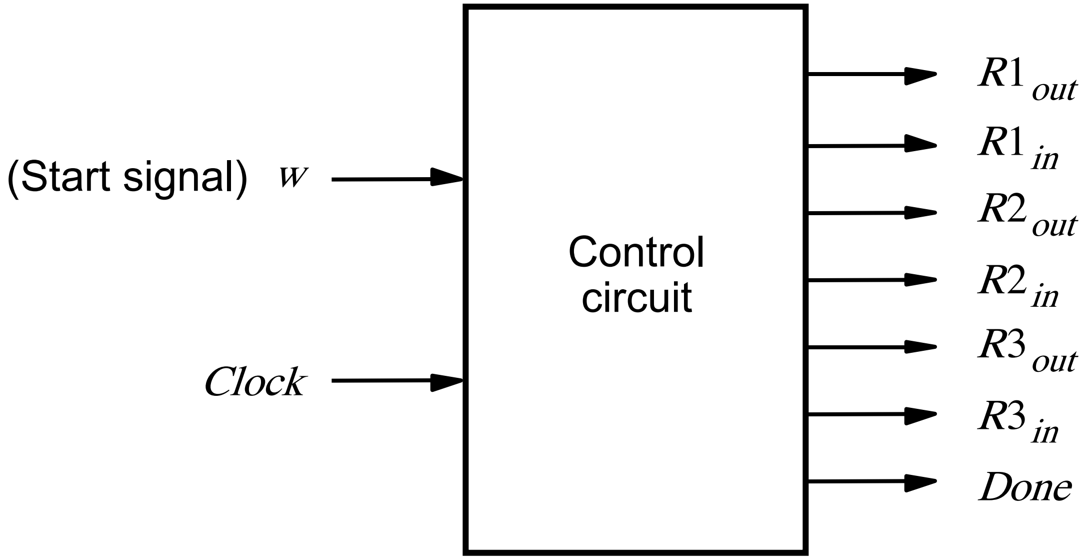
- Consider the control required to swap the contents of R1 and R2 using R3 for temporary storage
  - What are the individual register transfers required to effect the swap?
  - Which control signals need to be asserted for each transfer?
  - When & how should the control signals be sequenced?



In successive steps:

1. R3 <= R2: R3in <= ‘1’; R2out <= ‘1’
2. R2 <= R1: R2in <= ‘1’; R1out <= ‘1’
3. R1 <= R3: R1in <= ‘1’; R3out <= ‘1’

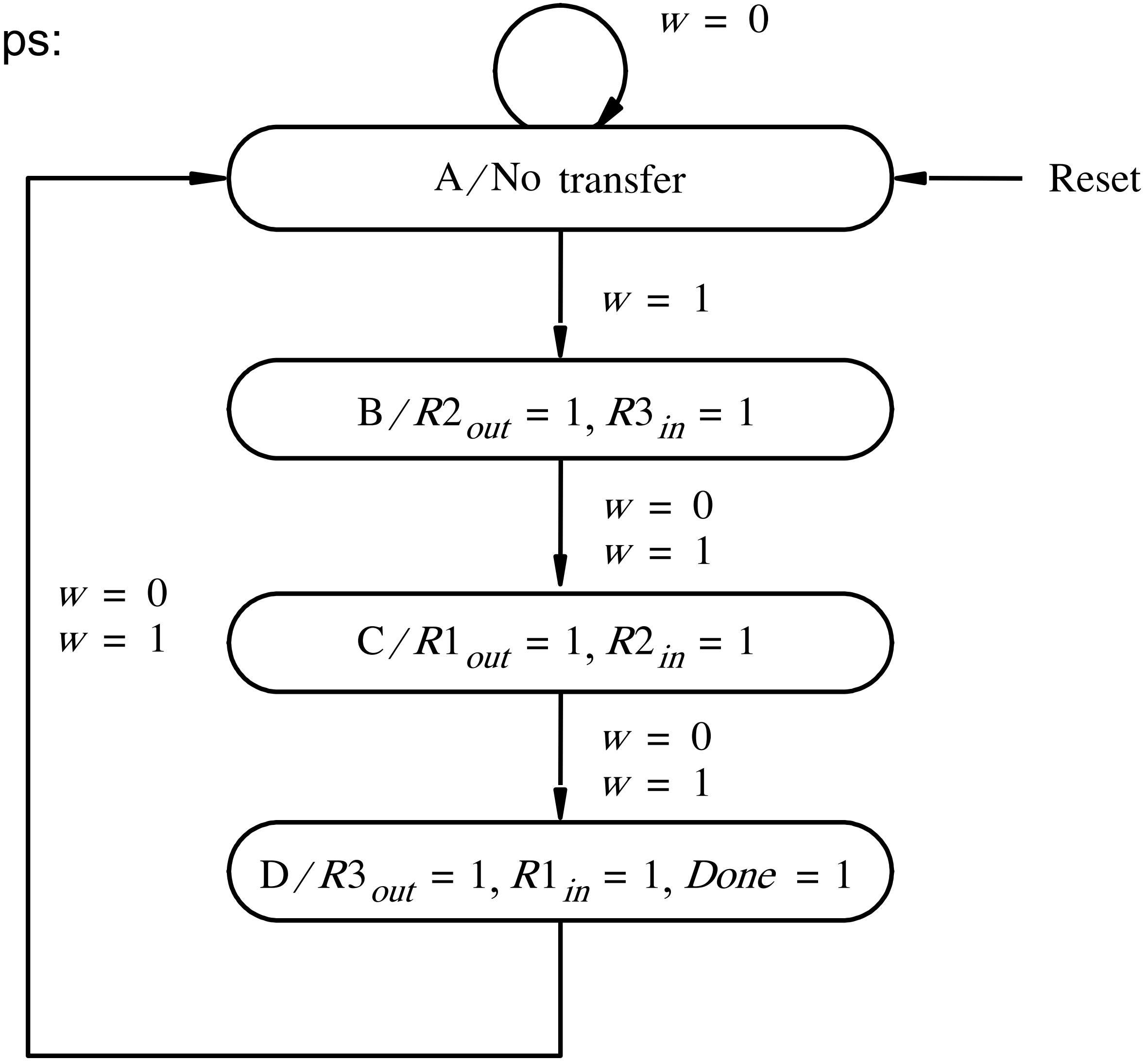
Signals needed by control circuit



B&V3, Figure 8.10

Moore state diagram for example 2

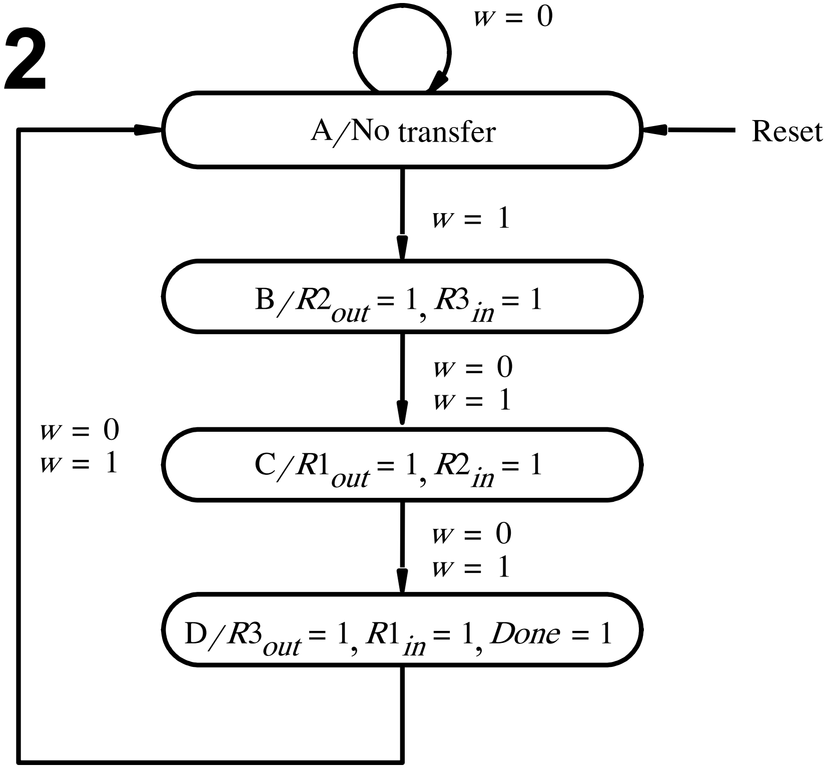
In successive steps:
1. R3 <= R2
2. R2 <= R1
3. R1 <= R3



$w = 0$
A/No transfer
Reset
$w = 1$
B/$R2_{out} = 1, R3_{in} = 1$
$w = 0$
$w = 1$
$w = 0$
$w = 1$
C/$R1_{out} = 1, R2_{in} = 1$
$w = 0$
$w = 1$
D/$R3_{out} = 1, R1_{in} = 1, Done = 1$

B&V3, Figure 8.11

State table for example 2



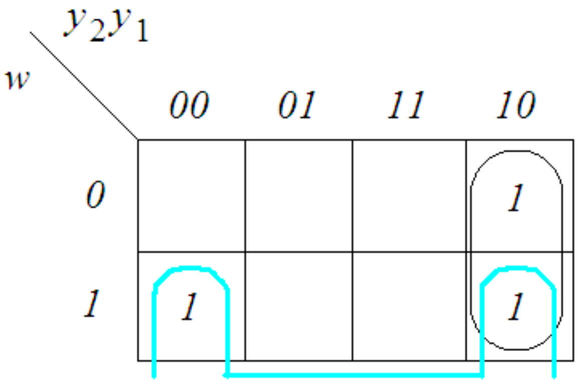
Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

B&V3, Figure 8.12

State-assigned table, next-state and output expressions for example 2 using D-type FFs

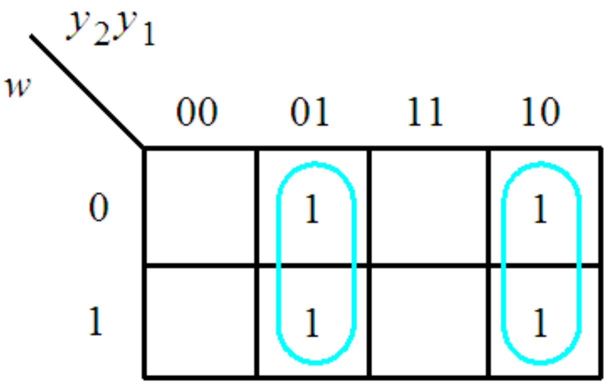
	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	0 1	0	0	0	0	0	0	0
B	01	10	1 0	0	0	1	0	0	1	0
C	10	11	1 1	1	0	0	1	0	0	0
D	11	00	0 0	0	1	0	0	1	0	1

B&V3, Figure 8.13



w \ $y_2 y_1$	00	01	11	10
0				1
1	1			1

$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$



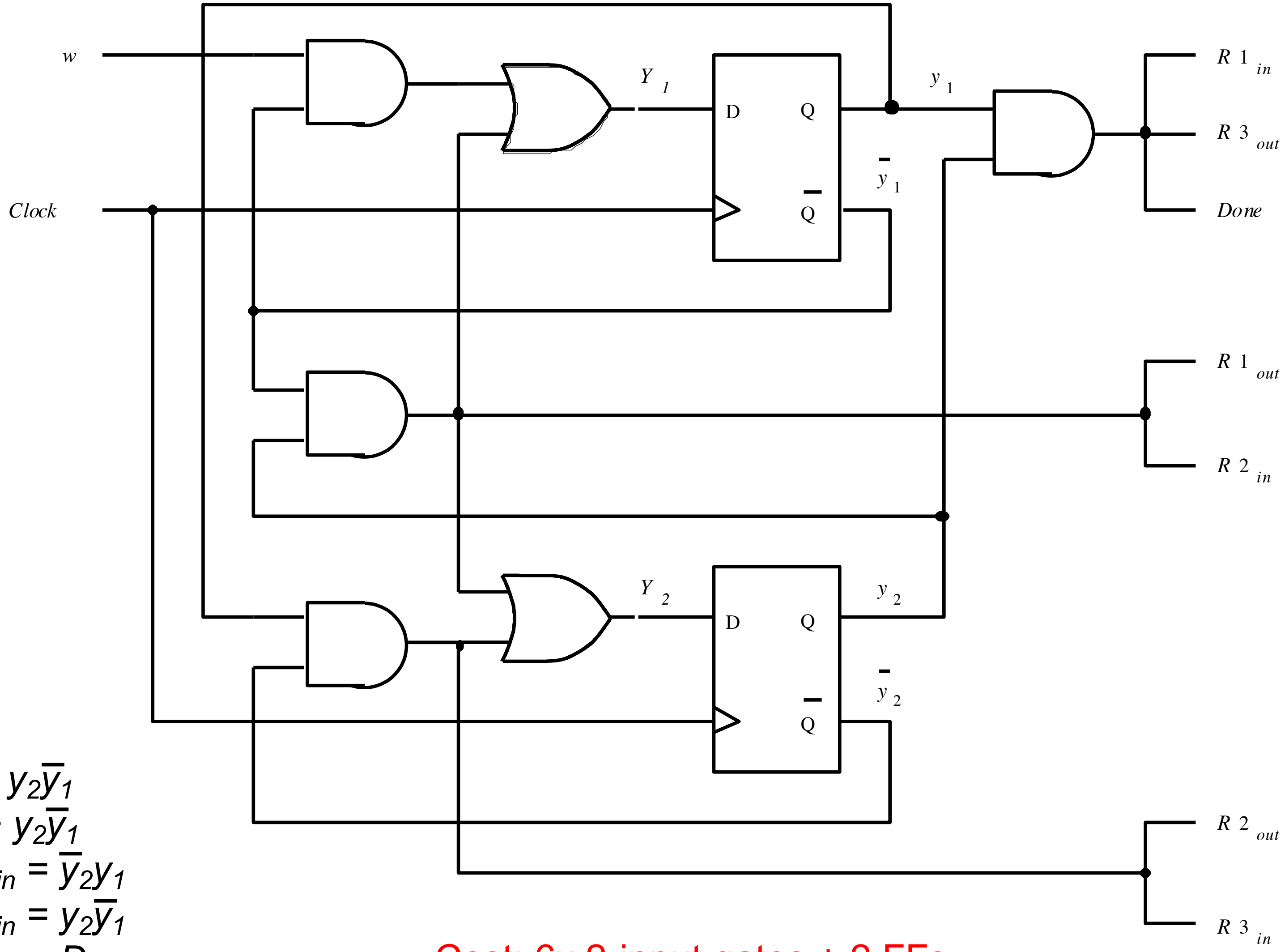
w \ $y_2 y_1$	00	01	11	10
0		1		1
1		1		1

$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

$R2_{out} = R3_{in} = \bar{y}_2 y_1$
$R1_{out} = R2_{in} = y_2 \bar{y}_1$
$R1_{in} = R3_{out} = Done = y_2 y_1$

B&V3, Figure 8.14

Final implementation of example 2



$Y_1 = w\overline{y}_1 + y_2\overline{y}_1$
$Y_2 = \overline{y}_2y_1 + y_2\overline{y}_1$
$R2_{out} = R3_{in} = \overline{y}_2y_1$
$R1_{out} = R2_{in} = y_2\overline{y}_1$
$R1_{in} = R3_{out} = Done = y_2y_1$

Cost: 6x 2-input gates + 2 FFs

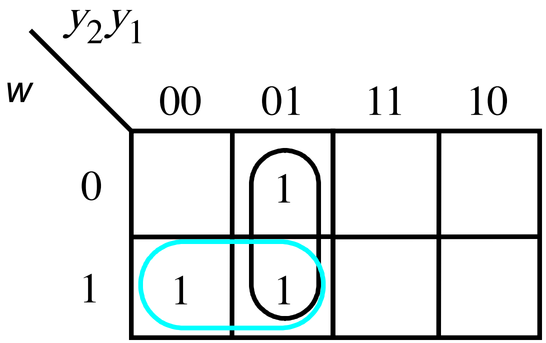
B&V3, Figure 8.15

Improved state assignment for example 2

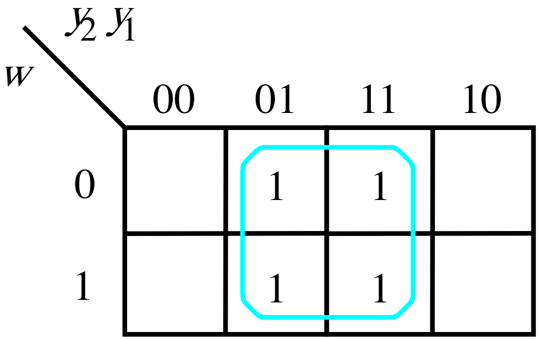
- Swapping the assignments for states *C* and *D*…

	Present state	Next state		Outputs						
		$w=0$	$w=1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	0 1	0	0	0	0	0	0	0
B	01	11	1 1	0	0	1	0	0	1	0
C	11	10	1 0	1	0	0	1	0	0	0
D	10	00	0 0	0	1	0	0	1	0	1

B&V3, Figure 8.18



$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$



$$Y_2 = y_1$$

B&V3, Figure 8.19

$R2_{out} = R3_{in} = \bar{y}_2y_1$
$R1_{out} = R2_{in} = y_2y_1$
$R1_{in} = R3_{out} = Done = y_2\bar{y}_1$

Cost: 5x 2-input gates + 2 FFs

One-hot encoding of example 2

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	$y_4y_3y_2y_1$	$Y_4Y_3Y_2Y_1$	$Y_4Y_3Y_2Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 001	0001	0010	0	0	0	0	0	0	0
B	0 010	0100	0100	0	0	1	0	0	1	0
C	0 100	1000	1000	1	0	0	1	0	0	0
D	1 000	0001	0001	0	1	0	0	1	0	1

B&V3, Figure 8.21

- Treating the remaining 12 valuations of the state variables as don't cares results in:

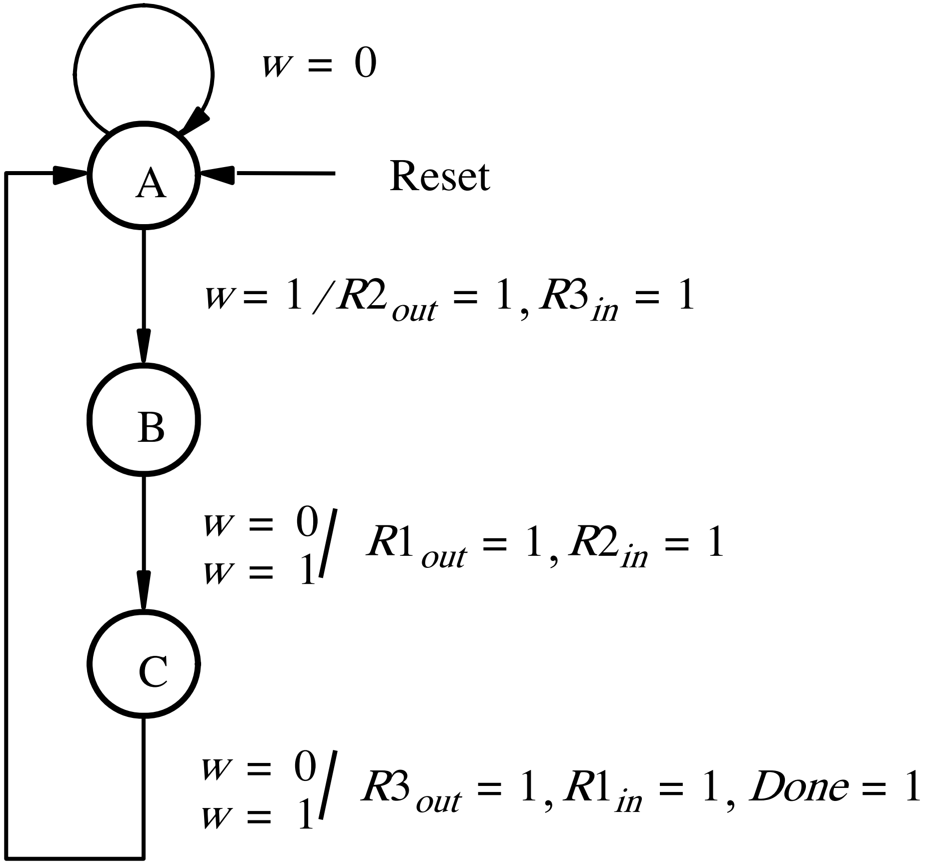
$$Y_1 = \overline{w}y_1 + y_4, \quad Y_2 = wy_1, \quad Y_3 = y_2 \quad \text{and} \quad Y_4 = y_3$$

- The output expressions are just the outputs of the flip-flops:

$$R2_{out} = R3_{in} = y_2, \quad R1_{out} = R2_{in} = y_3 \quad \text{and} \quad R1_{in} = R3_{out} = Done = y_4$$

- These expressions are simpler than previously seen, but 4 FFs are still needed
- Simpler expressions, as often result from one-hot encodings, may lead to faster circuits

Mealy-type FSM for swapping two registers



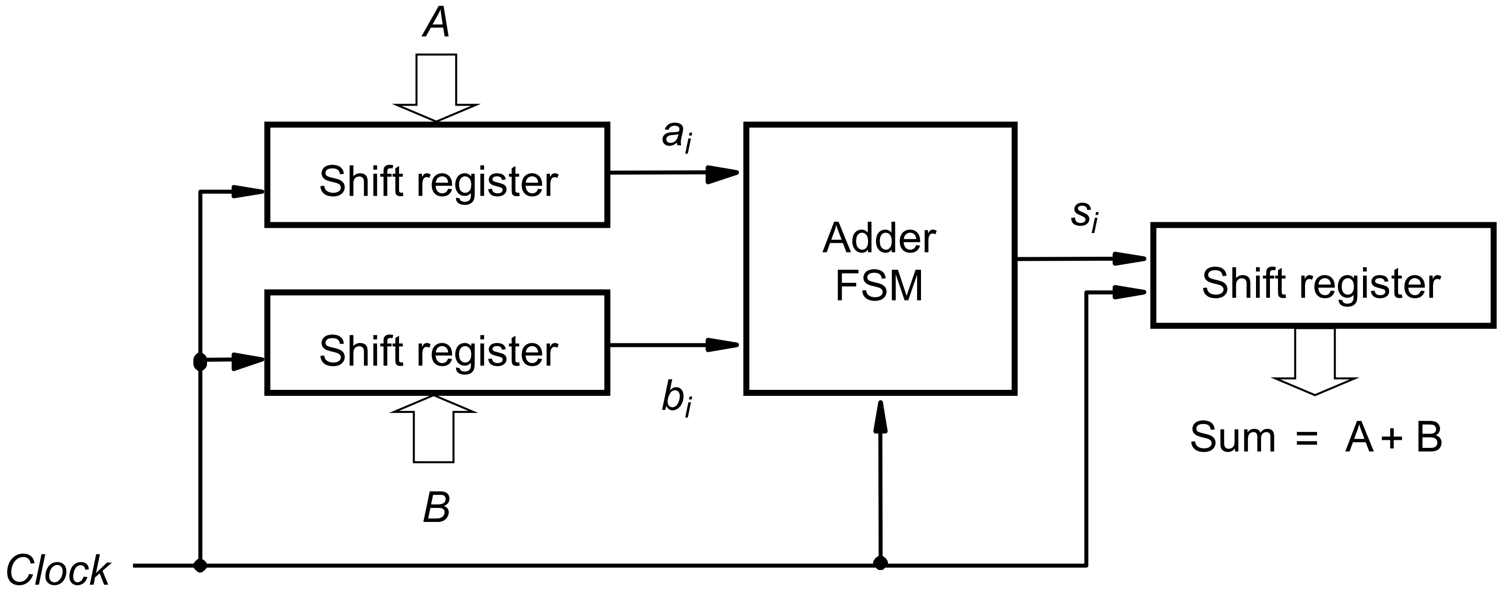
$w = 0$
A
Reset
$w = 1 / R2_{out} = 1, R3_{in} = 1$
B
$w = 0$ / $w = 1$ / $R1_{out} = 1, R2_{in} = 1$
C
$w = 0$ / $w = 1$ / $R3_{out} = 1, R1_{in} = 1, Done = 1$

B&V3, Figure 8.28

- While the Mealy implementation only requires 3 states, this does not necessarily imply a simpler circuit since we still need 2 FFs.
- The most important difference with the Moore version is the timing of the output signals, which are generated one clock cycle sooner.
- Note also that the entire swap only takes 3 clock cycles for the Mealy-type FSM, whereas it takes 4 clock cycles to complete for the Moore machine.

Complete design example: serial addition

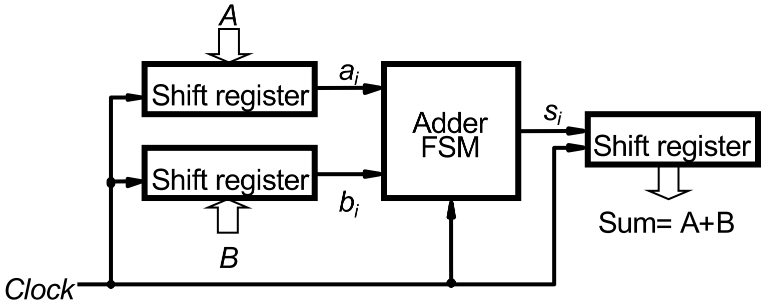
- We’ve looked at several addition schemes that added two *n*-bit numbers in parallel (e.g., ripple-carry, carry-lookahead)
- In these schemes, the speed of the adder is important, but fast adders are more complex and thus more expensive
- If speed is not important, then a more cost-effective option is to use a *serial adder* in which bits are added a pair at a time



B&V3, Figure 8.39

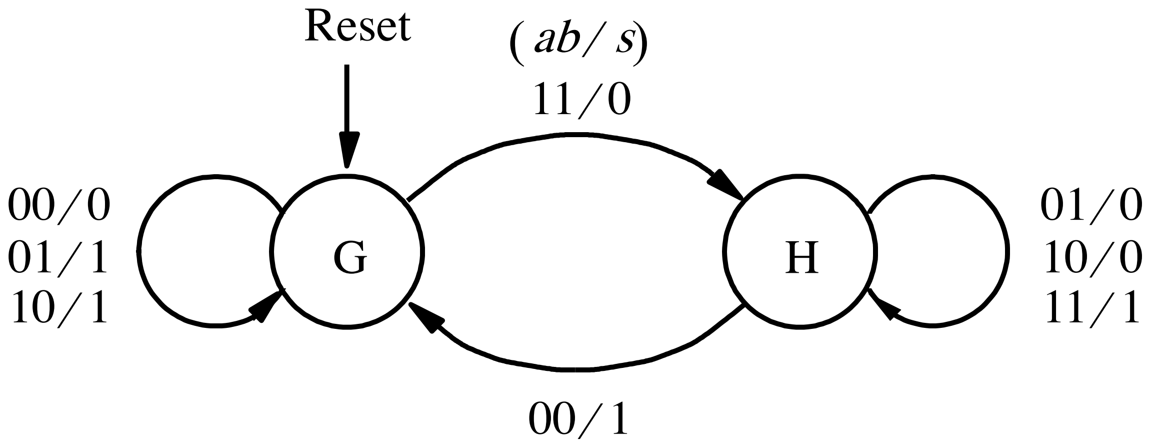
Serial addition

- Let $A = a_{n-1}a_{n-2}\ldots a_0$ and $B = b_{n-1}b_{n-2}\ldots b_0$ be two unsigned numbers that have to be added to produce $S = s_{n-1}s_{n-2}\ldots s_0$
- Our task is to design a circuit that will perform the serial addition, dealing with a pair of bits in one clock cycle
- Having loaded a pair of numbers in parallel, the process starts by adding a_0 and b_0 and shifting the result, s_0, into the sum register. In the next clock cycle, bits a_1 and b_1 are added, including a possible carry from bit-position 0.
- Assume we are to use positive edge-triggered D-type flip-flops in the design



State diagram for the serial adder FSM

- An FSM is needed since the sum bit produced differs depending upon the carry produced in the previous cycle
- We therefore need two states depending upon the value of the carry-in bit



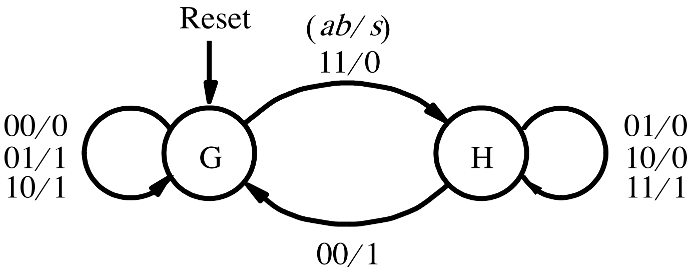
G: carry-in = 0

H: carry-in = 1

B&V3, Figure 8.40

State table for the serial adder FSM

- The state table is readily obtained from the state diagram



Present state	Next state				Outputs *s*			
	ab=00	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

B&V3, Figure 8.41

State-assigned table for serial adder FSM

Present	Next state				Output			
state	*ab*=00	01	10	11	00	01	10	11
y	*Y*				*s*			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

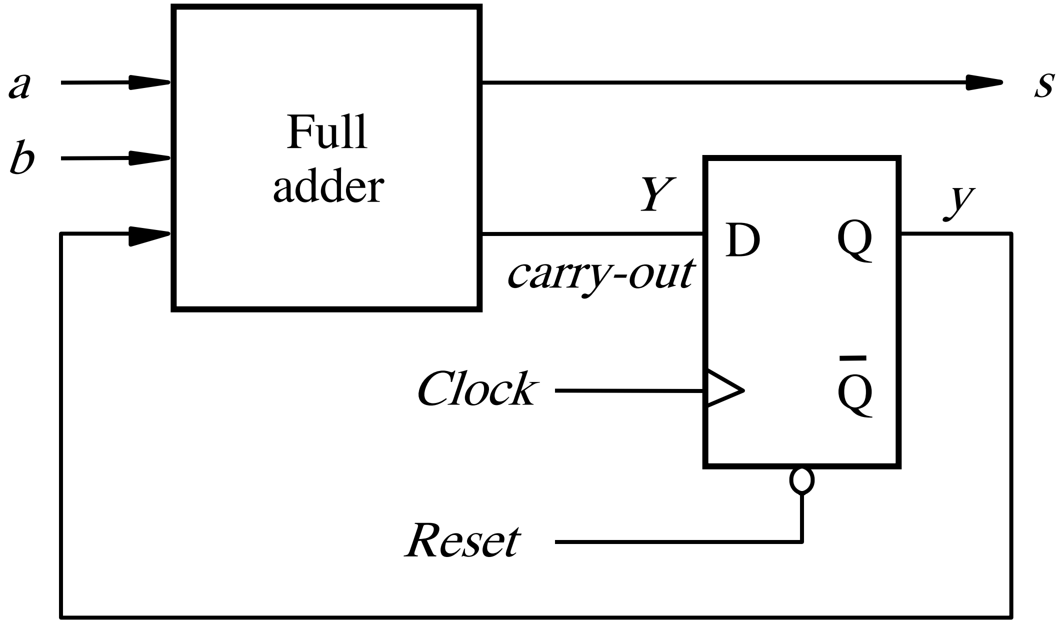
B&V3, Figure 8.42

- A simple state assignment leads to the following next-state and output equations:

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

- These are the same as for a full-adder with carry-in *y,* carry-out *Y*, and sum *s*

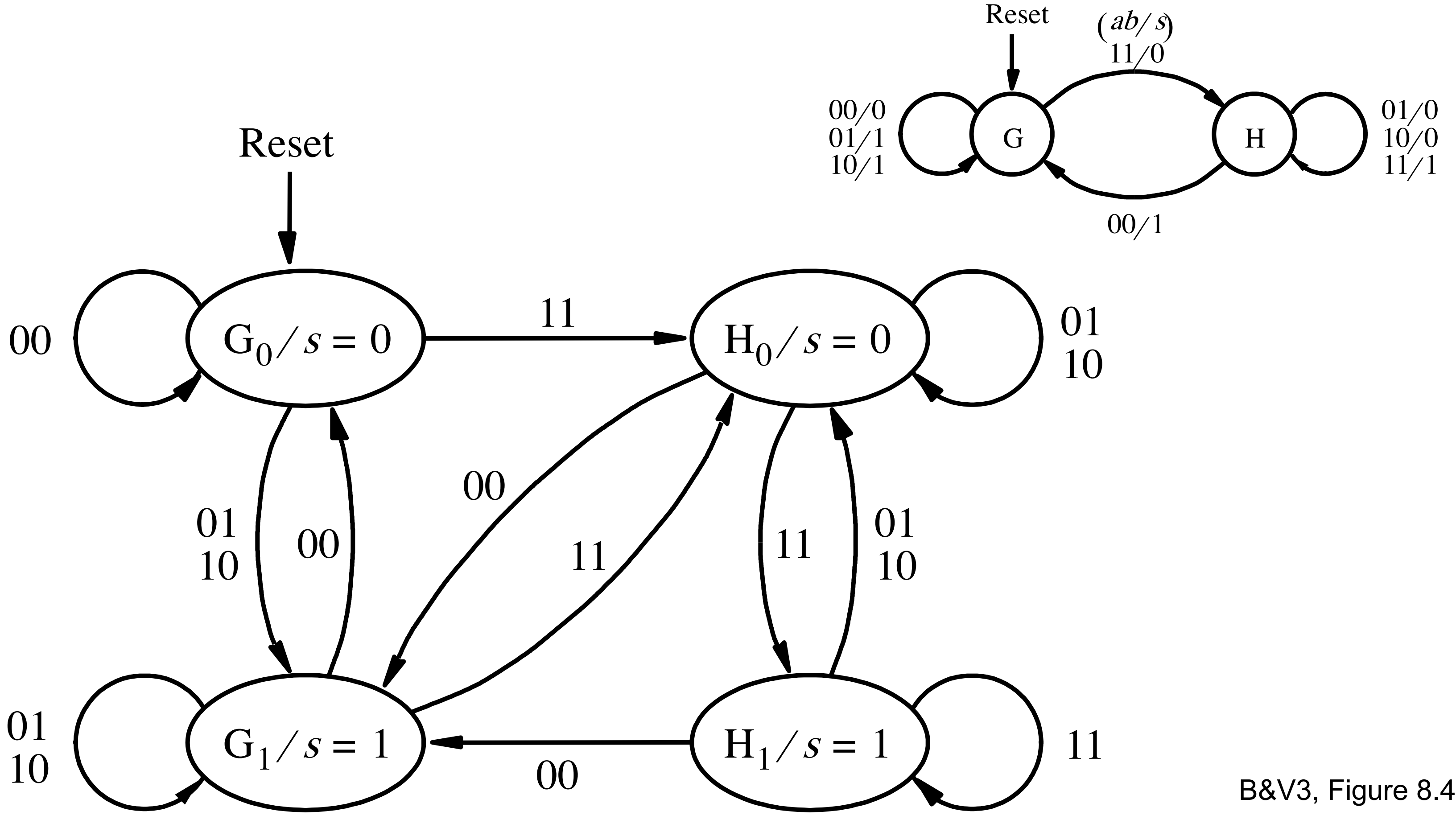
Circuit for the serial adder FSM



B&V3, Figure 8.43

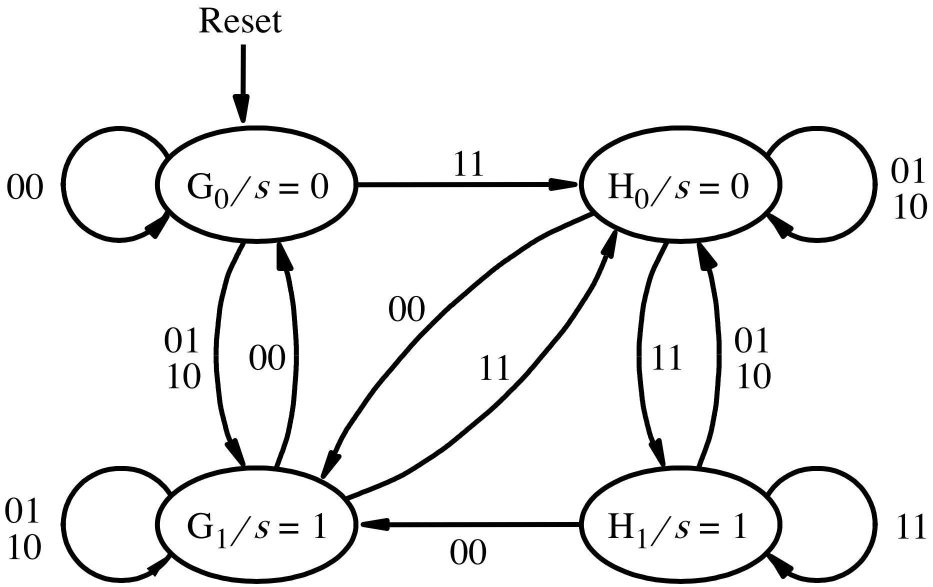
State diagram for a Moore-type serial adder FSM

- Now let’s consider the design of the equivalent Moore-type FSM
  - We then need a separate state, i.e. two states, for each different output we found in the state diagram of the Mealy machine (slide L06/S51)



B&V3, Figure 8.44

State table for the Moore-type serial adder FSM



Present state	Next state				Output
	ab=00	01	10	11	*s*
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

B&V3, Figure 8.45

State-assigned table for the Moore-type serial adder FSM

Present state	Next state				Output
	ab=00	01	10	11	
y_2y_1	Y_2Y_1				s
00	0 0	01	0 1	10	0
01	0 0	01	0 1	10	1
10	0 1	10	1 0	11	0
11	0 1	10	1 0	11	1

B&V3, Figure 8.46

- The next-state and output equations are:

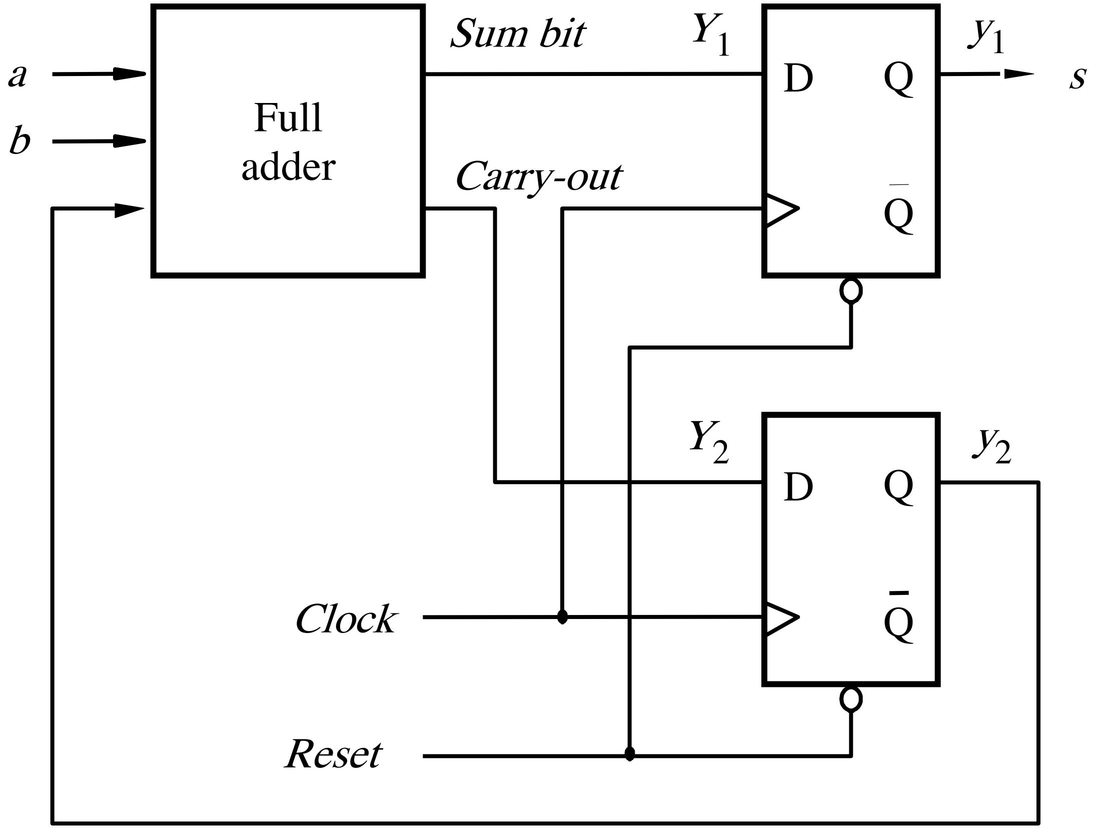
$$Y_1 = a \oplus b \oplus y_2$$

$$Y_2 = ab + ay_2 + by_2$$

$$s = y_1$$

- The expressions for Y1 and Y2 correspond to the sum and carry-out expressions in the full-adder circuit

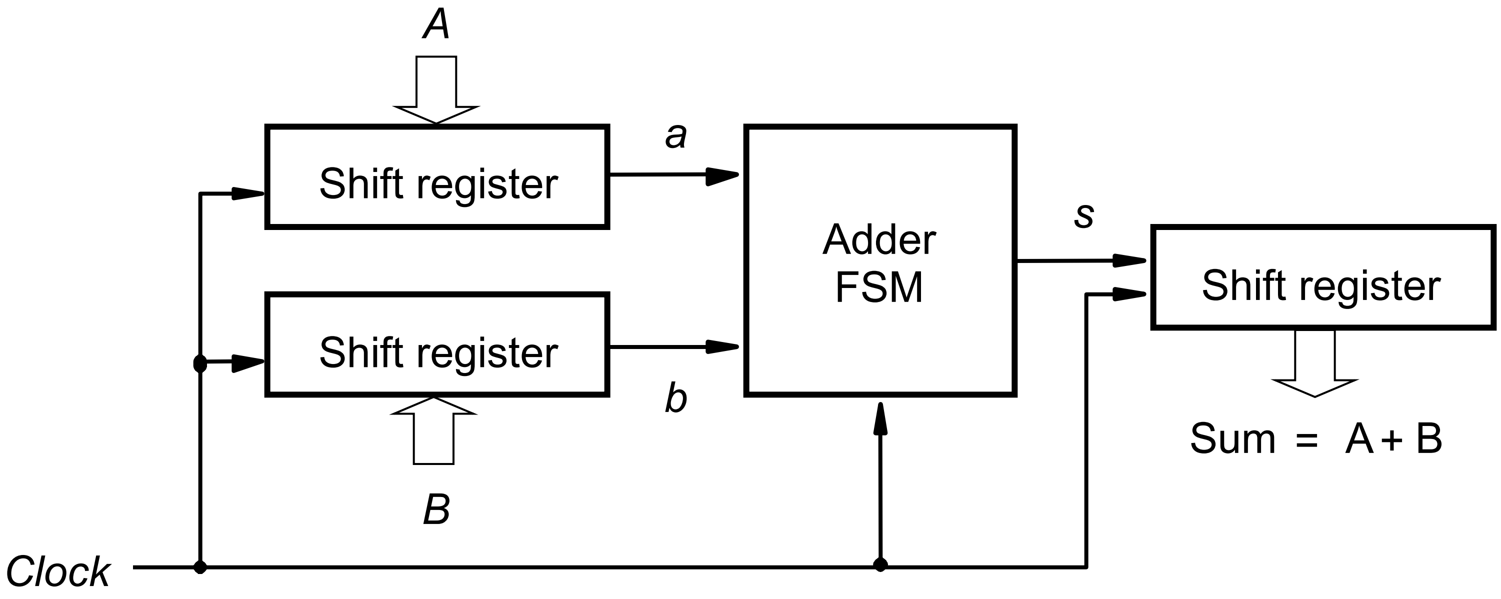
Circuit for the Moore-type serial adder FSM



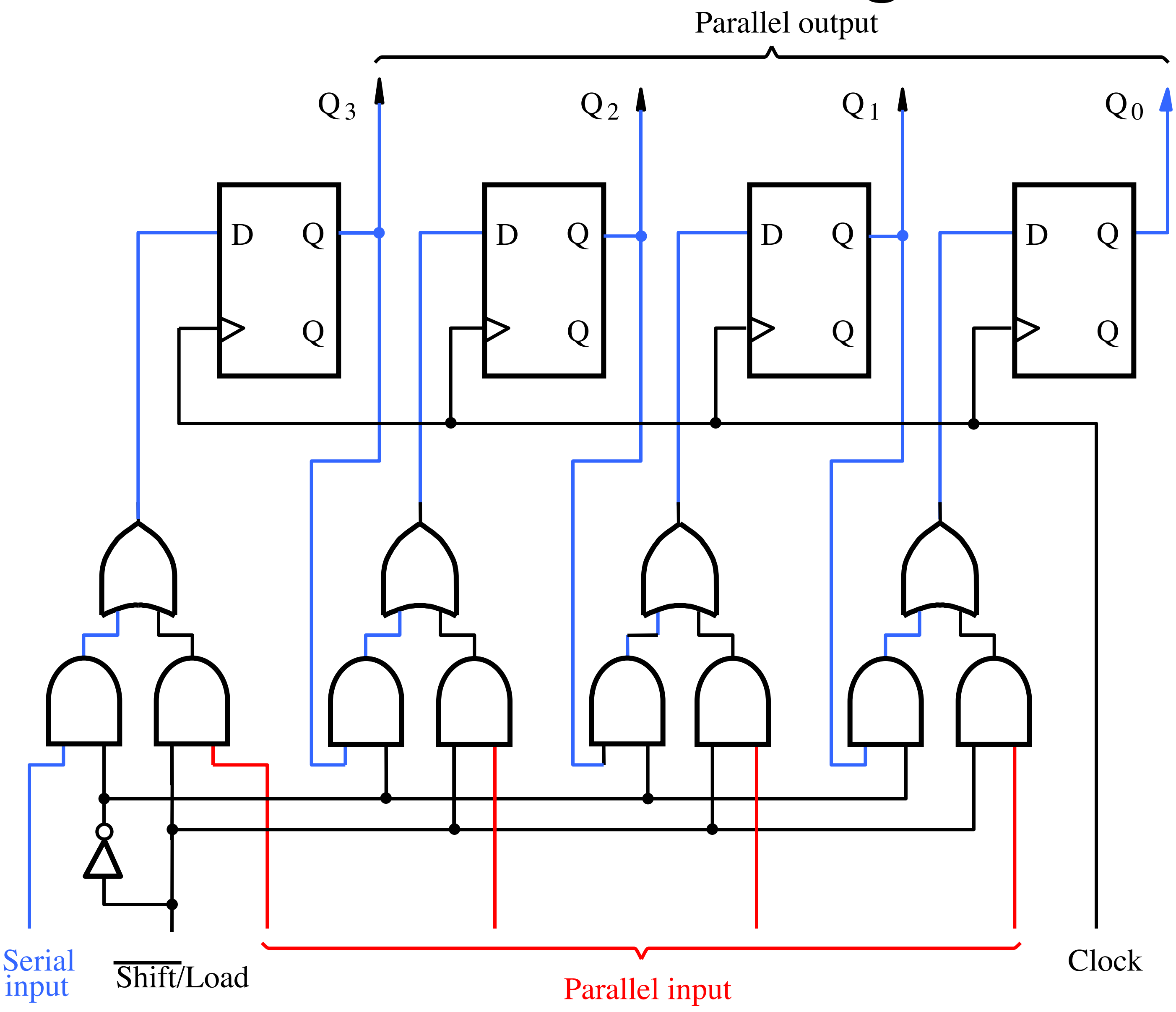
B&V3, Figure 8.47

- Referring back to the Mealy circuit of L06/S54, the output *s* is now passed through an extra flip-flop and thus delayed by one clock cycle

How do we build the serial adder? How do we control its operation?



Recall: Parallel-access shift register

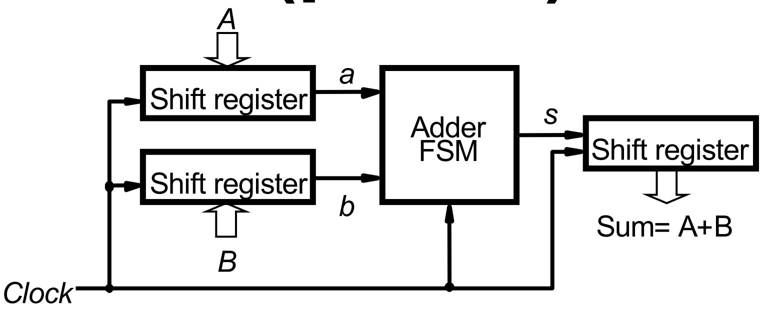


Code for a left-to-right shift register with an enable input

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
-- left-to-right shift register with parallel load and enable
ENTITY shiftrne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT (  R         : IN       STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
            L, E, w   : IN       STD_LOGIC ;
            Clock     : IN       STD_LOGIC ;
            Q         : BUFFER   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftrne ;
ARCHITECTURE Behavior OF shiftrne IS
BEGIN
    PROCESS
    BEGIN
          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
          IF E = '1' THEN                             -- if enabled
             IF L = '1' THEN                          --   depending upon the load signal
                Q <= R ;                              --     either load a new word in parallel
             ELSE
                Genbits: FOR i IN 0 TO N-2 LOOP       --     or shift the word to right
                     Q(i) <= Q(i+1) ;
                END LOOP ;
                Q(N-1) <= w ;
            END IF ;
          END IF ;
    END PROCESS ;
END Behavior ;
```

B&V3, Figure 8.48

VHDL code for the serial adder (part A)



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

ENTITY serial IS
    GENERIC ( length : INTEGER := 8 ) ;
    PORT ( Clock  : IN        STD_LOGIC ;
           Reset  : IN        STD_LOGIC ;
           A, B   : IN        STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
           Sum    : BUFFER    STD_LOGIC_VECTOR(length-1 DOWNTO 0) );
END serial ;

ARCHITECTURE Behavior OF serial IS
    COMPONENT shiftrne    -- include the parallel load shift register as a component
      GENERIC ( N : INTEGER := 4 ) ;
      PORT ( R        : IN        STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
             L, E, w  : IN        STD_LOGIC ;       -- load, enable, shift-in
             Clock    : IN        STD_LOGIC ;
             Q        : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;

SIGNAL QA, QB, Null_in : STD_LOGIC_VECTOR(length-1 DOWNTO 0) ;
SIGNAL s, Low, High, Run : STD_LOGIC ;
SIGNAL Count : INTEGER RANGE 0 TO length ;
TYPE State_type IS (G, H) ;        -- our Mealy machine
SIGNAL y : State_type ;
```

… continued in Part *b*

B&V3, Figure 8.49

VHDL code for the serial adder (part B)

```
BEGIN
     Low <= '0' ; High <= '1' ;
     ShiftA: shiftrne GENERIC MAP (N => length)
          PORT MAP ( A, Reset, High, Low, Clock, QA ) ;
     ShiftB: shiftrne GENERIC MAP (N => length)
          PORT MAP ( B, Reset, High, Low, Clock, QB ) ;
     AdderFSM: PROCESS ( Reset, Clock )
     BEGIN
          IF Reset = '1' THEN
               y <= G ;
          ELSIF Clock'EVENT AND Clock = '1' THEN
               CASE y IS
                  WHEN G =>
                       IF QA(0) = '1' AND QB(0) = '1' THEN
                            y <= H ;
                       ELSE y <= G ;
                       END IF ;
                  WHEN H =>
                       IF QA(0) = '0' AND QB(0) = '0' THEN
                            y <= G ;
                       ELSE y <= H ;
                       END IF ;
               END CASE ;
          END IF ;
     END PROCESS AdderFSM ;
```

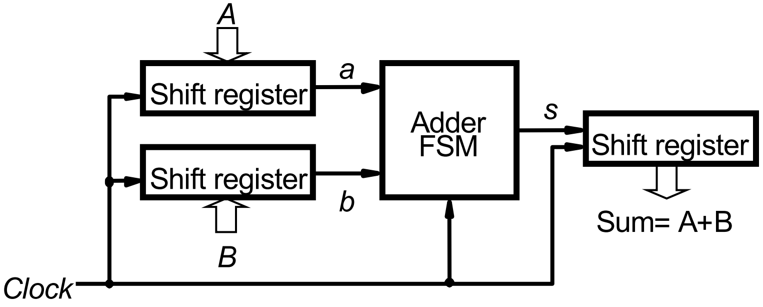
load, enable, shift-in

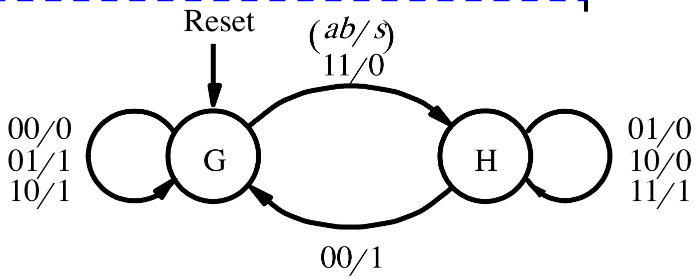
FSM state transitions

```
     WITH y SELECT
          s <=  QA(0) XOR QB(0) WHEN G,
                NOT ( QA(0) XOR QB(0) ) WHEN H ;
     Null_in <= (OTHERS => '0') ;
     ShiftSum: shiftrne GENERIC MAP ( N => length )
                    PORT MAP ( Null_in, Reset, Run, s, Clock, Sum ) ;
     Stop: PROCESS   -- shift in the result until the shift register is full
     BEGIN
          WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
          IF Reset = '1' THEN
               Count <= length ;
          ELSIF Run = '1' THEN
               Count <= Count -1 ;
          END IF ;
     END PROCESS ;
     Run <= '0' WHEN Count = 0 ELSE '1' ;  -- stops counter and ShiftSum
END Behavior ;
```

FSM output

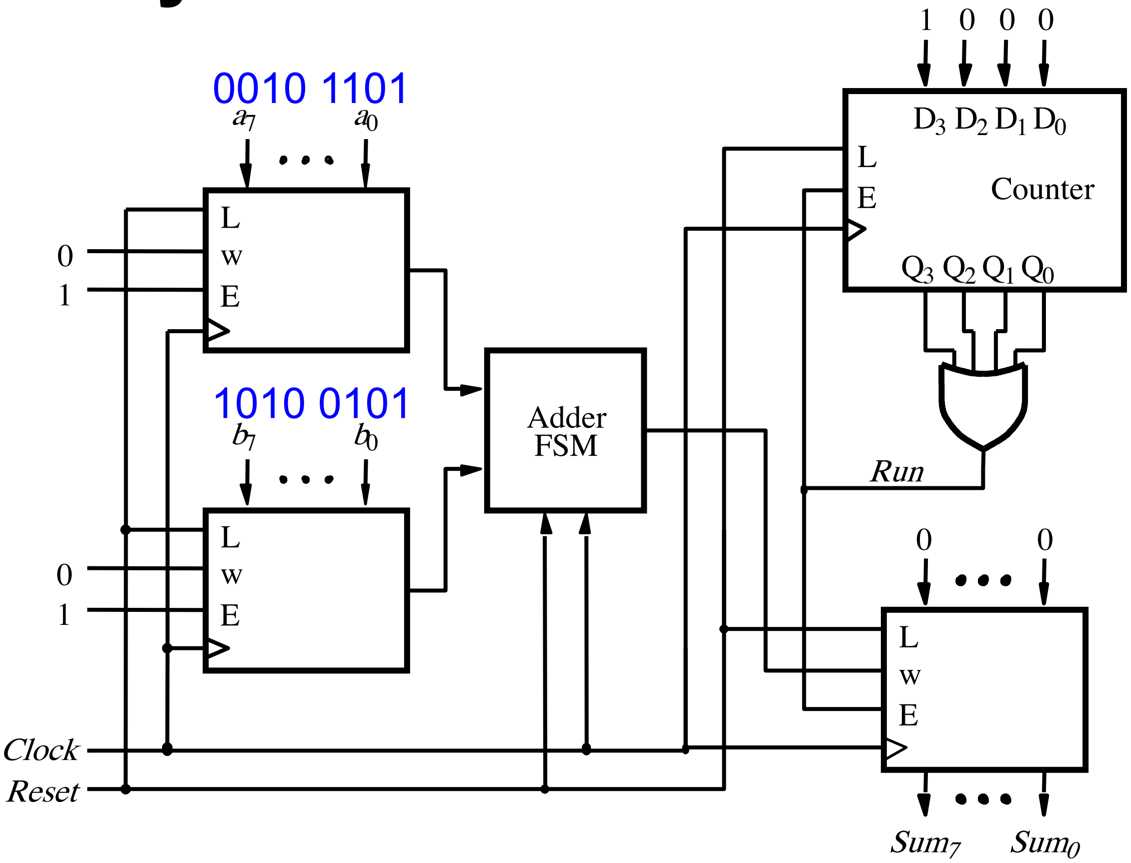
load, enable, shift-in



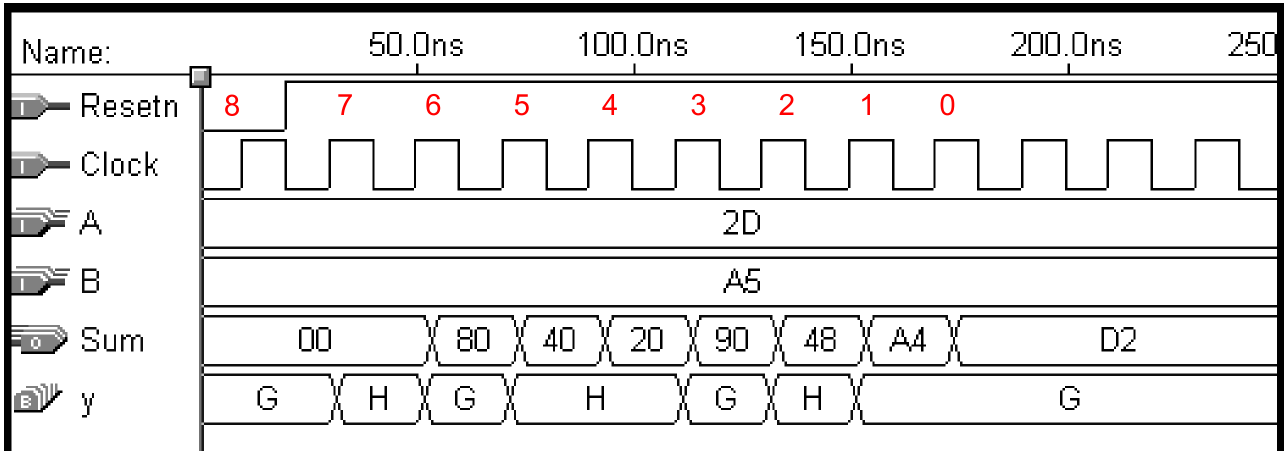


B&V3, Figure 8.49

Synthesized serial adder



a_0 b_0	Count	Run	State	Sum
	~R 8	1	G	0000 0000 (00)
1 1	↑Clk 7	1	H	0000 0000 (00)
0 0	↑Clk 6	1	G	1000 0000 (80)
1 1	↑Clk 5	1	H	0100 0000 (40)
1 0	↑Clk 4	1	H	0010 0000 (20)
0 0	↑Clk 3	1	G	1001 0000 (90)
1 1	↑Clk 2	1	H	0100 1000 (48)
0 0	↑Clk 1	1	G	1010 0100 (A4)
0 1	↑Clk 0	0	G	1101 0010 (D2)



B&V3, Figure 8.50

State minimization

- How do we know the state diagram we have constructed is as simple as can be – has as few states as possible?
- Minimizing the number of states:
  - ⇒ possibly fewer flip-flops needed to represent states
  - ⇒ complexity of the FSM's combinational logic may be reduced

- To reduce the number of states in a state diagram, some states must be equivalent to others in terms of their contribution to the overall behaviour of the FSM

- **Definition:** Two states S_i and S_j are said to be *equivalent* if and only if for every possible input sequence the same output sequence will be produced regardless of whether S_i or S_j is the initial state

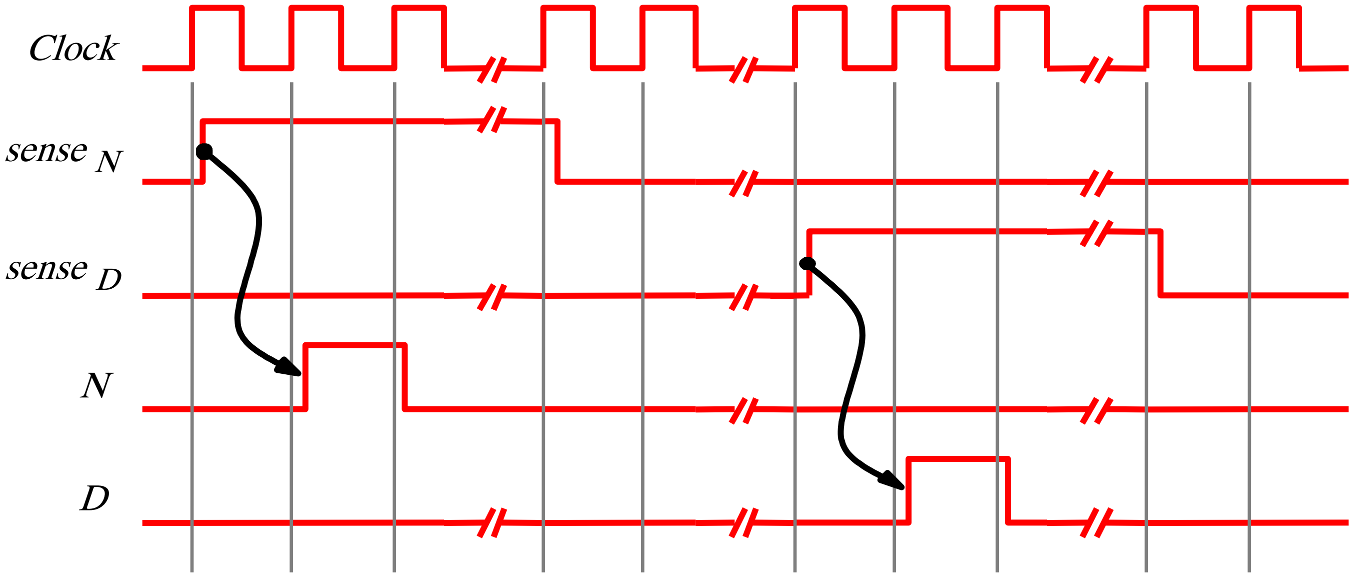
State minimization procedure

- It is possible to define an exhaustive minimization procedure, as used in CAD tools, but it is tedious
- We'll look at a more efficient but limited method to get the general idea
- We exploit the idea that it is easy to show that some states are definitely not equivalent and partition the set of states into equivalent sets of states on that basis:
  - First, partition the states into different sets on the basis of the different output values they produce
  - Next, consider the members of each set and determine whether or not they all have next states that belong to the same sets, i.e. refine the partitioning until all states within each set have the same next state set for each possible input value
  - When the partitioning cannot be further refined, replace each set with a single state – a minimal number of states has been found

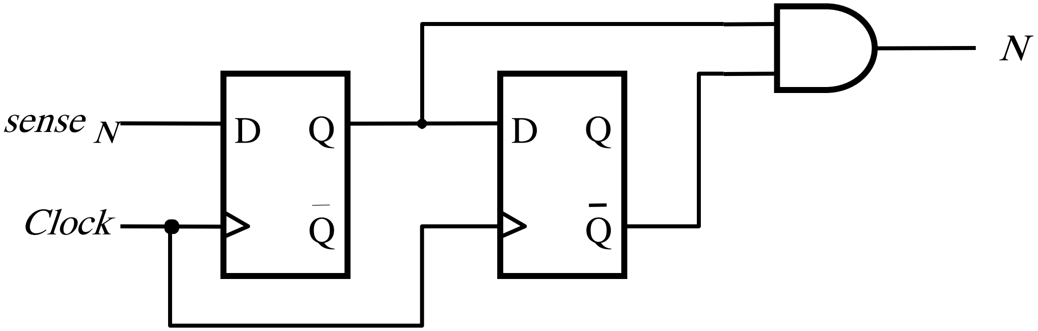
Vending machine (Example 8.6)

- Suppose we need to design the FSM for a vending machine with the following requirements:
  - The machine accepts nickels (5¢) and dimes (10¢)
  - It takes 15¢ for an item to be dispensed from the machine
  - If 20¢ is deposited, the machine will not provide change, but it will credit the buyer with 5¢ and wait for the buyer to make a second purchase
- All electronic signals are synchronized to the positive edge of the clock signal
- A mechanical coin receptor generates a very slow *sense* signal, and these trigger a single pulse corresponding to the type of coin deposited

Signals for the vending machine



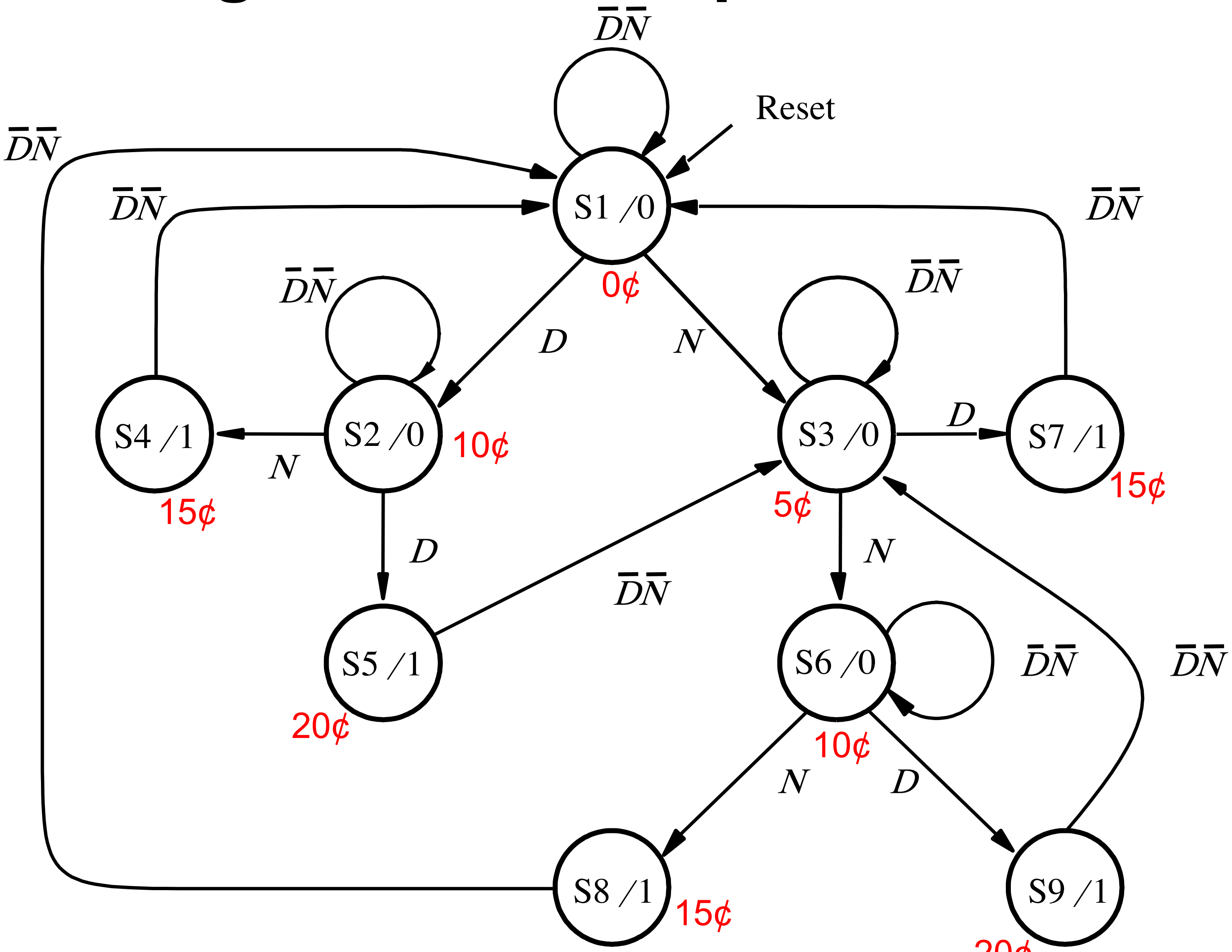
(a) Timing diagram



(b) Circuit that generates *N*

B&V3, Figure 8.53

State diagram for Example 8.6

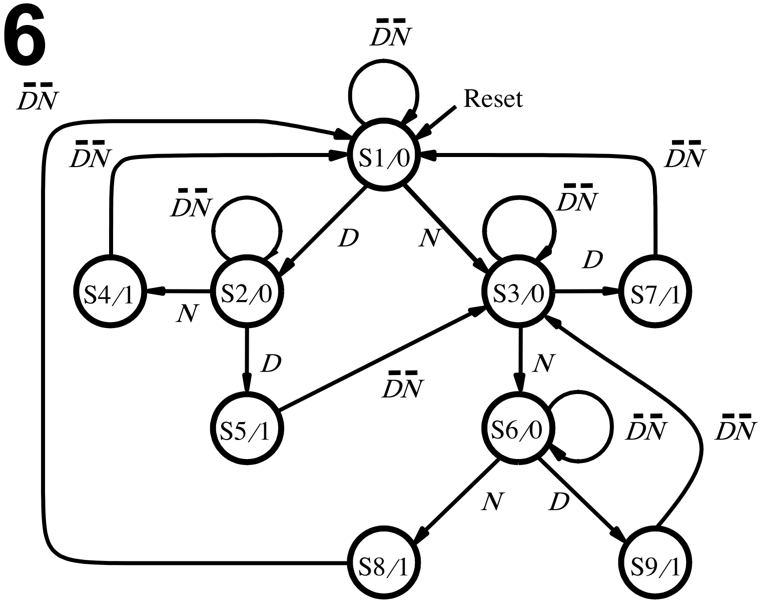


B&V3, Figure 8.54

State table for Example 8.6

Present state	Next state				Output
	DN= 00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

B&V3, Figure 8.55



Partition states based on output:

(S1 S2 S3 S6)(S4 S5 S7 S8 S9)

S S S S

S O S O

S O O O

Refine sets based on next state:

(S1)(S2 S6)(S3)(S4 S7 S8)(S5 S9)

Halt, since no further refinement possible

Can this state table be minimized?

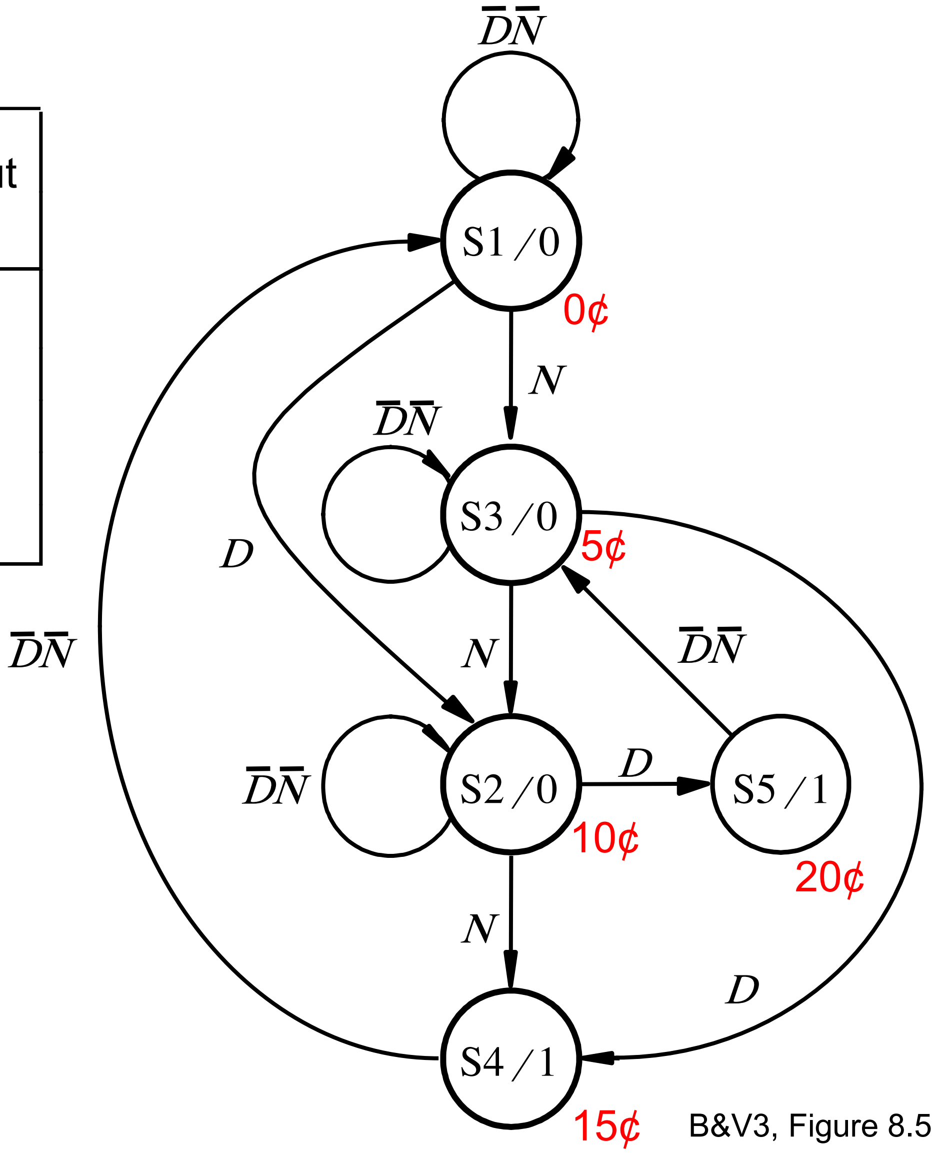
Minimized state table for Example 8.6

Present state	Next state				Output z
	DN =00	01	10	11	
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2	S4	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1

B&V3, Figure 8.56

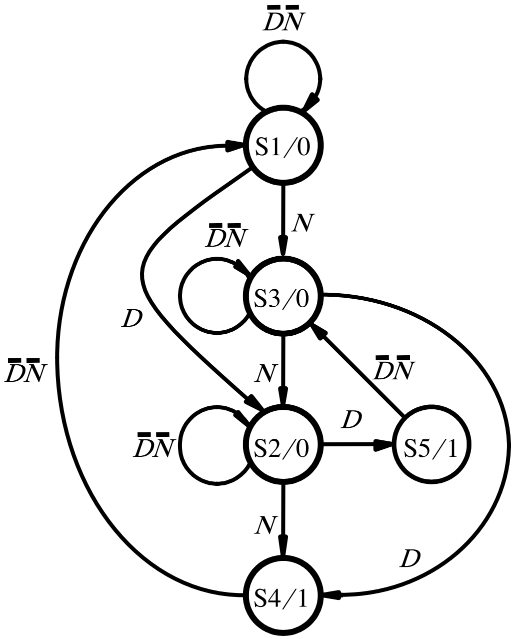
Minimized state diagram for Example 8.6

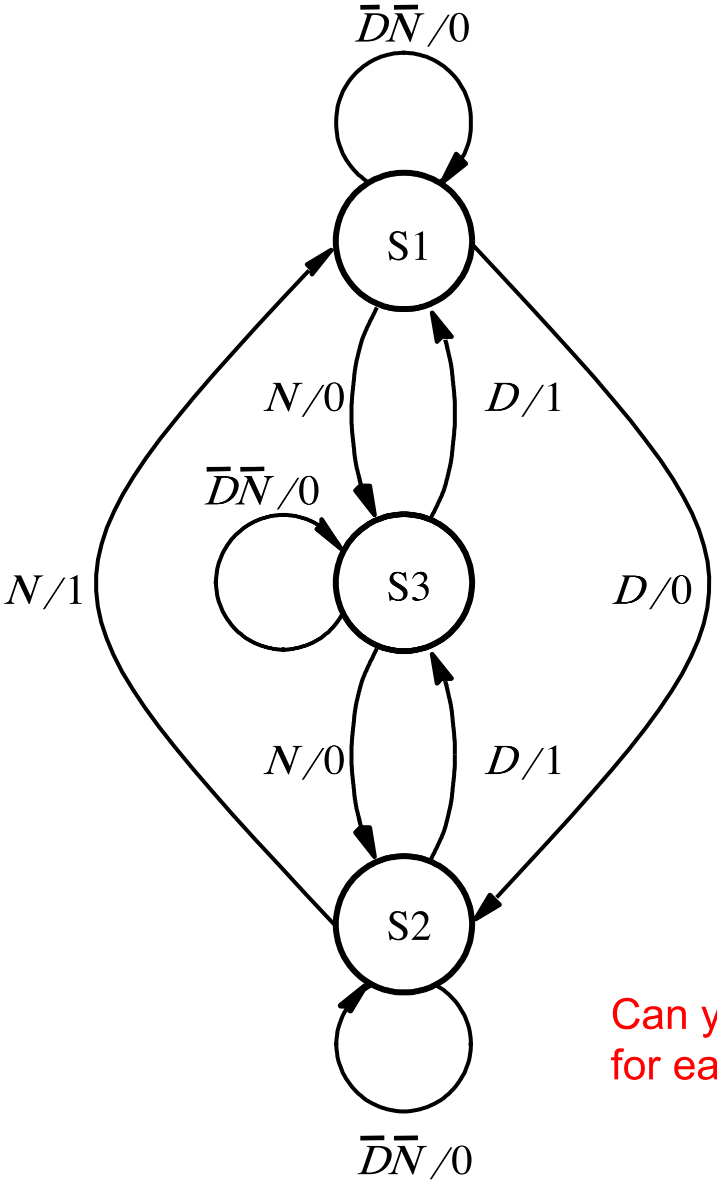
Present state	Next state				Output
	DN =00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2	S4	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1



B&V3, Figure 8.57

Mealy-type FSM for Example 8.6


$\overline{D}\overline{N}$
S1/0
N
$\overline{D}\overline{N}$
S3/0
D
$\overline{D}\overline{N}$
N
$\overline{D}\overline{N}$
$\overline{D}\overline{N}$
S2/0
D
S5/1
N
D
S4/1


$\overline{D}\overline{N}$ /0
S1
N/0
D/1
$\overline{D}\overline{N}$ /0
S3
N/1
D/0
N/0
D/1
S2
$\overline{D}\overline{N}$ /0

Can you derive the output expressions for each graph?

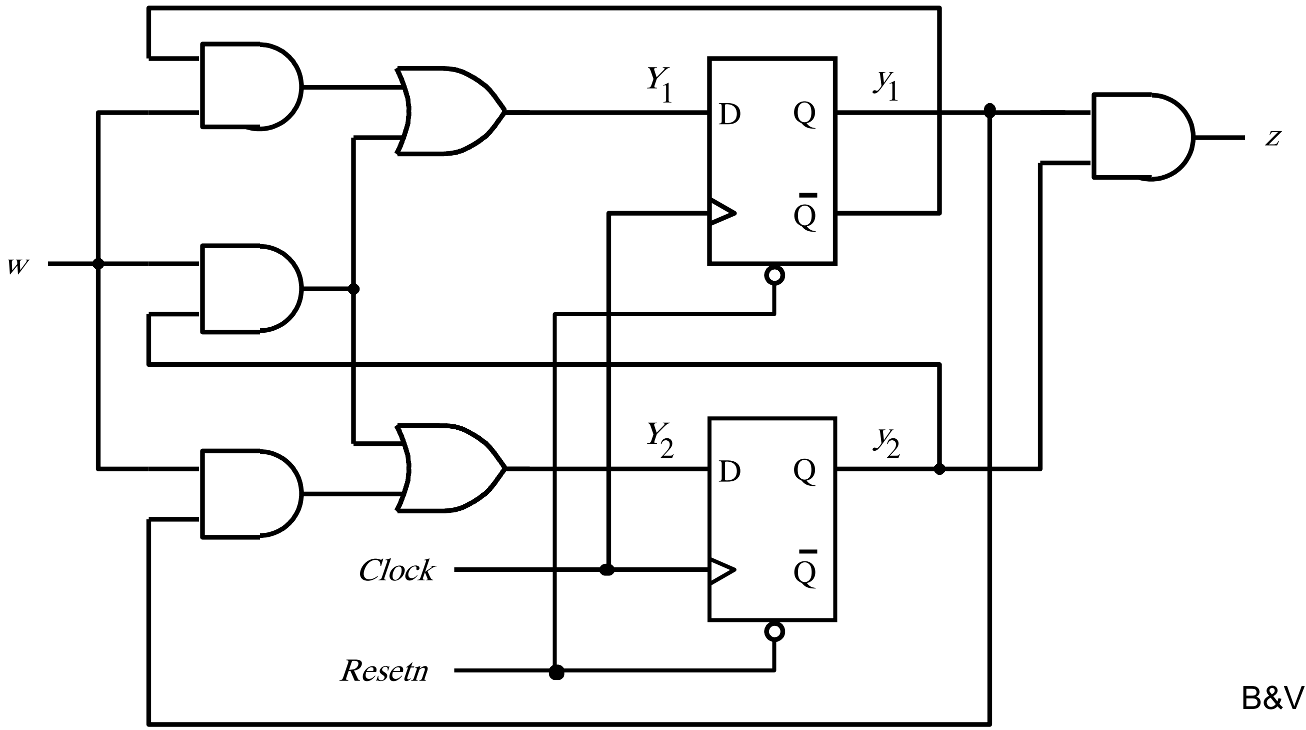
B&V3, Figure 8.58

Analysis of synchronous sequential circuits

- Designers must be able to analyze the behaviour of existing circuits – this is much easier than synthesizing them
  - To analyze a circuit, simply reverse the steps of the synthesis process
    1. FF outputs represent the present state variables
    2. Their inputs determine the next state the circuit will enter
    3. From this information we can construct the state-assigned table
    4. Which leads to the state table, and ultimately, the state diagram

Analysis example 8.8

- What is the function of this circuit?



B&V3, Figure 8.80

$$Y_1 = w\overline{y}_1 + wy_2$$

$$Y_2 = wy_1 + wy_2$$

$$z = y_1y_2$$

Tables for the circuit in Example 8.8

Present state	Next State		Output
	w = 0	w = 1	
y_2y_1	Y_2Y_1	Y_2Y_1	z
0 0	0 0	01	0
0 1	0 0	10	0
1 0	0 0	11	0
1 1	0 0	11	1

(a) State-assigned table

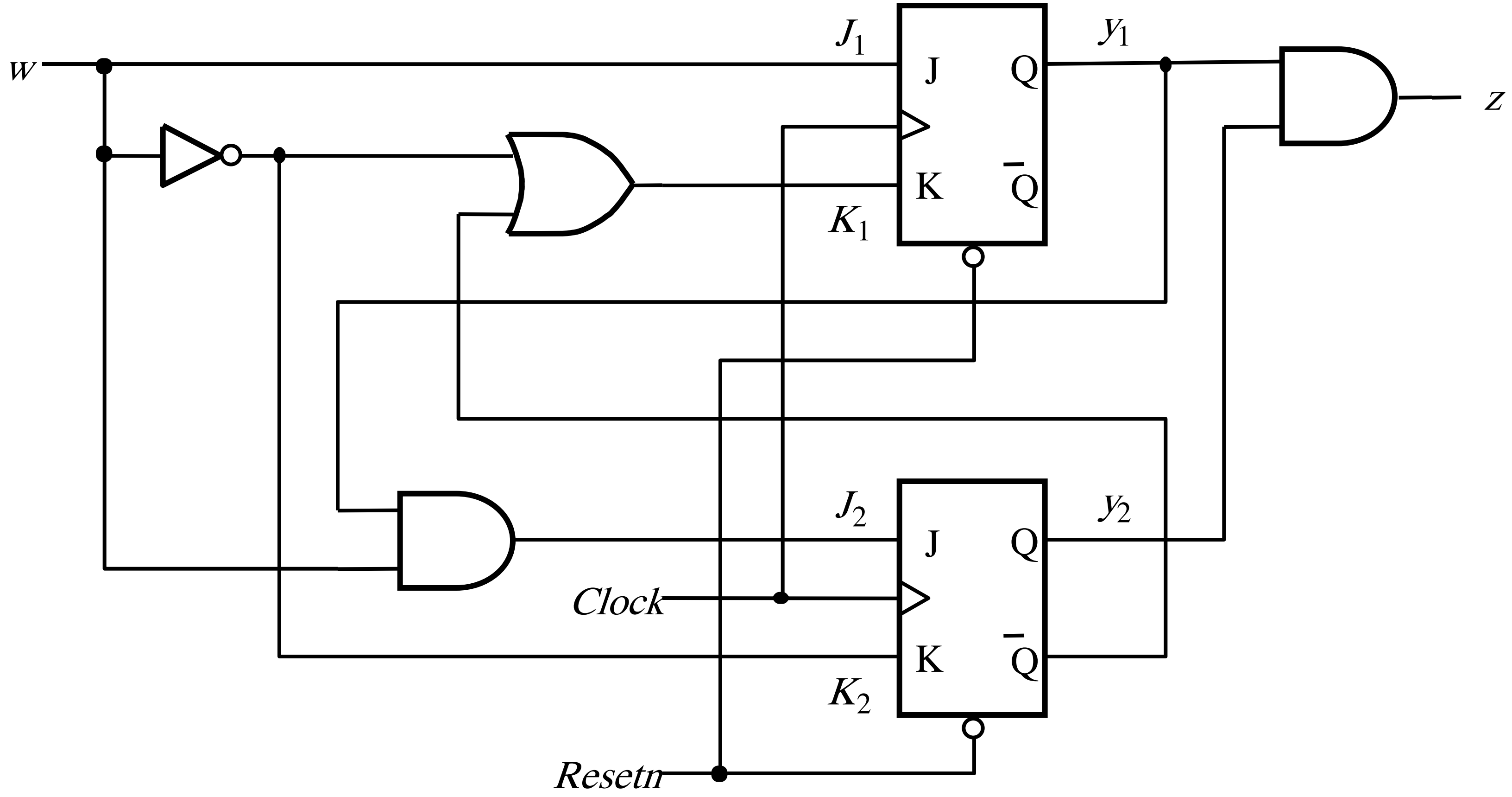
Present state	Next state		Output
	w = 0	w = 1	z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

(b) State table

B&V3, Figure 8.81

$$Y_1 = w\overline{y_1} + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

Example 8.9 using JK flip-flops



$J_1 = w$

$K_1 = \overline{w} + \overline{y_2}$

$J_2 = wy_1$

$K_2 = \overline{w}$

$z = y_2y_1$

B&V3, Figure 8.82

The excitation table for the circuit in L06/S82

$J_1 = w$

$K_1 = \overline{w} + \overline{y_2}$

$J_2 = wy_1$

$K_2 = \overline{w}$

$z = y_2y_1$

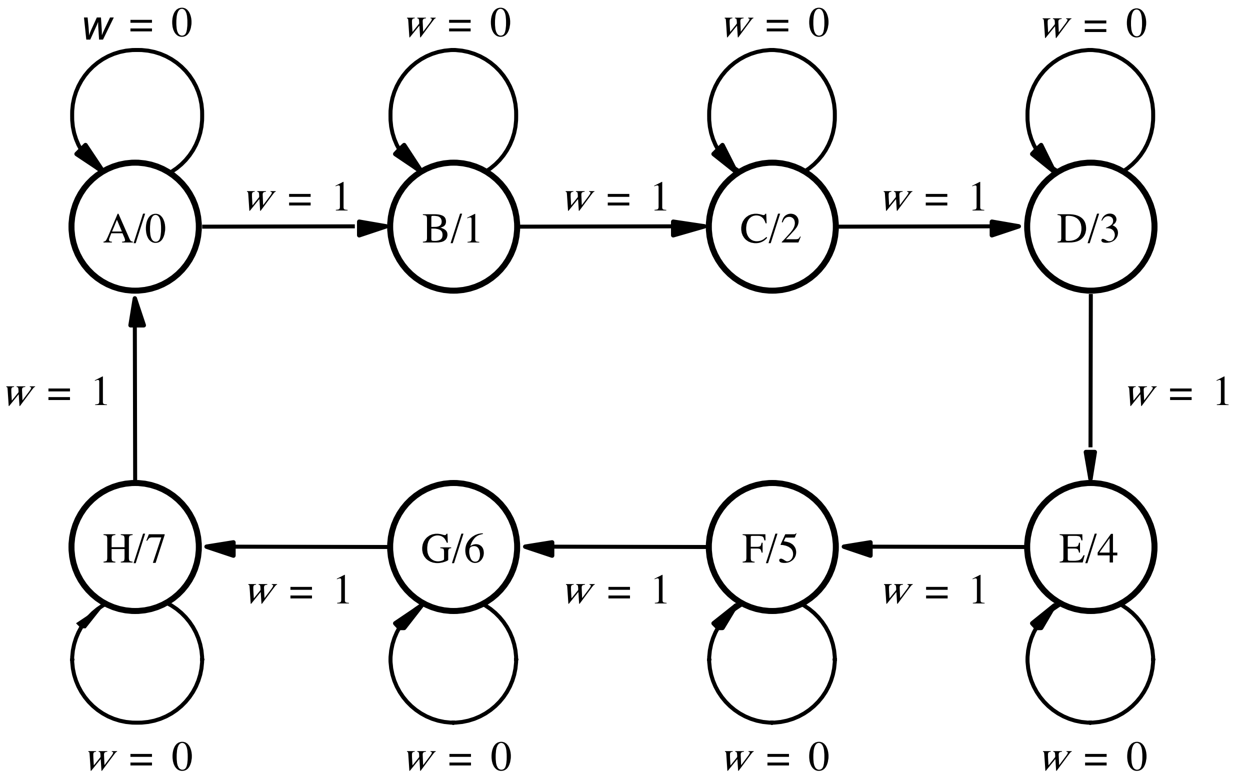
Present	Flip-flop inputs				Output
state	$w = 0$		$w = 1$		
y_2y_1	J_2K_2	J_1K_1	J_2K_2	J_1K_1	z
00	01	0 1	0 0	1 1	0
01	01	0 1	1 0	1 1	0
10	01	0 1	0 0	1 0	0
11	01	0 1	1 0	1 0	1

B&V3, Figure 8.83

Present	Next State		Output
state	w = 0	w = 1	
y_2y_1	Y_2Y_1	Y_2Y_1	z
0 0	0 0	01	0
0 1	0 0	10	0
1 0	0 0	11	0
1 1	0 0	11	1

Design of a counter using the sequential circuit approach

- Say we are to design a 0 – 7 up counter with enable



B&V3, Figure 8.60

State table for the counter

Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

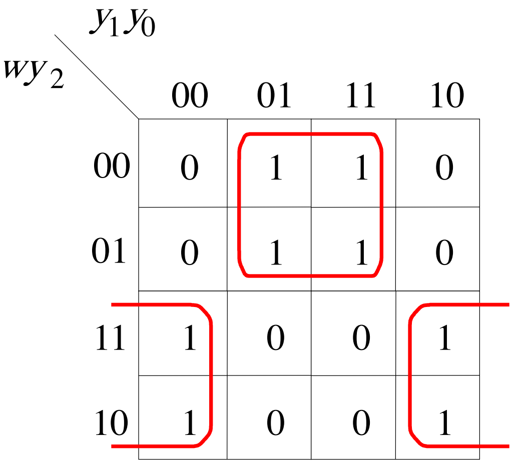
B&V3, Figure 8.61

State-assigned table for the counter

	Present state	Next state		Count
		$w = 0$	$w = 1$	
	$y_2y_1y_0$	$Y_2Y_1Y_0$	$Y_2Y_1Y_0$	$z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

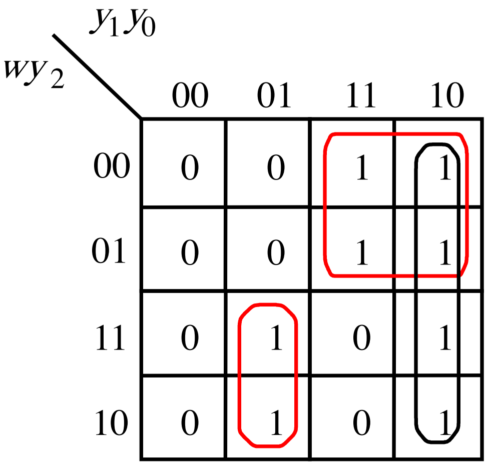
B&V3, Figure 8.62

K-maps for D flip-flops for the counter



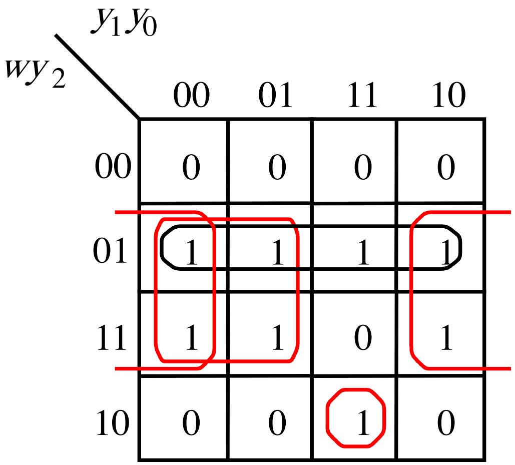
$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	0	0	1
10	1	0	0	1

$Y_0 = \bar{w}y_0 + w\bar{y}_0$



$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	1	0	1
10	0	1	0	1

$Y_1 = \bar{w}y_1 + y_1\bar{y}_0 + wy_0\bar{y}_1$

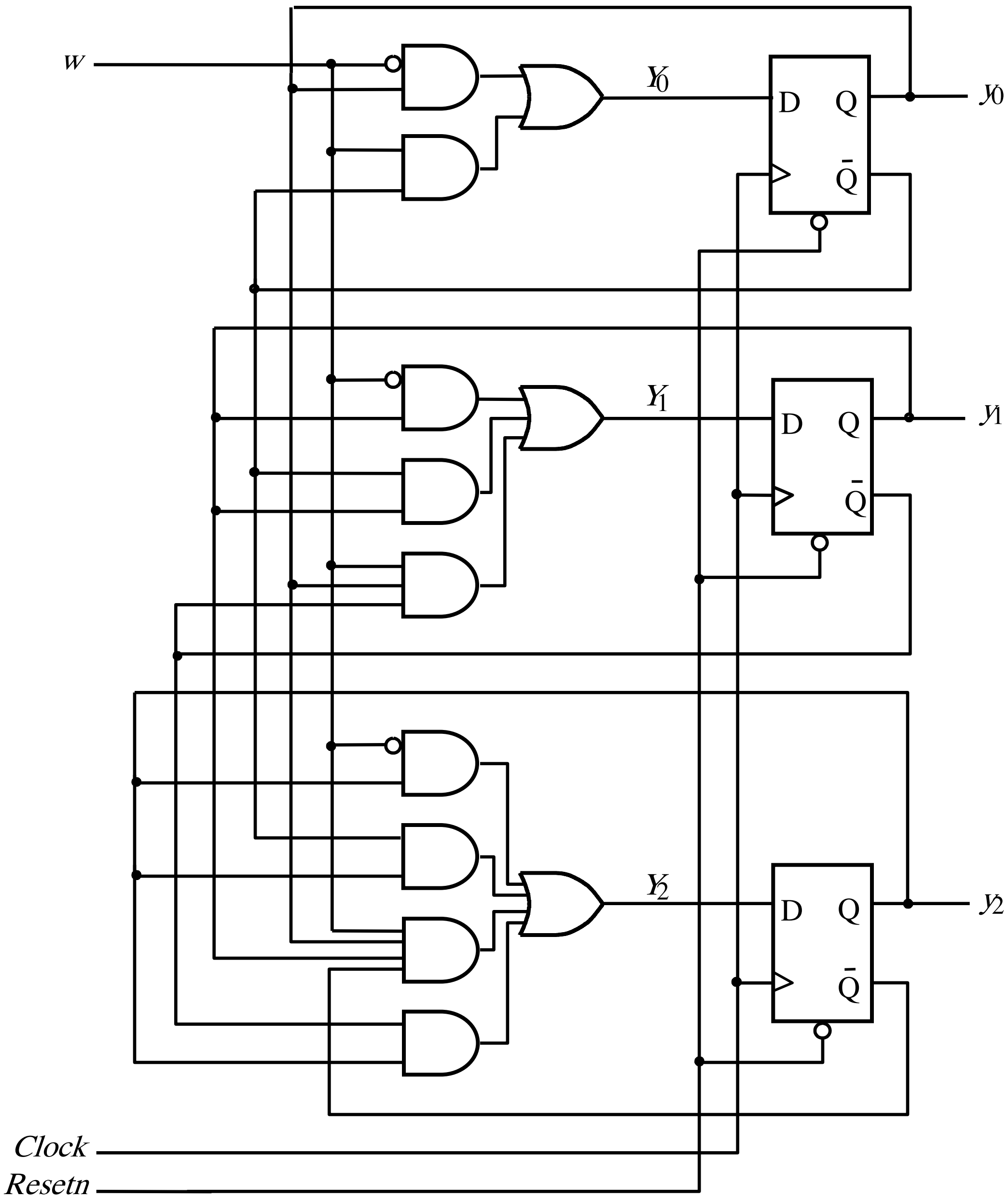


$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	0	1
10	0	0	1	0

$Y_2 = \bar{w}y_2 + \bar{y}_0y_2 + \bar{y}_1y_2 + wy_0y_1\bar{y}_2$

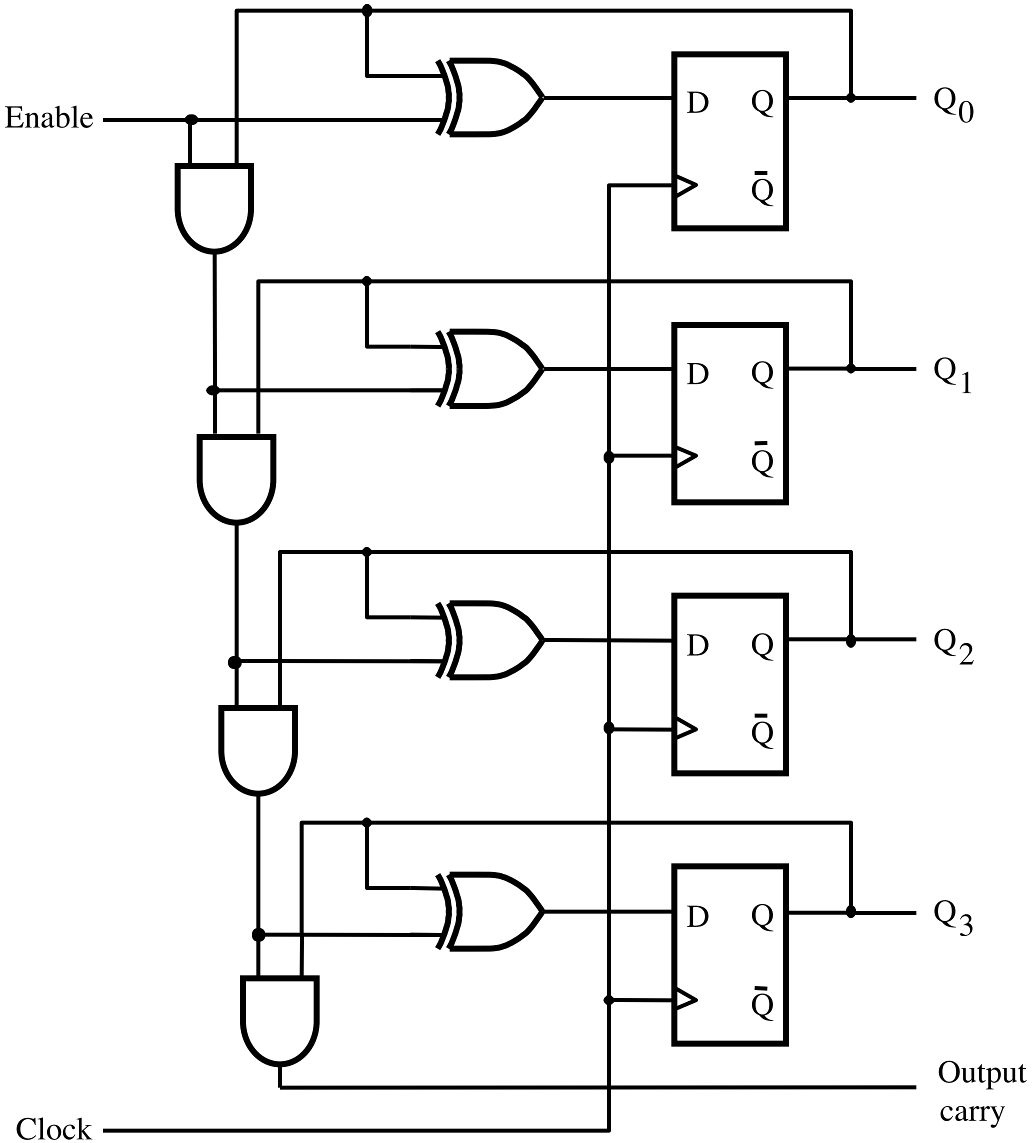
B&V3, Figure 8.63

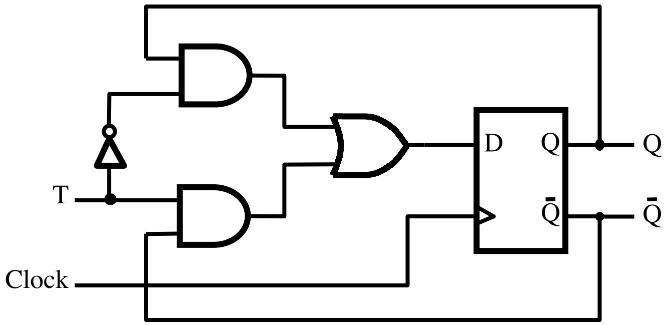
Circuit diagram for the counter implemented with D flip-flops



B&V3, Figure 8.64

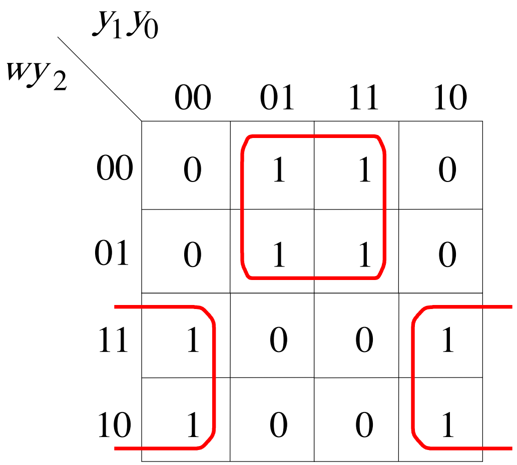
A four-bit counter with D flip-flops





T flip-flop

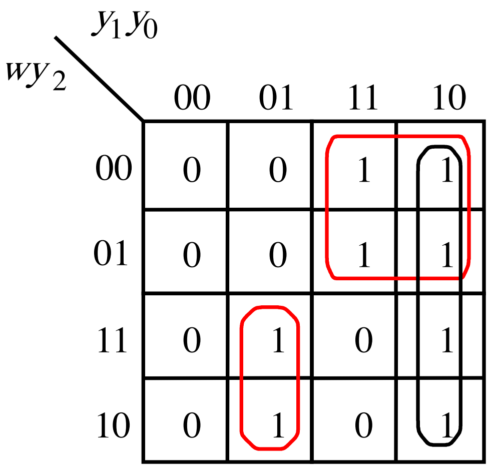
K-maps for D flip-flops for the counter



wy_2 \ y_1y_0	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	0	0	1
10	1	0	0	1

$Y_0 = \bar{w}y_0 + w\bar{y}_0$

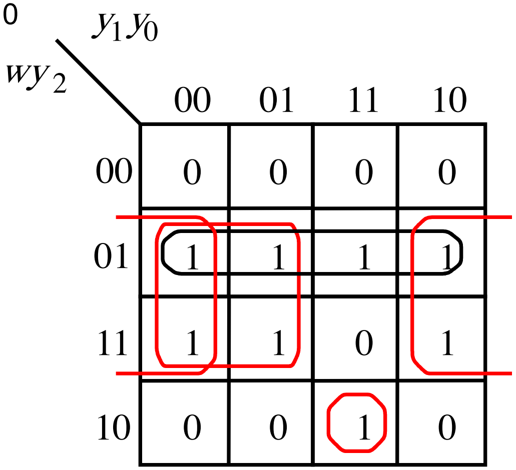
$D_0 = w \oplus y_0$



wy_2 \ y_1y_0	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	1	0	1
10	0	1	0	1

$Y_1 = \bar{w}y_1 + y_1\bar{y}_0 + wy_0\bar{y}_1$

$D_1 = wy_0 \oplus y_1$



wy_2 \ y_1y_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	0	1
10	0	0	1	0

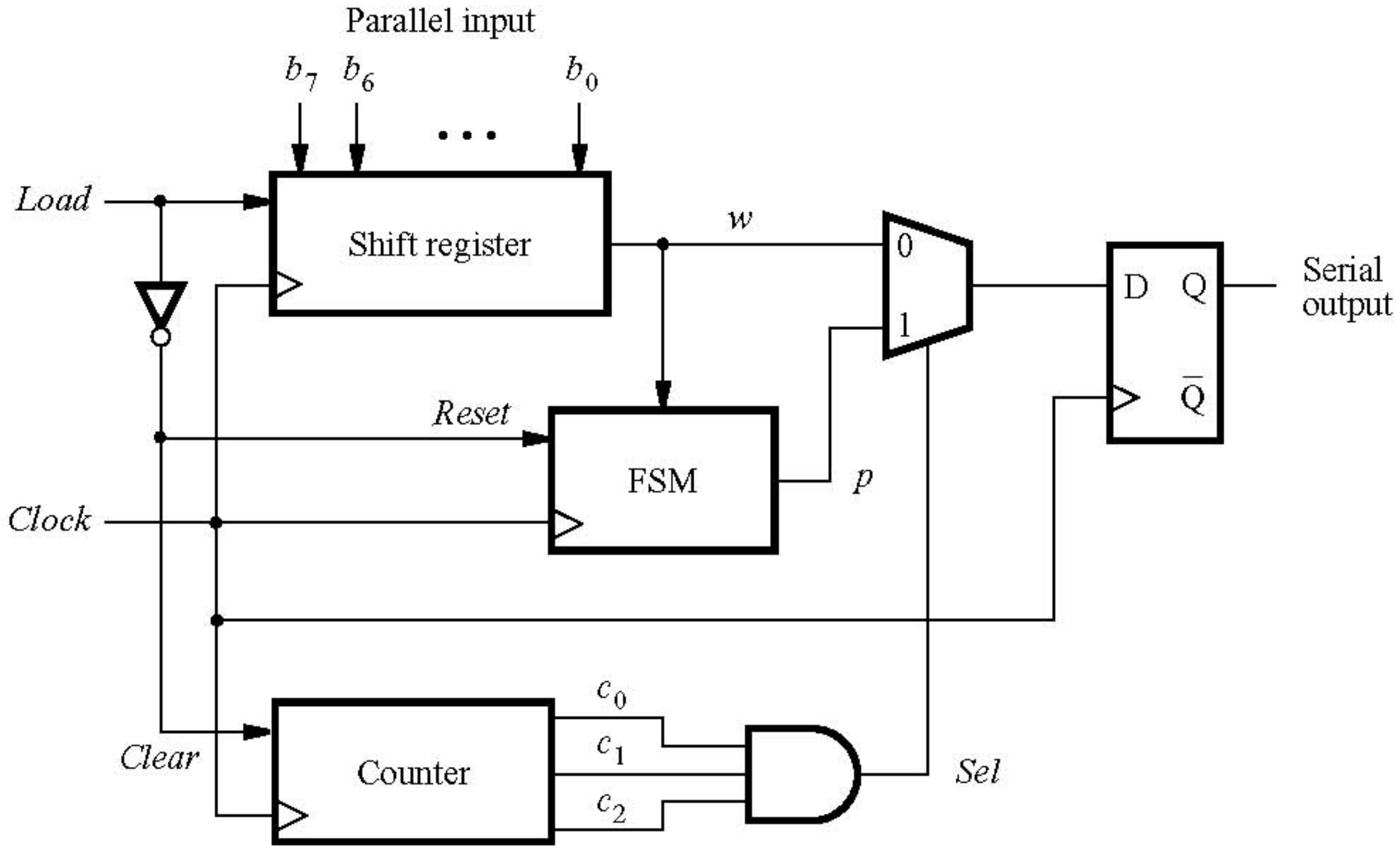
$Y_2 = \bar{w}y_2 + \bar{y}_0y_2 + \bar{y}_1y_2 + wy_0y_1\bar{y}_2$

$D_2 = wy_0y_1 \oplus y_2$

B&V3, Figure 8.63

Design Exercise: Parity generator for serial communication

- Design an even parity generator to produce parity bit *p* to replace $b_7 = 0$ of each ASCII byte *B* that is to be serially transmitted by the system below


Parallel input
b_7
b_6
b_0
Load
Shift register
w
0
1
D
Q
$\overline{Q}$
Serial output
Reset
FSM
p
Clock
Clear
Counter
c_0
c_1
c_2
Sel