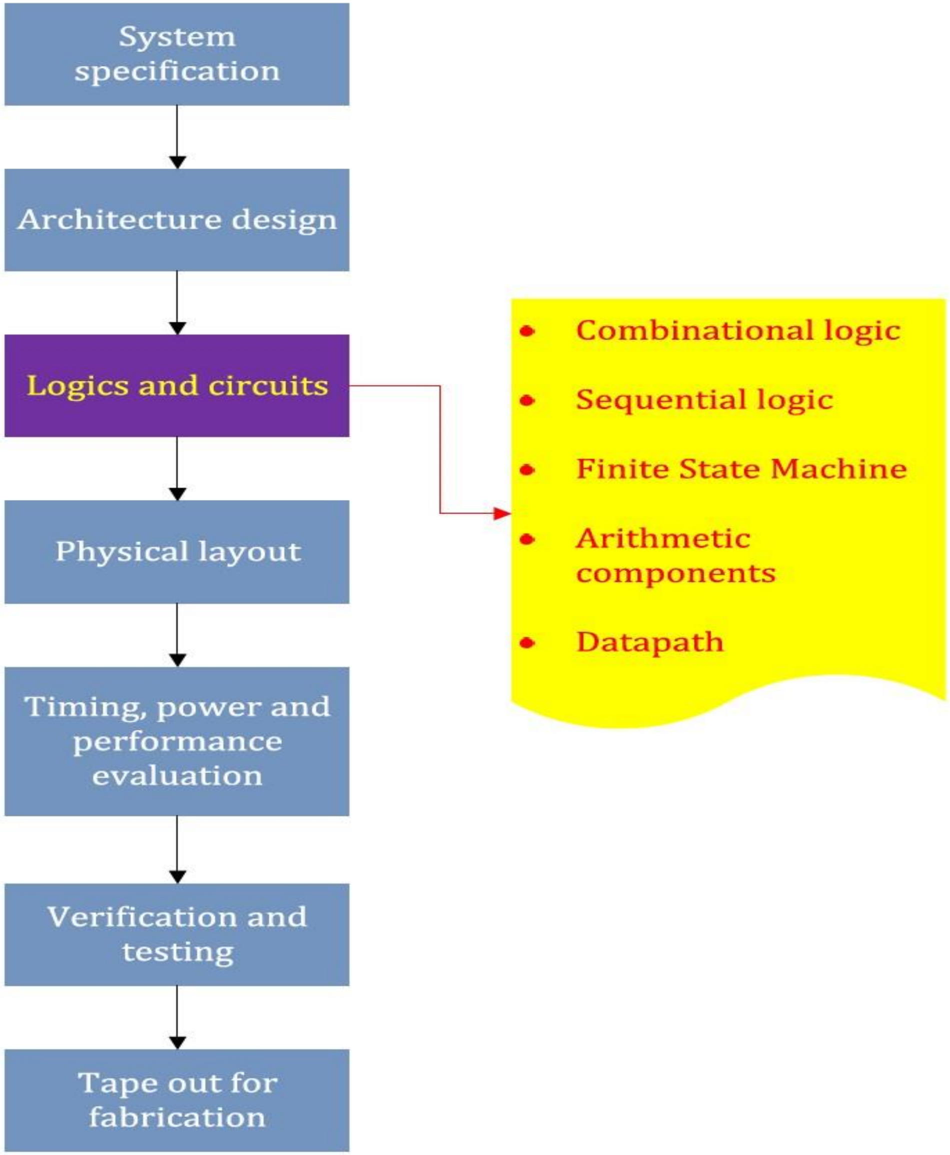
Digital Logic Design

- Agenda of Lecture (Recap of)
  - Combinational logics
  - Sequential logics
  - Finite state machine
  - Arithmetic circuits
  - Datapath



System specification

Architecture design

Logics and circuits

Physical layout

Timing, power and performance evaluation

Verification and testing

Tape out for fabrication

- Combinational logic
- Sequential logic
- Finite State Machine
- Arithmetic components
- Datapath

Figure 5-1 Logic/circuit design phase in the design flow

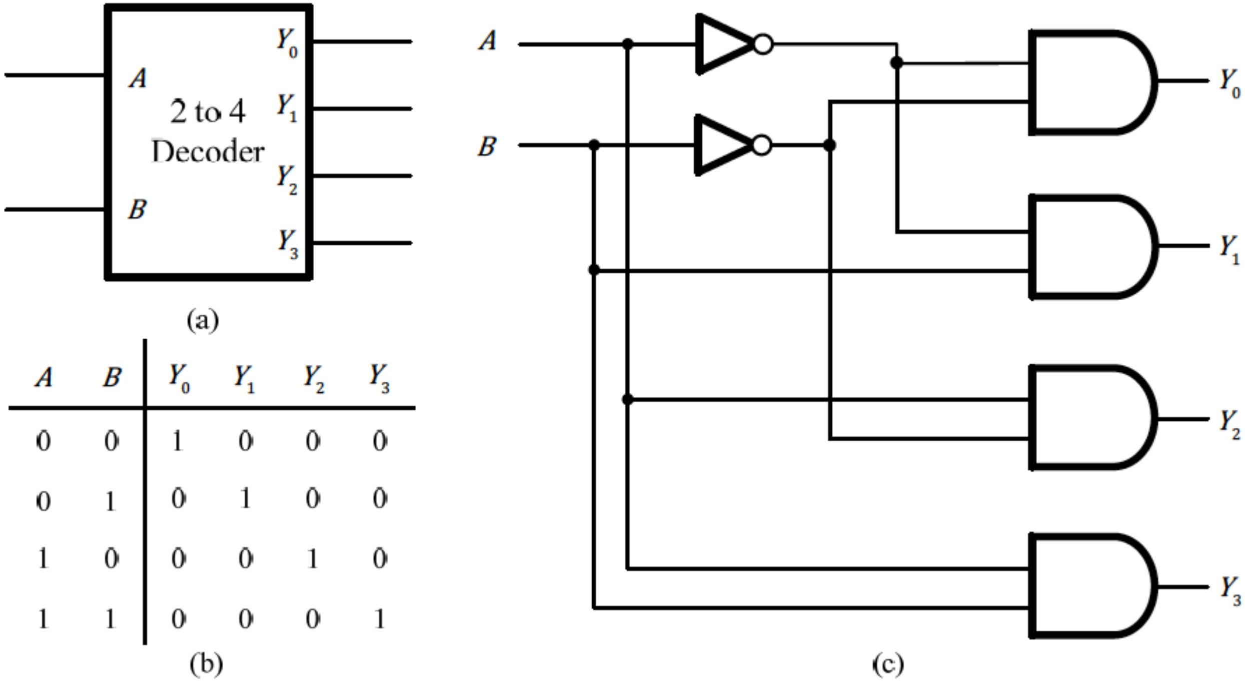
- Digital logics can be divided into two main classes: combinational and sequential logics.
- Combinational logic usually implements a Boolean expression, where the output is purely a function of the present input.
- In contrast to combinational logic, the typical feature of sequential logic is its memory mechanism, which can store previous logic values, also known as states.
  - The storage elements are commonly implemented by flip-flop or latch.
  - Thus the output of a sequential circuit is a combined function of the present input and the state of the circuit.

Combinational Logics

- Combinational logic circuits are composed of gates or inverters to execute a particular function and may have one or more outputs.
- They play a significant role as function blocks in the digital circuit, especially, arithmetic logic circuit (ALU) where mathematical calculations are performed like addition, subtraction and multiplication.
- The basic combinational logic circuits include, gate, decoder, encoder and multiplexer.
- Some other complex combinational building blocks like adder and multiplier are used in ALU unit.

Decoder

In an n-input decoder, one of 2^n possible output lines will be selected at a time. Each output is mapped to one possible logic combination of input value. Figure 5-2 shows a 2 to 4 decoder consisting of four AND gates and two inverters. A and B are the inputs and Y_0 to Y_3 are the outputs, respectively. In a decoder, only one output will be set to be true according to the specific input combination at the time. For example, when $AB = 00$, Y_0 will become 1 and will be selected as the current output. This decoder is formally represented in the following figure.



(a)

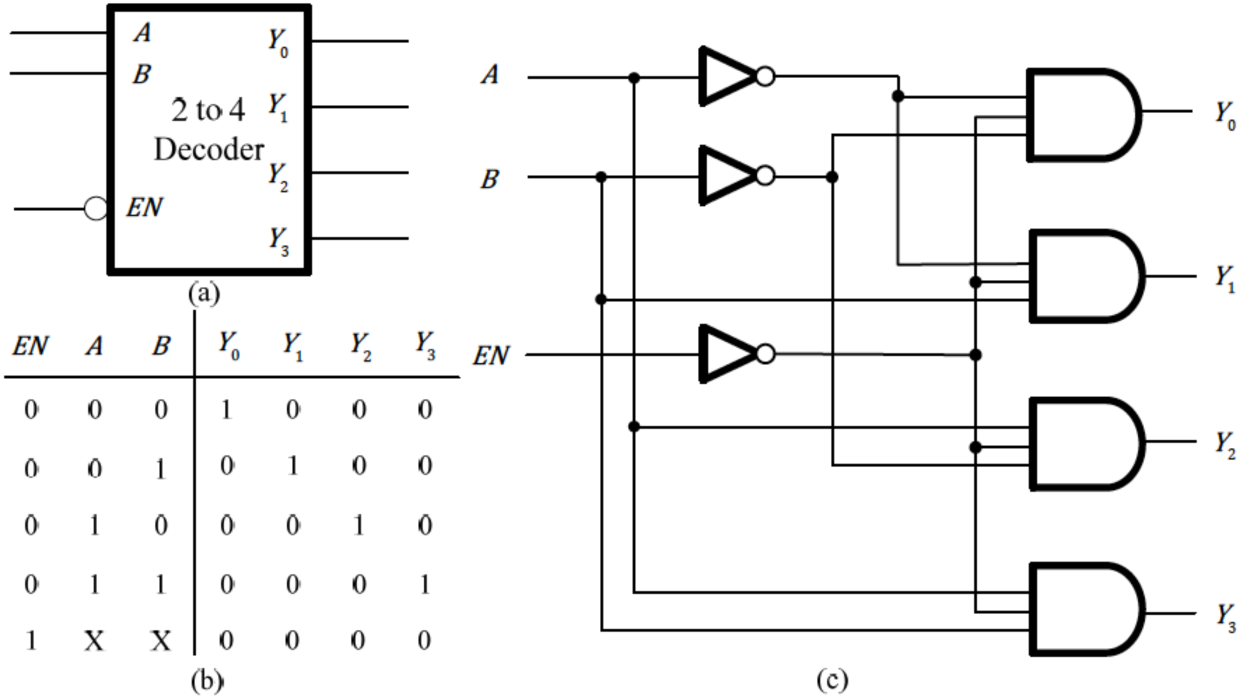
A	B	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(b)

(c)

Figure 5-2 A 2 to 4 decoder: (a) graphic symbol, (b) truth table, and (c) circuit implementation

A decoder with enable control capability



EN	*A*	*B*	Y_0	Y_1	Y_2	Y_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

Figure 5-3 A 2 to 4 decoder with an enable input signal: (a) graphic symbol, (b) truth table, and (c) circuit implementation

- ***Structure of large decoder***

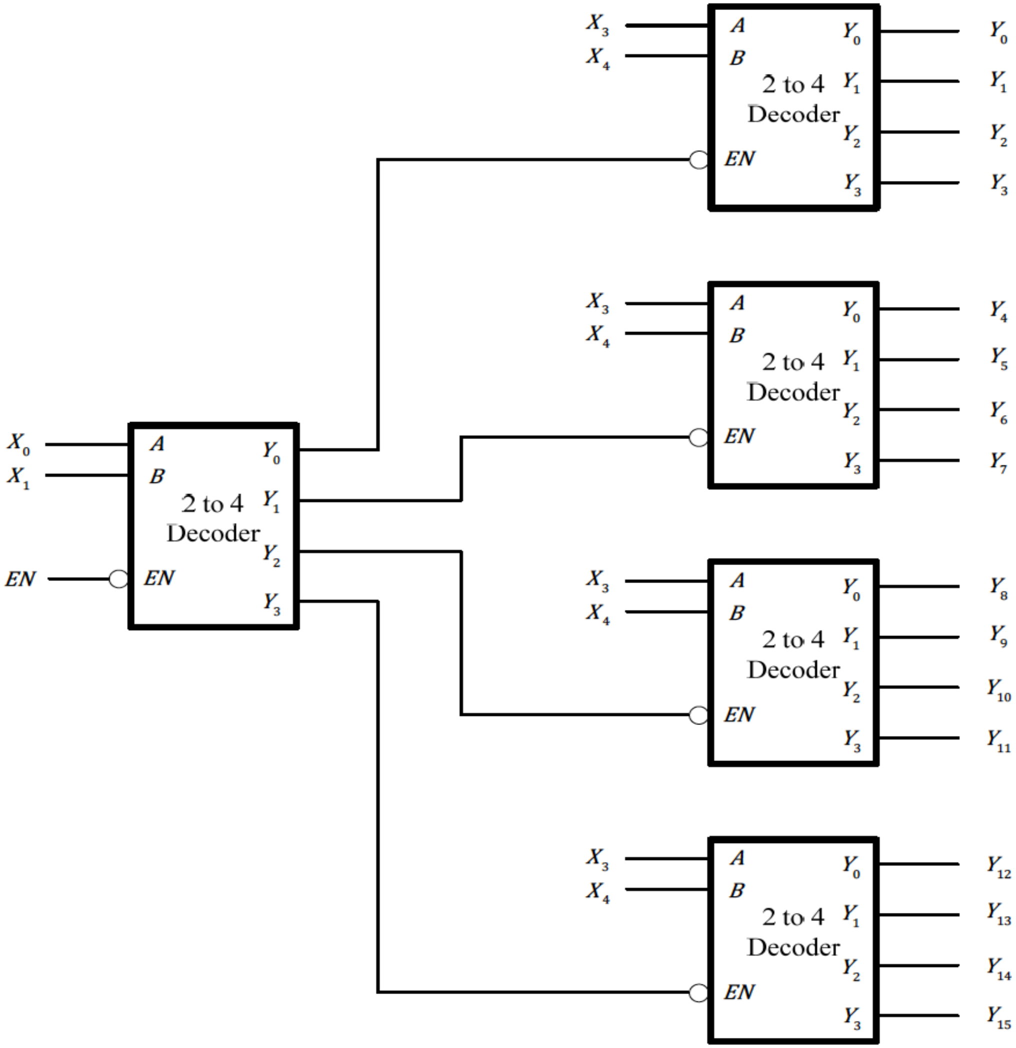


Figure 5-4 A 4-to-16 decoder

- ***Decoder is used as address selection unit in memory design***

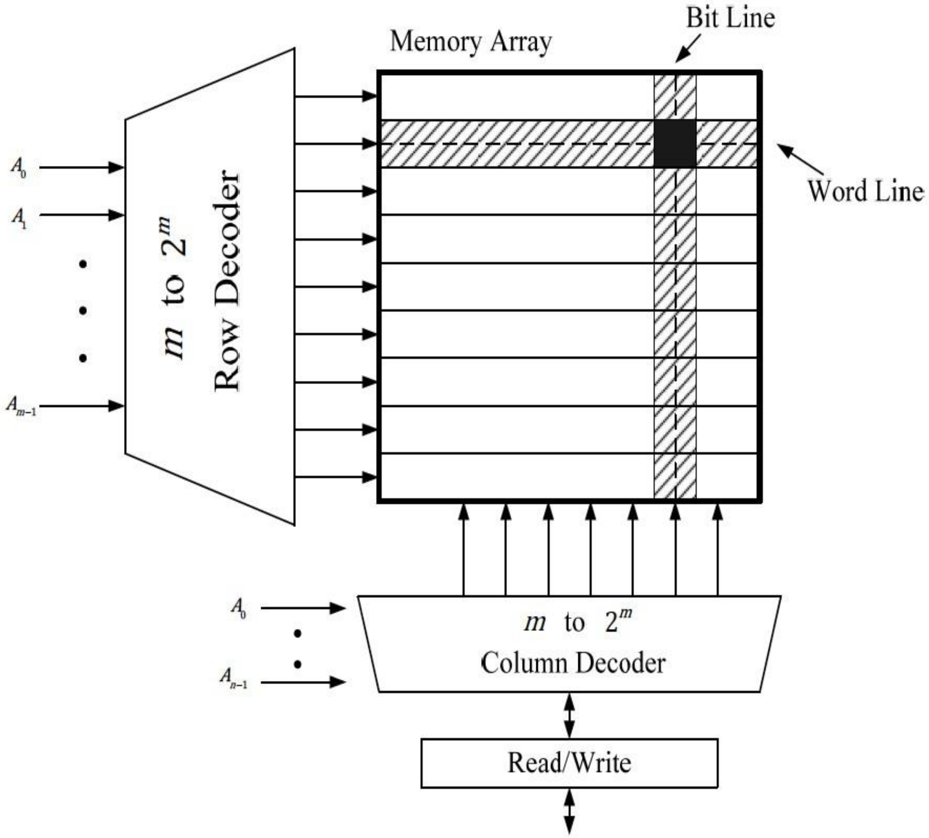
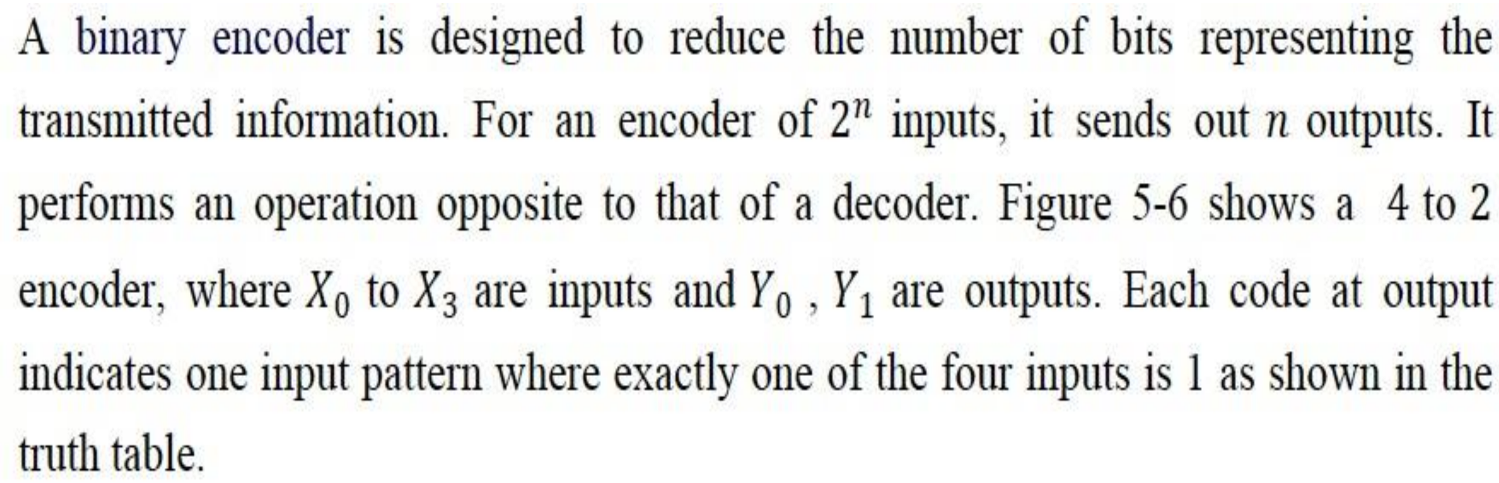


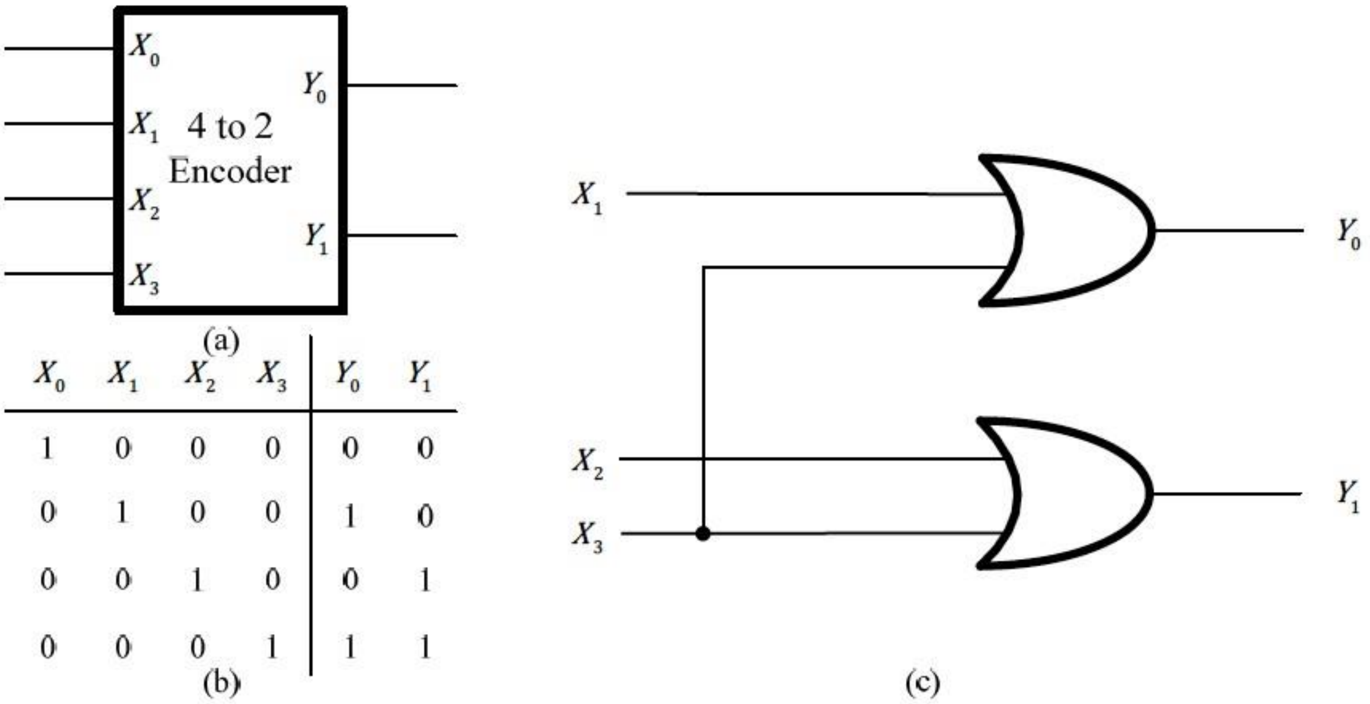
Figure 5-5 Application of decoder in SRAM

Encoder



A binary encoder is designed to reduce the number of bits representing the transmitted information. For an encoder of 2^n inputs, it sends out n outputs. It performs an operation opposite to that of a decoder. Figure 5-6 shows a 4 to 2 encoder, where X_0 to X_3 are inputs and Y_0 , Y_1 are outputs. Each code at output indicates one input pattern where exactly one of the four inputs is 1 as shown in the truth table.

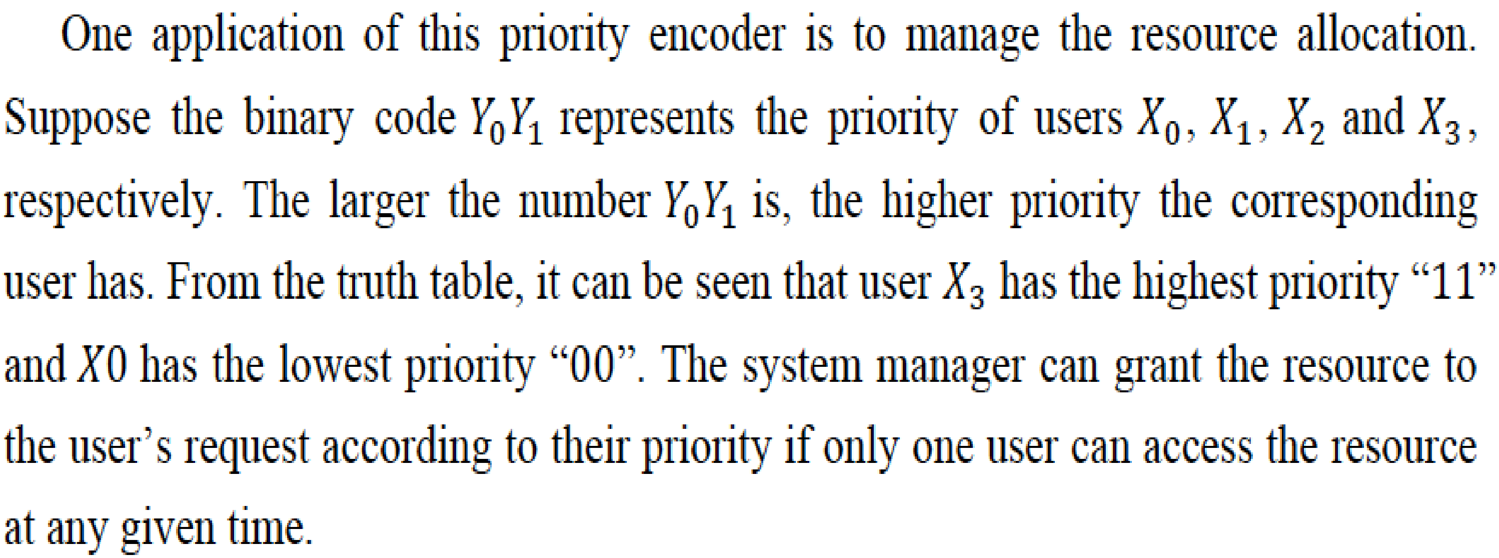
An encoder circuit



X_0	X_1	X_2	X_3	Y_0	Y_1
1	0	0	0	0	0
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	1	1	1

Figure 5-6 A 4 to 2 binary encoder: (a) graphic symbol, (b) truth table, and (c) circuit implementation

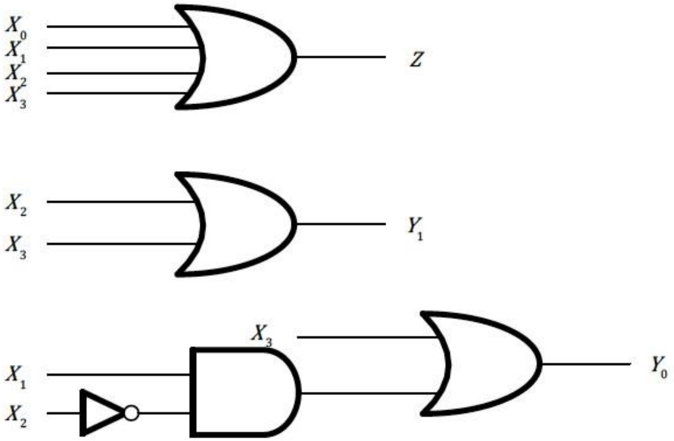
Application: priority encoder



One application of this priority encoder is to manage the resource allocation. Suppose the binary code Y_0Y_1 represents the priority of users X_0, X_1, X_2 and X_3, respectively. The larger the number Y_0Y_1 is, the higher priority the corresponding user has. From the truth table, it can be seen that user X_3 has the highest priority “11” and $X0$ has the lowest priority “00”. The system manager can grant the resource to the user’s request according to their priority if only one user can access the resource at any given time.

X_0	X_1	X_2	X_3	Y_0	Y_1	Z
1	0	0	0	0	0	1
X	1	0	0	1	0	1
X	X	1	0	0	1	1
X	X	X	1	1	1	1
0	0	0	0	0	0	0

(a)

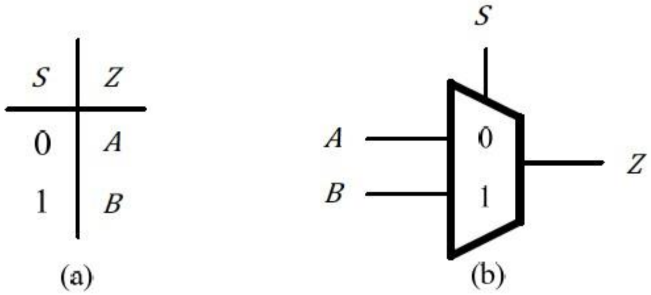


(b)

Figure 5-7 A priority encoder: (a) truth table, and (b) circuit implementation

Multiplexer

A multiplexer is a combinational digital block used for signal selection. Basically, a multiplexer has multiple inputs and only one output. Only one of the input signals will be selected out each time. One or more control signals are required to indicate which input is selected. Figure 5-8 shows the graphic symbol and truth table of a 2 to 1 multiplexer. *S* is the selection signal. *A*, *B* are the data input and *Z* is the output. Whenever a selection signal is inserted, either *A* or *B* will be chosen and sent out as the output.



S	*Z*
0	*A*
1	*B*

Figure 5-8 A 2 to 1 multiplexer: (a) truth table, and (b) logic symbol

- *Multiplexer circuit*

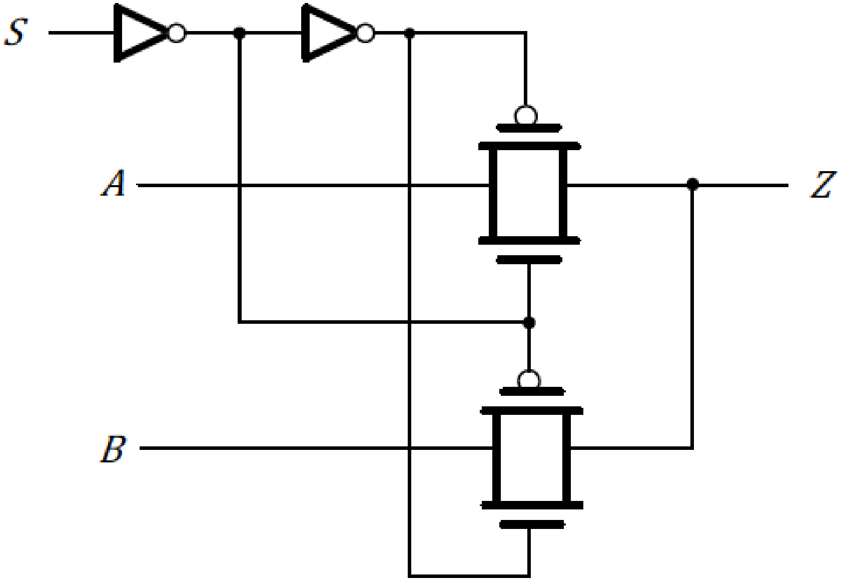


Figure 5-9 Transistor level structure of a multiplexer

- *Gate level structure*

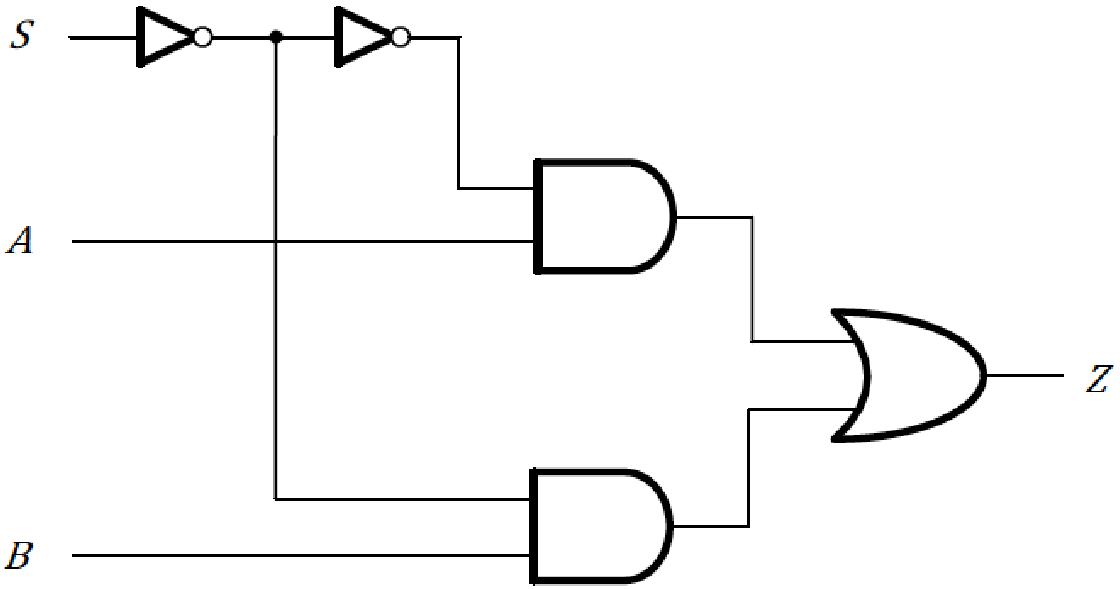
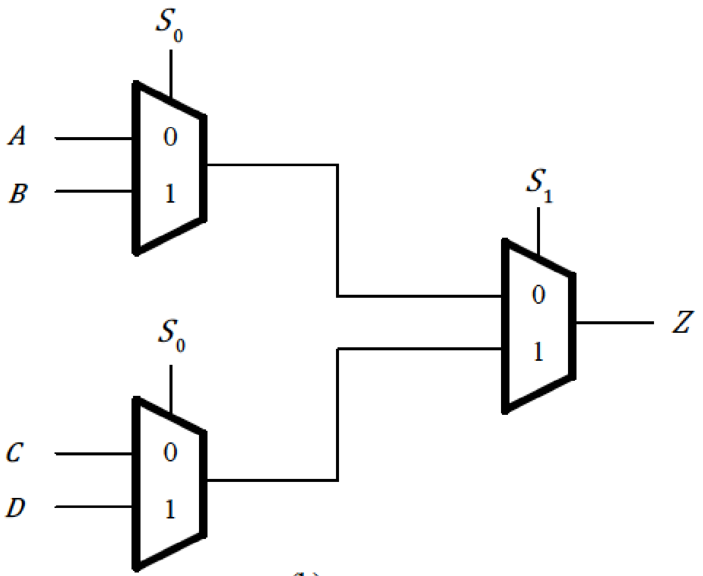


Figure 5-10 Gate level structure of a multiplexer

- ***A 4 to 1 multiplexer***

S_0	S_1	Z
0	0	A
0	1	C
1	0	B
1	1	D

(a)



(b)

Figure 5-11 A 4 to 1 multiplexer: (a) truth table, and (b) circuit structure

Arithmetic Logic Blocks

- Computation in a digital system is carried out by arithmetic logic blocks.
- There are basically four arithmetic operations: addition, subtraction, multiplication and division.
  - All of them are executed by using the fundamental gates.
- Their performance efficiency is critical in a digital system because they are used “heavily” for almost for all applications.
- In the following, we will discuss the basic operation principles and different topologies of two most significant arithmetic combinational blocks:
  - Adder and Multiplier.

Adder

- *Circuit implementation of a full adder*

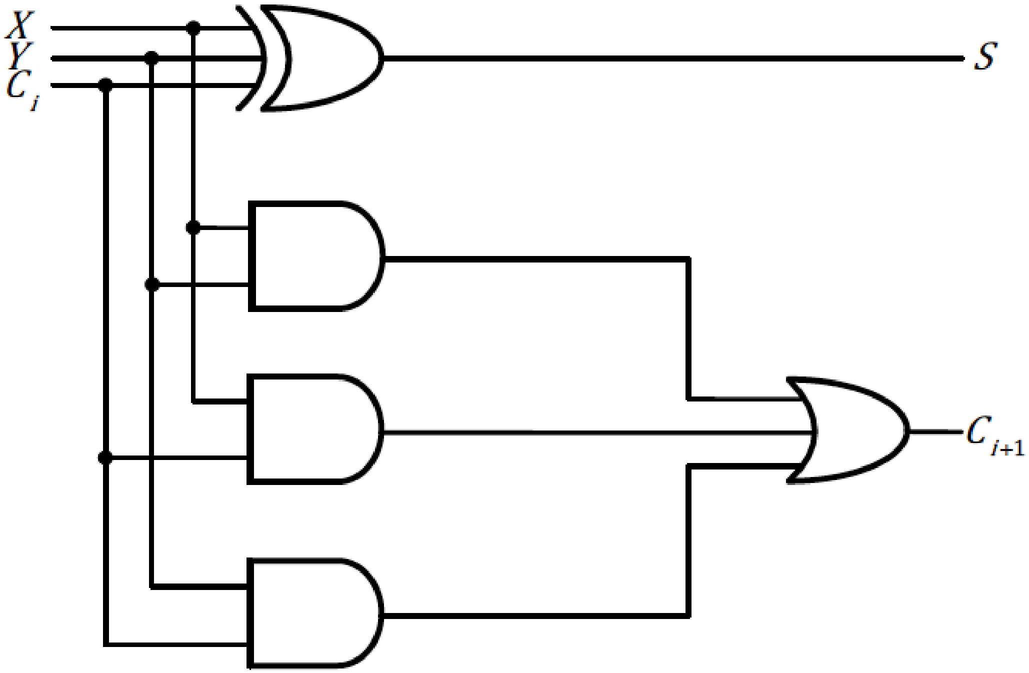


Figure 5-16 Circuit implementation of a full adder

- ***Ripple Carry Adder***

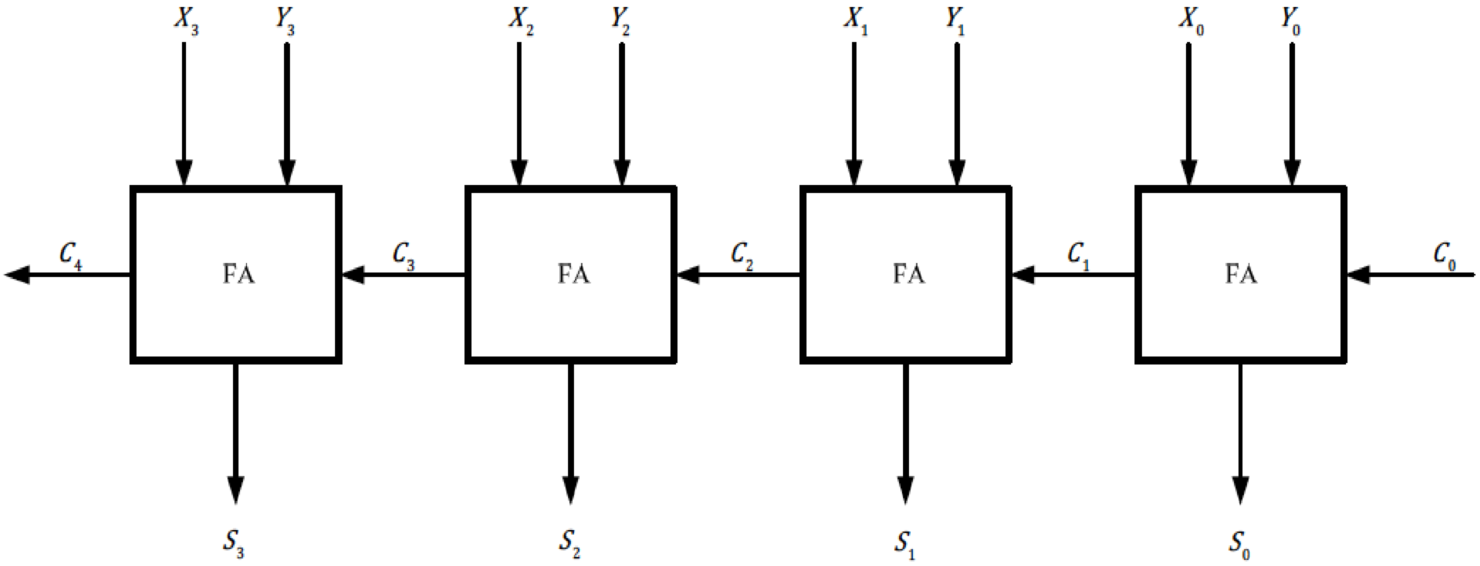


Figure 5-18 Circuit structure of a ripple carry adder

- ***Circuit implementation***

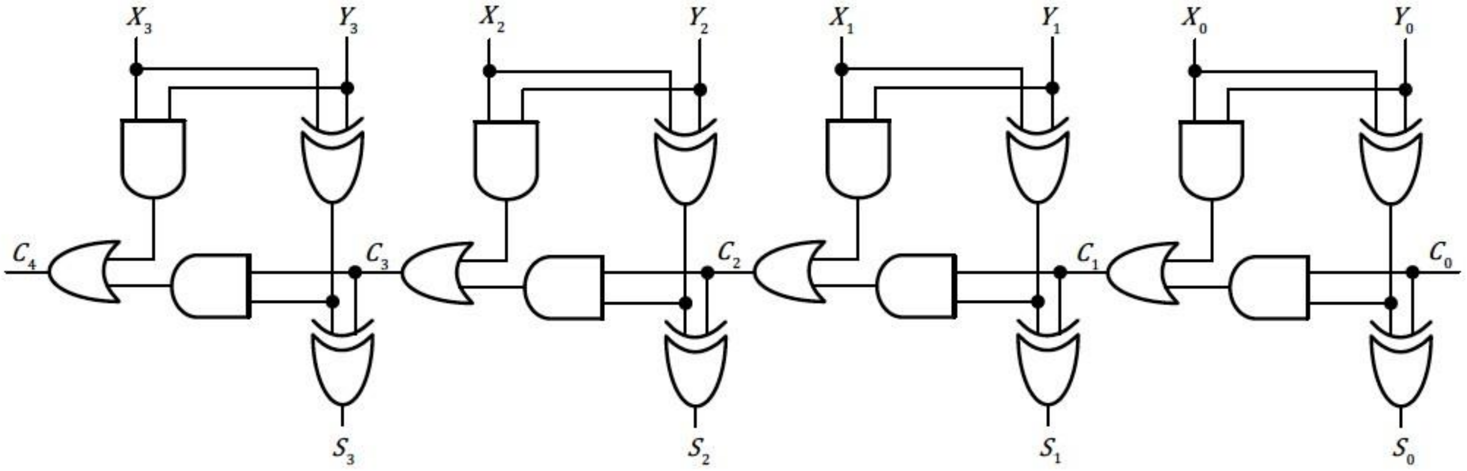


Figure 5-19 Circuit implementation of a 4-bit ripple carry adder

- ***Carry look-ahead adder***

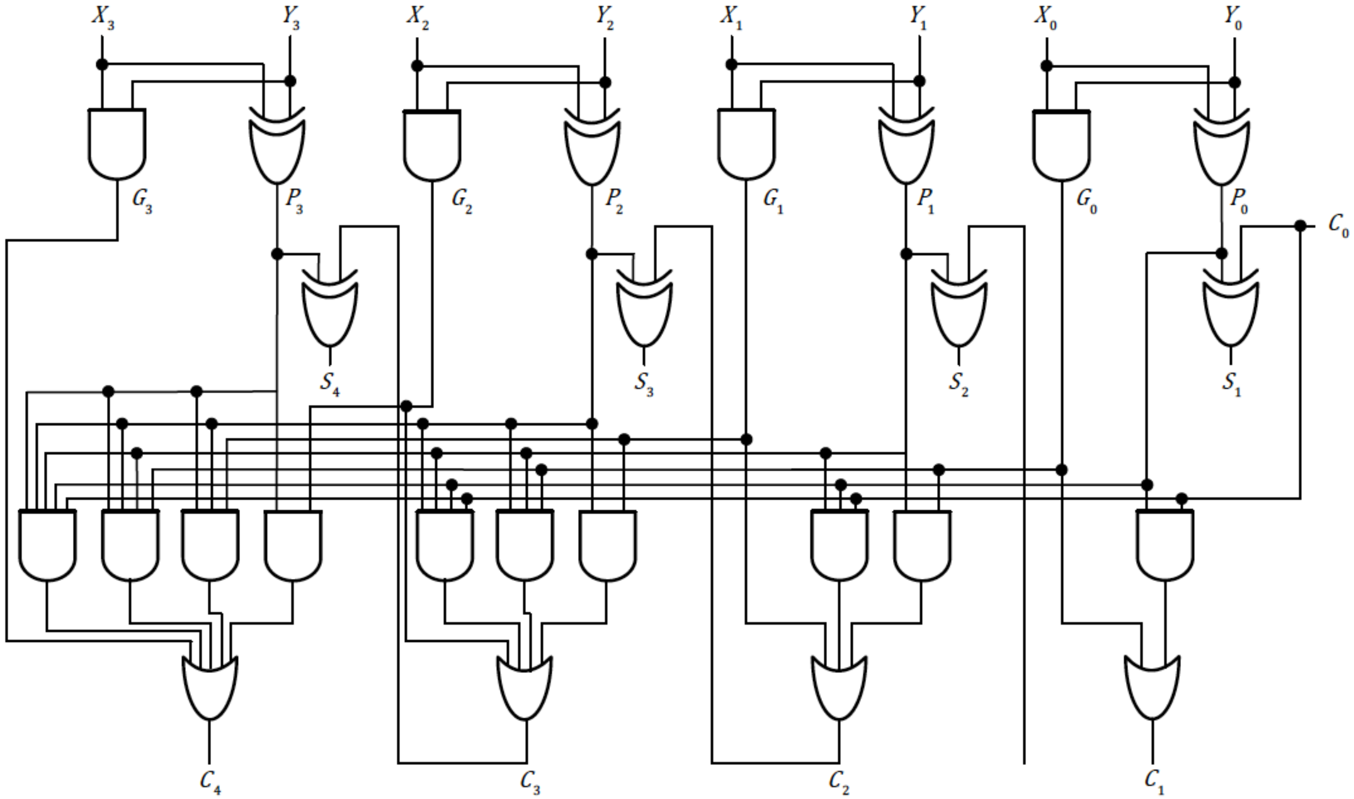


Figure 5-20 The circuit implementation of a 4-bit carry look-ahead adder

Addition/Subtraction

In digital circuit design, subtraction can be performed by adding two operands with one of their value inverted. Thus, it is easy to combine these two operations, addition and subtraction, together in one particular block by adding some peripheral signal and circuits. One implementation is shown in Figure 5-24. A ripple carry adder is reconfigured to perform either addition or subtraction by adding a multiplex and an inverter. If $C_0 =(\overline{Add})\ /Sub$ equals 0, the whole circuit works as a normal adder. If $C_0 = (\overline{Add})\ /Sub$ equals 1, one of the operands will be inverted and sent into the adder which then essentially operates subtraction. At the same time, the value of C_0 which may be either 1 or 0 will be added into the final result to perform the $2's$ complement.

- ***Circuit structure***

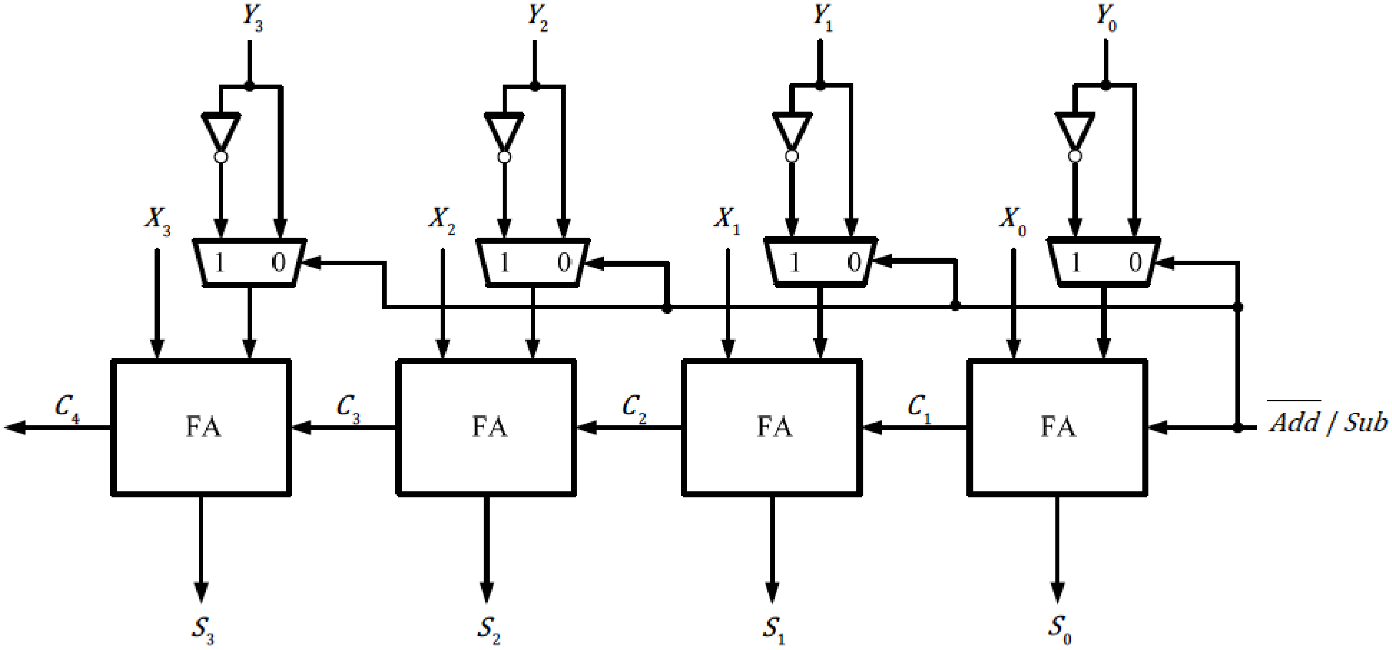


Figure 5-24 Circuit structure of addition/subtraction block

- *An alternative circuit structure*

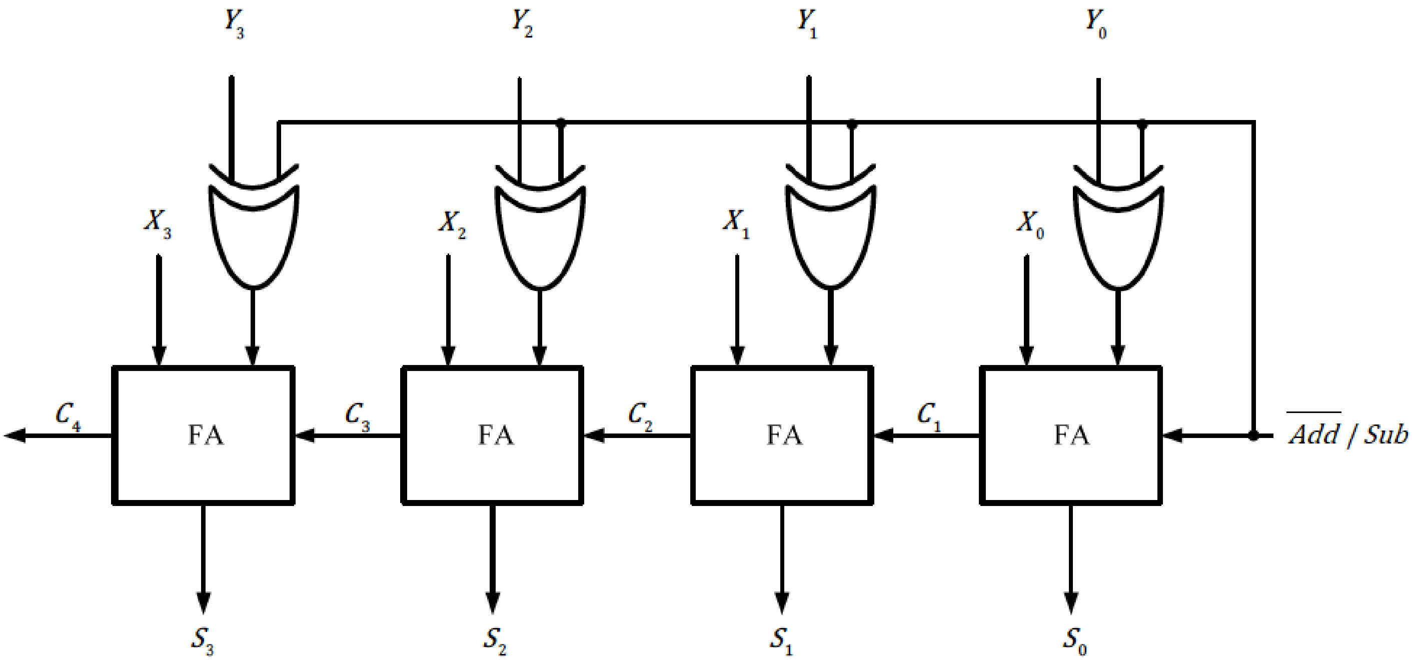
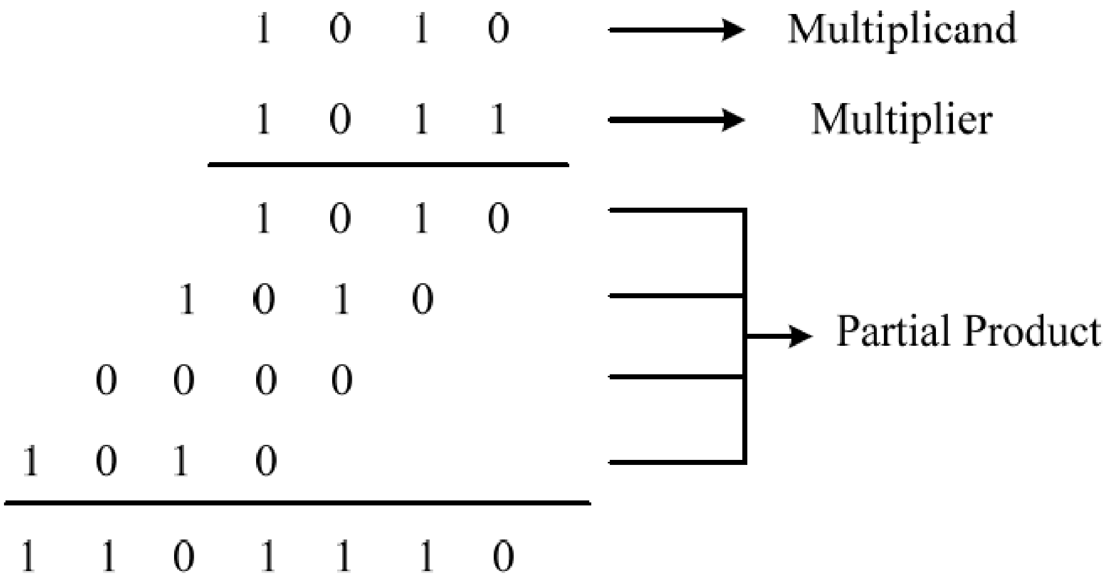


Figure 5-25 An alternative circuit structure for addition/subtraction block

Multiplier

- Multiplication is a complex operation in ALU which usually consumes large hardware resources and has a large propagation delay.
- The multiplier, in nature, is an arrangement of adder array (suppose we only use combinational circuit).
- Before digging into the details of the multiplier, let us look at the binary multiplication procedure.
  - Suppose we have two four-bits binary number X and Y. A simple multiplication is performed by generating a set of partial products and summing them to get the final result. Each partial product is produced by multiplying X with each bit of Y and shifting the set of partial products with regard to the bit position of Y.

- ***Figure 5-26 illustrates the multiplication procedure of two 4- bits binary numbers.***



```
            1 0 1 0
            1 0 1 1
            -------
            1 0 1 0
          1 0 1 0
        0 0 0 0
      1 0 1 0
      -------------
      1 1 0 1 1 1 0
```

Figure 5-26 Multiplication

From the above graph, the multiplier should be designed to efficiently realize the function of generating and summing the partial products. Generally, the partial product generation can be achieved by simply applying a 2-input AND gate. Figure 5-27 is an example of how a line of partial products is generated.

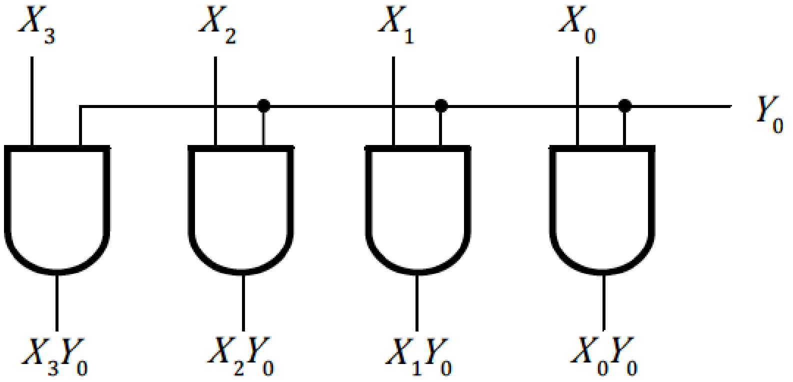


Figure 5-27 Partial product generation

Shift left operation

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
original number	0	0	0	0	1	1	0	1	*0*	*dec. 13*

Shift left operation

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
original number	0	0	0	0	1	1	0	1	*0*	*dec. 13*
after shift left	0	0	0	1	1	0	1	0		*dec. 26*

Shift left operation

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
original number	0	0	0	0	1	1	0	1	*0*	*dec. 13*
after shift left	0	0	0	1	1	0	1	0		*dec. 26*

Shift left → × 2

Multiplication using shift and add

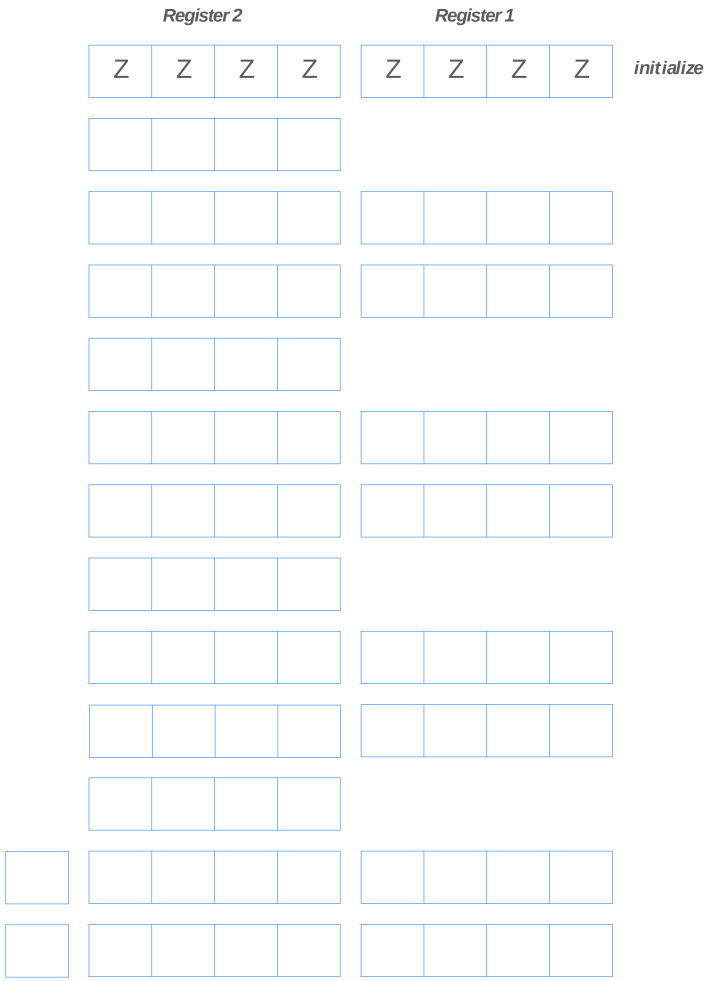
$$
\begin{array}{cccccccccll}
 & & & & & 1 & 0 & 1 & 1 & A_3A_2A_1A_0 & \text{(decimal 11)} \\
 & & & & \times & 1 & 1 & 0 & 1 & B_3B_2B_1B_0 & \text{(decimal 13)} \\
\hline
 & & & & & 1 & 0 & 1 & 1 & \text{since } B_0 = 1 & \\
+ & & & & 0 & 0 & 0 & 0 & Z & \text{since } B_1 = 0 & \\
\hline
 & & & & 0 & 1 & 0 & 1 & 1 & \text{addition} & \\
+ & & & 1 & 0 & 1 & 1 & Z & Z & \text{since } B_2 = 1 & \\
\hline
 & & & 1 & 1 & 0 & 1 & 1 & 1 & \text{addition} & \\
+ & & 1 & 0 & 1 & 1 & Z & Z & Z & \text{since } B_3 = 1 & \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{addition} & \text{(decimal 143)}
\end{array}
$$

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
	1	0	1	1	Z	Z	since $B_2 = 1$	
+	1	1	0	1	1	1	addition	
	10	1	1	Z	Z	Z	since $B_3 = 1$	
+	100	0	1	1	1	1	addition	(decimal 143)

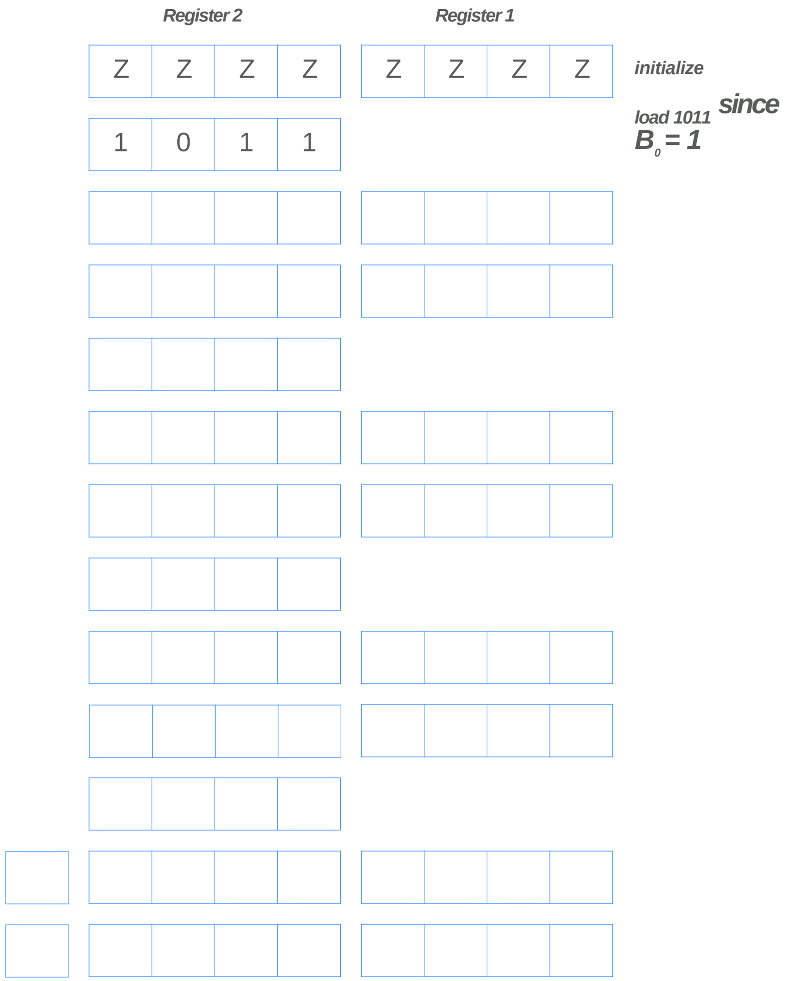
Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.



Multiplication using shift and add

					1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×					1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
					1	0	1	1	since $B_0 = 1$	
+				0	0	0	0	Z	since $B_1 = 0$	
				0	1	0	1	1	addition	
+			1	0	1	1	Z	Z	since $B_2 = 1$	
			1	1	0	1	1	1	addition	
+		10	1	1	Z	Z	Z		since $B_3 = 1$	
	10	0	0	1	1	1	1		addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.



Register 2				Register 1				
Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
1	0	1	1					*load 1011* ***since*** $B_0 = 1$

Multiplication using shift and add

```
          1  0  1  1   A3A2A1A0      (decimal 11)
     ×    1  1  0  1   B3B2B1B0      (decimal 13)
   ----------------
          1  0  1  1   since B0 = 1
 +     0  0  0  0  Z   since B1 = 0
   ----------------
       0  1  0  1  1   addition
 +  1  0  1  1  Z  Z   since B2 = 1
   ----------------
    1  1  0  1  1  1   addition
 + 10  1  1  Z  Z  Z   since B3 = 1
   ----------------
 100  0  1  1  1  1   addition       (decimal 143)
```

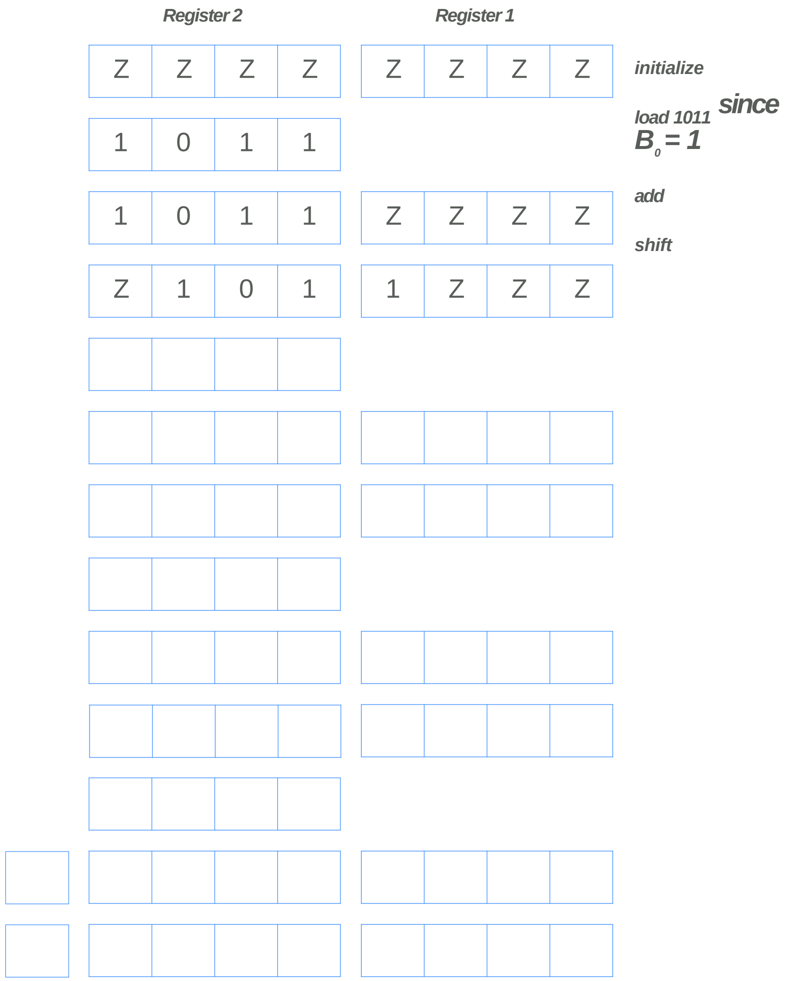
Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add*

Multiplication using shift and add

	1 0 1 1	$A_3A_2A_1A_0$	(decimal 11)
×	1 1 0 1	$B_3B_2B_1B_0$	(decimal 13)
	1 0 1 1	since $B_0 = 1$	
+	0 0 0 0 Z	since $B_1 = 0$	
	0 1 0 1 1	addition	
+	1 0 1 1 Z Z	since $B_2 = 1$	
	1 1 0 1 1 1	addition	
+	10 1 1 Z Z Z	since $B_3 = 1$	
	100 0 1 1 1 1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

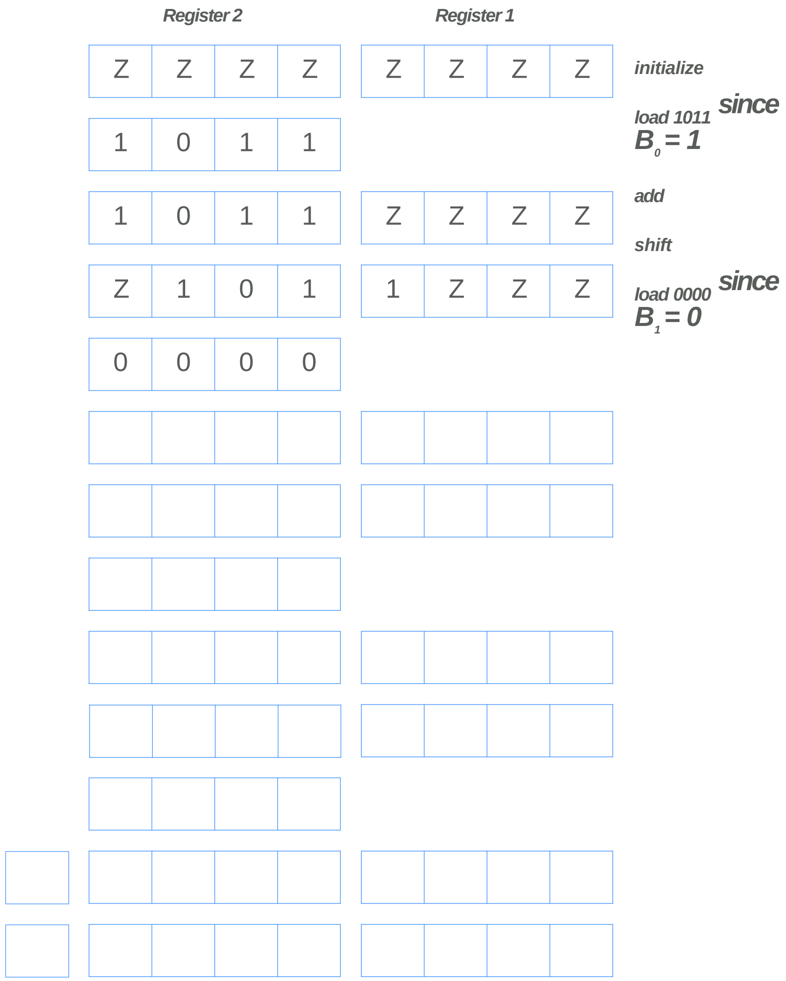


Register 2	Register 1	
Z Z Z Z	Z Z Z Z	initialize
1 0 1 1		load 1011 since $B_0 = 1$
1 0 1 1	Z Z Z Z	add
Z 1 0 1	1 Z Z Z	shift

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.



	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add*
	Z	1	0	1	1	Z	Z	Z	*shift*
	0	0	0	0					*load 0000* since $B_1 = 0$

Multiplication using shift and add

					1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×					1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
					1	0	1	1	since $B_0 = 1$	
+				0	0	0	0	Z	since $B_1 = 0$	
				0	1	0	1	1	addition	
+			1	0	1	1	Z	Z	since $B_2 = 1$	
			1	1	0	1	1	1	addition	
+		1	0	1	1	Z	Z	Z	since $B_3 = 1$	
	1	0	0	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* **since** $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add* *shift*
	Z	1	0	1	1	Z	Z	Z	*load 0000* **since** $B_1 = 0$
	0	0	0	0					*add*
	0	1	0	1	1	Z	Z	Z	

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				*Register 1*				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* *since* $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add*
	Z	1	0	1	1	Z	Z	Z	*shift* *load 0000* *since* $B_1 = 0$
	0	0	0	0					*add*
	0	1	0	1	1	Z	Z	Z	*shift*
	Z	0	1	0	1	1	Z	Z	

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add* *shift*
	Z	1	0	1	1	Z	Z	Z	*load 0000* since $B_1 = 0$
	0	0	0	0					*add*
	0	1	0	1	1	Z	Z	Z	*shift*
	Z	0	1	0	1	1	Z	Z	
	1	0	1	1					*load 1011* since $B_2 = 1$

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* *since* ***$B_0 = 1$***
	1	0	1	1	Z	Z	Z	Z	*add* *shift*
	Z	1	0	1	1	Z	Z	Z	*load 0000* *since* ***$B_1 = 0$***
	0	0	0	0					*add*
	0	1	0	1	1	Z	Z	Z	*shift*
	Z	0	1	0	1	1	Z	Z	
	1	0	1	1					*load 1011* *since* ***$B_2 = 1$***
	1	1	0	1	1	1	Z	Z	*add*

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				*Register 1*					
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*	
	1	0	1	1					*load 1011*	since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add*	
	Z	1	0	1	1	Z	Z	Z	*shift*	
	0	0	0	0					*load 0000*	since $B_1 = 0$
	0	1	0	1	1	Z	Z	Z	*add*	
	Z	0	1	0	1	1	Z	Z	*shift*	
	1	0	1	1					*load 1011*	since $B_2 = 1$
	1	1	0	1	1	1	Z	Z	*add*	
	Z	1	1	0	1	1	1	Z	*shift*	

Multiplication using shift and add

					1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×					1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
					1	0	1	1	since $B_0 = 1$	
+				0	0	0	0	Z	since $B_1 = 0$	
				0	1	0	1	1	addition	
+			1	0	1	1	Z	Z	since $B_2 = 1$	
			1	1	0	1	1	1	addition	
+		1	0	1	1	Z	Z	Z	since $B_3 = 1$	
	1	0	0	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	*initialize*
	1	0	1	1					*load 1011* since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	*add* *shift*
	Z	1	0	1	1	Z	Z	Z	*load 0000* since $B_1 = 0$
	0	0	0	0					*add*
	0	1	0	1	1	Z	Z	Z	*shift*
	Z	0	1	0	1	1	Z	Z	
	1	0	1	1					*load 1011* since $B_2 = 1$
	1	1	0	1	1	1	Z	Z	*add*
	Z	1	1	0	1	1	1	Z	*shift*
	1	0	1	1					*load 1011* since $B_3 = 1$

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	initialize
	1	0	1	1					load 1011 since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	add
	Z	1	0	1	1	Z	Z	Z	shift
	0	0	0	0					load 0000 since $B_1 = 0$
	0	1	0	1	1	Z	Z	Z	add
	Z	0	1	0	1	1	Z	Z	shift
	1	0	1	1					load 1011 since $B_2 = 1$
	1	1	0	1	1	1	Z	Z	add
	Z	1	1	0	1	1	1	Z	shift
	1	0	1	1					load 1011 since $B_3 = 1$
1	0	0	0	1	1	1	1	Z	add

Multiplication using shift and add

			1	0	1	1	$A_3A_2A_1A_0$	(decimal 11)
×			1	1	0	1	$B_3B_2B_1B_0$	(decimal 13)
			1	0	1	1	since $B_0 = 1$	
+		0	0	0	0	Z	since $B_1 = 0$	
		0	1	0	1	1	addition	
+	1	0	1	1	Z	Z	since $B_2 = 1$	
	1	1	0	1	1	1	addition	
+	10	1	1	Z	Z	Z	since $B_3 = 1$	
	100	0	1	1	1	1	addition	(decimal 143)

Note that Z = 0. We use Z to denote 0s which are independent of the numbers being multiplied.

	Register 2				Register 1				
	Z	Z	Z	Z	Z	Z	Z	Z	initialize
	1	0	1	1					load 1011 since $B_0 = 1$
	1	0	1	1	Z	Z	Z	Z	add
	Z	1	0	1	1	Z	Z	Z	shift
	0	0	0	0					load 0000 since $B_1 = 0$
	0	1	0	1	1	Z	Z	Z	add
	Z	0	1	0	1	1	Z	Z	shift
	1	0	1	1					load 1011 since $B_2 = 1$
	1	1	0	1	1	1	Z	Z	add
	Z	1	1	0	1	1	1	Z	shift
	1	0	1	1					load 1011 since $B_3 = 1$
1	0	0	0	1	1	1	1	Z	add
Z	1	0	0	0	1	1	1	1	shift

Sequential Logics

- In this section, we will focus on another class of circuit: sequential logic circuit.
- Why do we need the sequential circuit?
  - Suppose we have several combinational logic blocks with different functions, and the output of one block is the input of another one. Then how do we manage the data flow between the blocks, if each block needs different time to complete its computation?

  - A solution is to employ the memory circuit to store the instant logic outputs from the combinational blocks, and latch them to the other blocks in a synchronized manner controlled by clocks.

- *Clearly, the clock period should be longer than that needed for all combinational logic blocks to complete their computation, which is usually the slowest block in the circuit. In this way, large scale digital system can be practically built.*

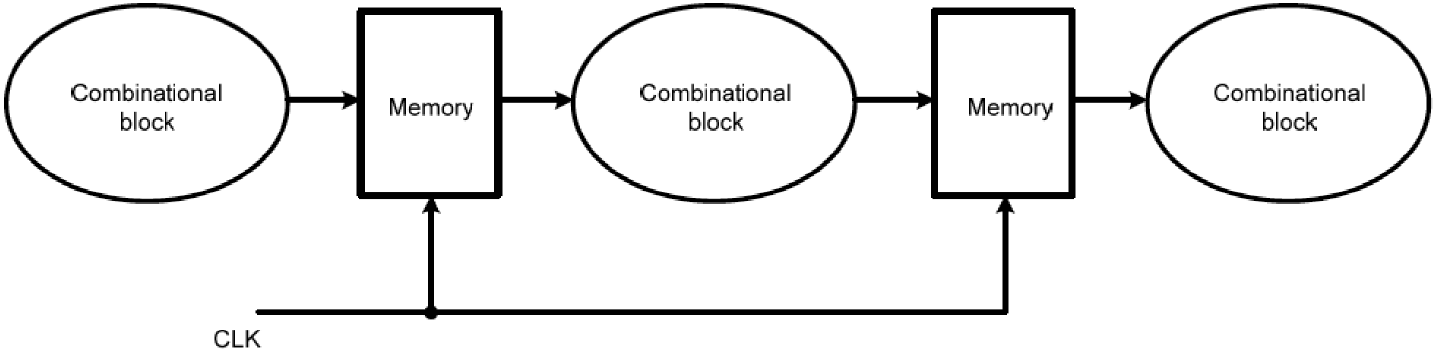
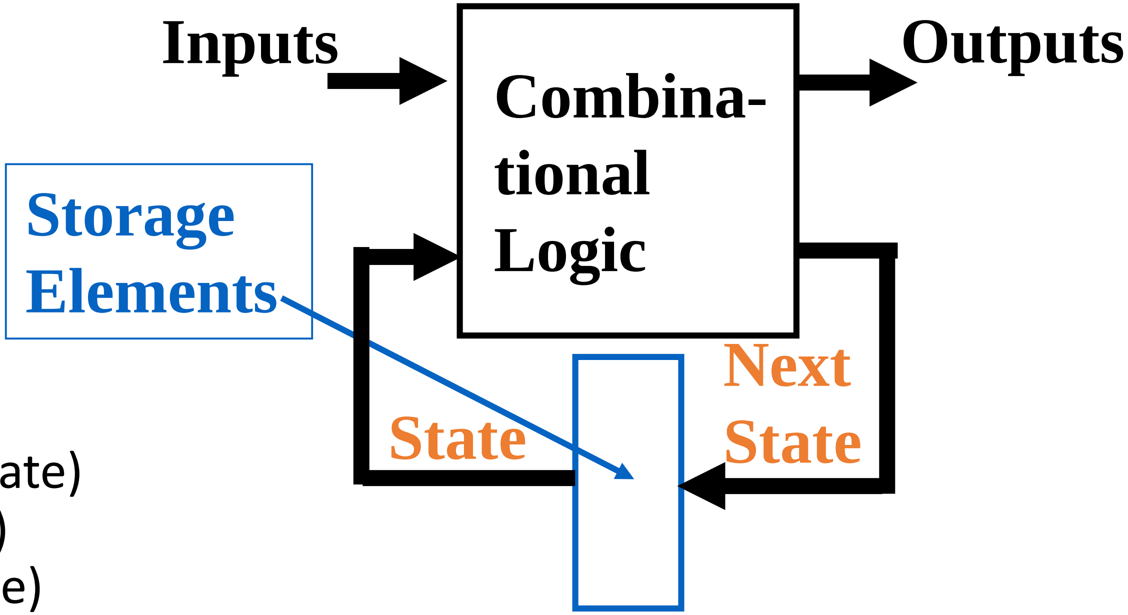


Figure 5-32 Structure of the sequential circuit



- Combinatorial Logic
  - *Next state function*
    Next State = f(Inputs, State)
  - *Output function* (Mealy)
    Outputs = g(Inputs, State)
  - *Output function* (Moore)
    Outputs = h(State)
- Output function type depends on specification and affects the design significantly

Types of Sequential Circuits

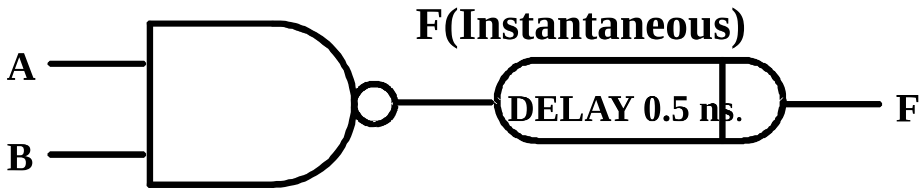
- Depends on the times at which:
  - storage elements observe their inputs, and
  - storage elements change their state
- Synchronous
  - Behavior defined from knowledge of its signals at discrete instances of time
  - Storage elements observe inputs and can change state only in relation to a timing signal (clock pulses from a clock)
- Asynchronous
  - Behavior defined from knowledge of inputs an any instant of time and the order in continuous time in which inputs change
  - If clock just regarded as another input, all circuits are asynchronous!
  - Nevertheless, the synchronous abstraction makes complex designs tractable!

Discrete Event Simulation

- In order to understand the time behavior of a sequential circuit we use discrete event simulation.
- Rules:
  - Gates modeled by an ideal (instantaneous) function and a fixed gate delay
  - Any change in input values is evaluated to see if it causes a change in output value
  - Changes in output values are scheduled for the fixed gate delay after the input change
  - At the time for a scheduled output change, the output value is changed along with any inputs it drives

Simulated NAND Gate

- Example: A 2-Input NAND gate with a 0.5 ns. delay:

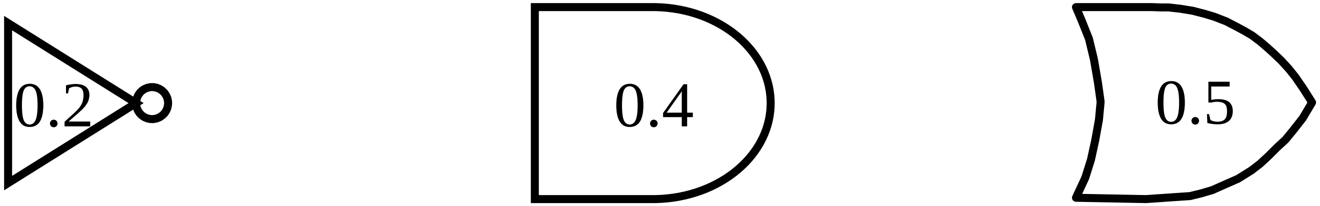


- Assume A and B have been 1 for a long time
- At time t=0, A changes to a 0 at t= 0.8 ns, back to 1.

t (ns)	A	B	F(I)	F	Comment
$-\infty$	1	1	0	0	A=B=1 for a long time
0	$1 \Rightarrow 0$	1	$1 \Leftarrow 0$	0	F(I) changes to 1
0.5	0	1	1	$1 \Leftarrow 0$	F changes to 1 after a 0.5 ns delay
0.8	$1 \Leftarrow 0$	1	$1 \Rightarrow 0$	1	F(Instantaneous) changes to 0
0.13	1	1	0	$1 \Rightarrow 0$	F changes to 0 after a 0.5 ns delay

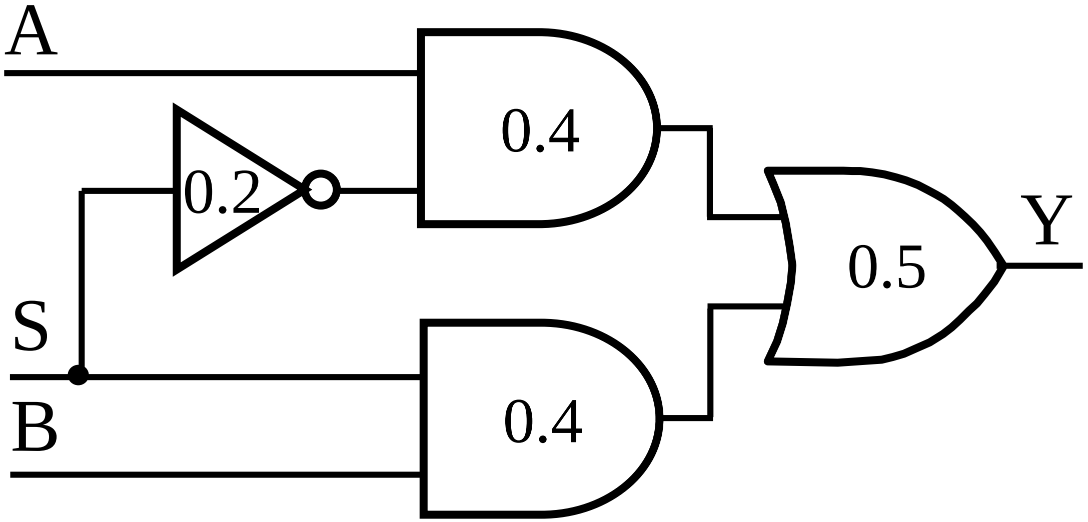
Gate Delay Models

- Suppose gates with delay *n* ns are represented for *n* = 0.2 ns, *n* = 0.4 ns, *n* = 0.5 ns, respectively:

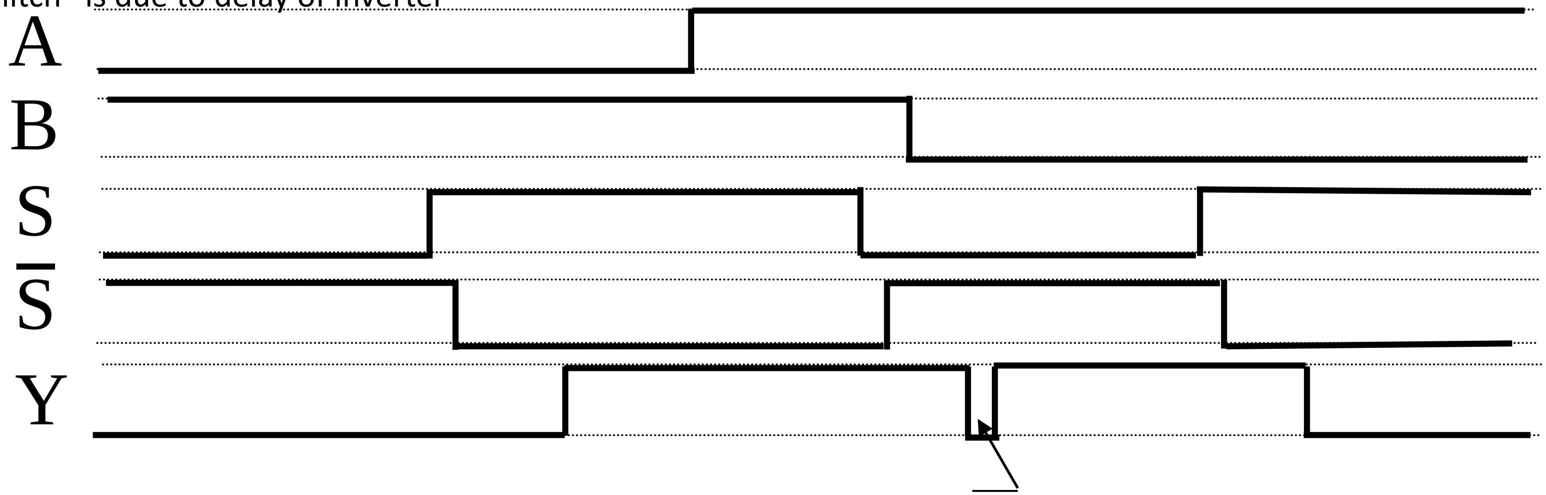


Circuit Delay Model

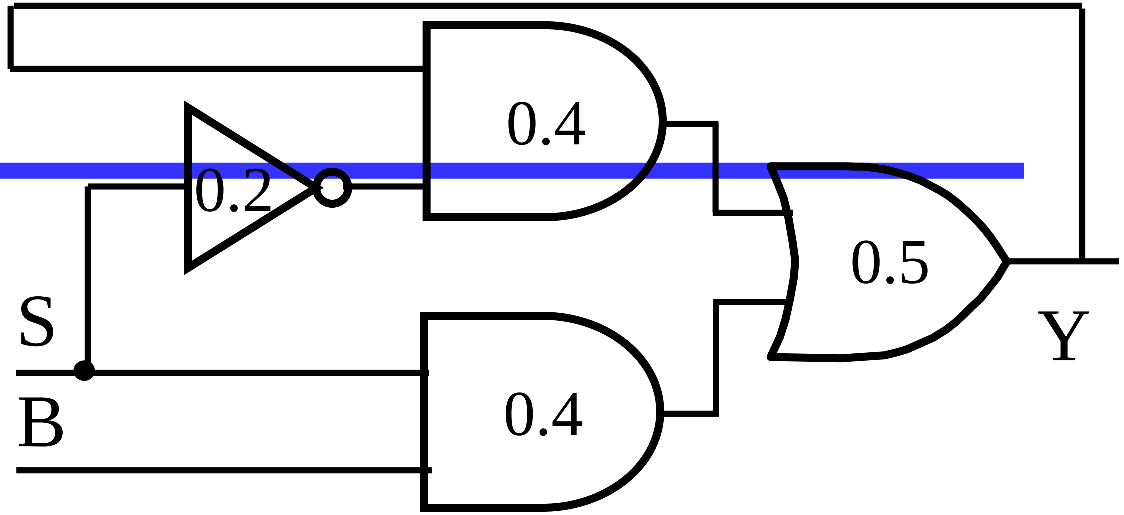
- Consider a simple 2-input multiplexer:
- With function:
  - Y = A for S = 1
  - Y = B for S = 0



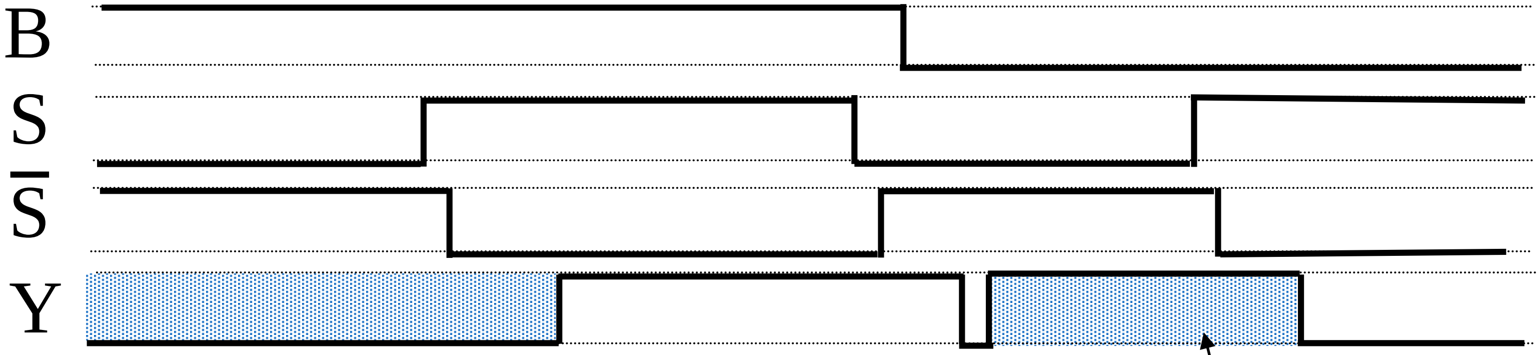
- “Glitch” is due to delay of inverter



Storing State



- What if A connected to Y?
- Circuit becomes:
- With function:
  - Y = B for S = 1, and Y(t) dependent on Y(t – 0.9) for S = 0
- The simple combinational circuit
- has now become a sequential circuit because its output is a function of a time sequence of input signals!



Y is stored value in shaded area

Storing State (Continued)

- Simulation example as input signals change with time. Changes occur every 100 ns, so that the tenths of ns delays are negligible.

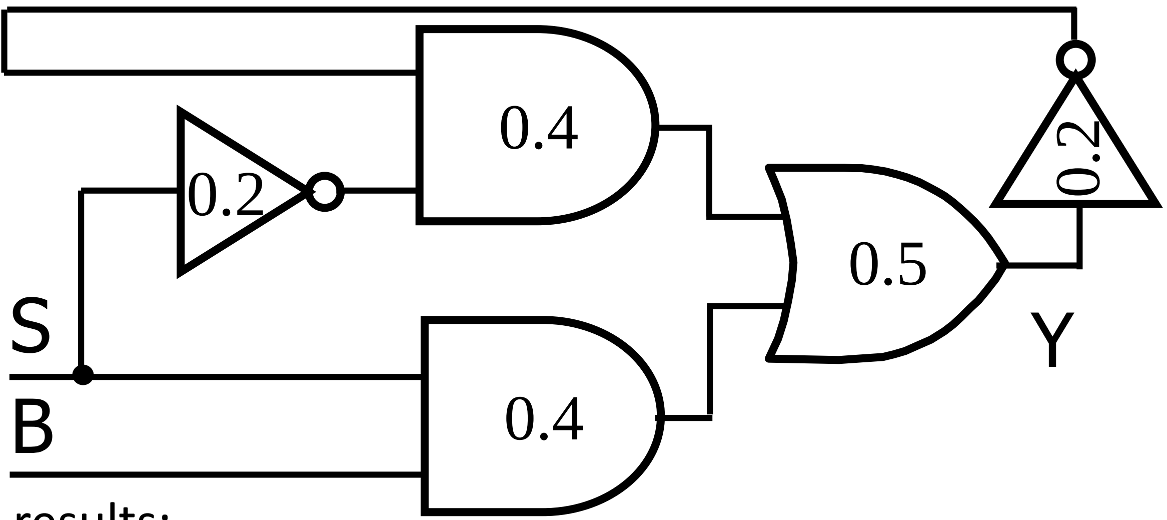
Time

B	S	Y	Comment
1	0	0	Y “remembers” 0
1	1	1	Y = B when S = 1
1	0	1	Now Y “remembers” B = 1 for S = 0
0	0	1	No change in Y when B changes
0	1	0	Y = B when S = 1
0	0	0	Y “remembers” B = 0 for S = 0
1	0	0	No change in Y when B changes

- Y represent the state of the circuit, not just an output.

Storing State (Continued)

- Suppose we place an inverter in the "feedback path."

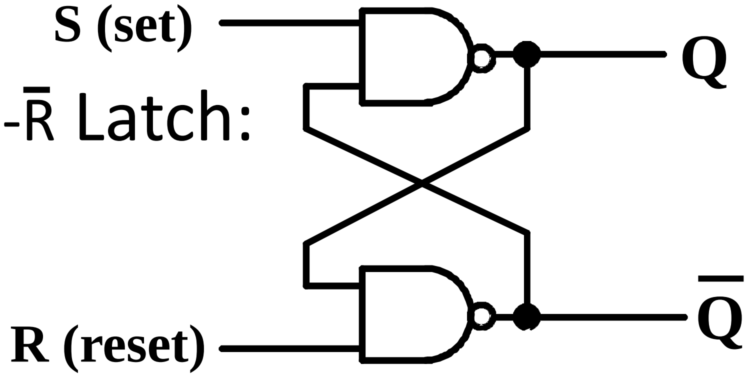


- The following behavior results:
- The circuit is said to be unstable.
- For S = 0, the circuit has become what is called an *oscillator*. Can be used as crude clock.

B	S	Y	Comment
0	1	0	Y = B when S = 1
1	1	1	
1	0	1	Now Y "remembers" A
1	0	0	Y, 1.1 ns later
1	0	1	Y, 1.1 ns later
1	0	0	Y, 1.1 ns later

Basic (NAND) $\overline{S}$ – $\overline{R}$ Latch

- "Cross-Coupling"two NAND gates gives the $\overline{S}$ -$\overline{R}$ Latch:
- Which has the time sequence behavior:



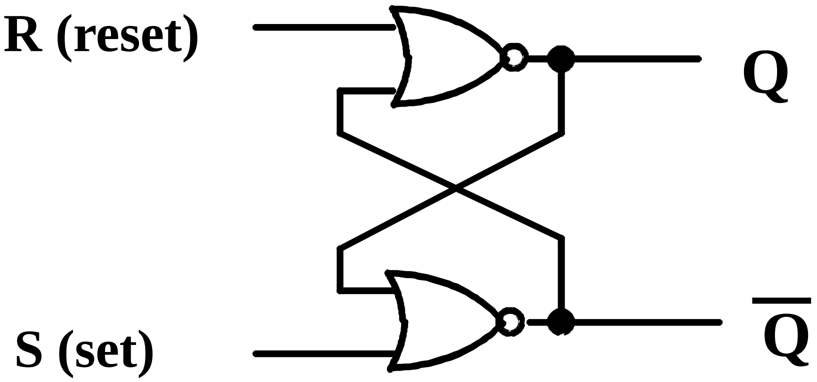
Time

R	S	Q	$\overline{Q}$	Comment
1	1	?	?	Stored state unknown
1	0	1	0	"Set" Q to 1
1	1	1	0	Now Q "remembers" 1
0	1	0	1	"Reset" Q to 0
1	1	0	1	Now Q "remembers" 0
0	0	1	1	Both go high
1	1	?	?	Unstable!

- S = 0, R = 0 is forbidden as input pattern

Basic (NOR) S – R Latch

- Cross-coupling two NOR gates gives the S – R Latch:
- Which has the time sequence behavior:

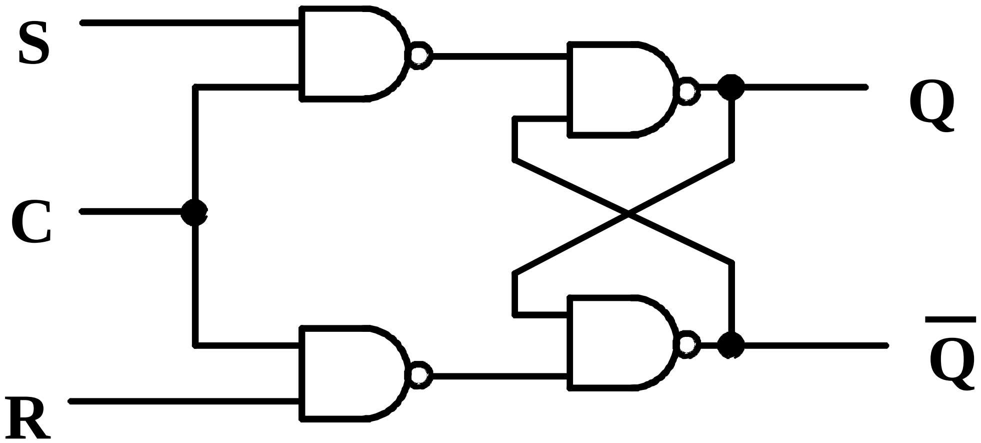


Time ↓

R	S	Q	$\overline{Q}$	Comment
0	0	?	?	Stored state unknown
0	1	1	0	“Set” Q to 1
0	0	1	0	Now Q “remembers” 1
1	0	0	1	“Reset” Q to 0
0	0	0	1	Now Q “remembers” 0
1	1	0	0	Both go low
0	0	?	?	Unstable!

Clocked S - R Latch

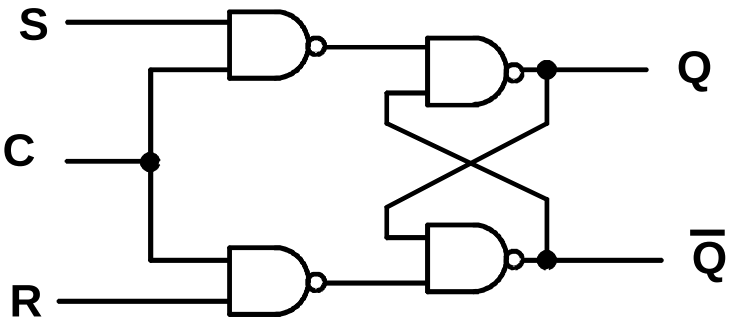
- Adding two NAND gates to the basic $\overline{S}$ - $\overline{R}$ NAND latch gives the clocked S – R latch:



- Has a time sequence behavior similar to the basic S-R latch except that the S and R inputs are only observed when the line C is high.
- C means “control” or “clock”.

Clocked S - R Latch (continued)

- The Clocked S-R Latch can be described by a table:

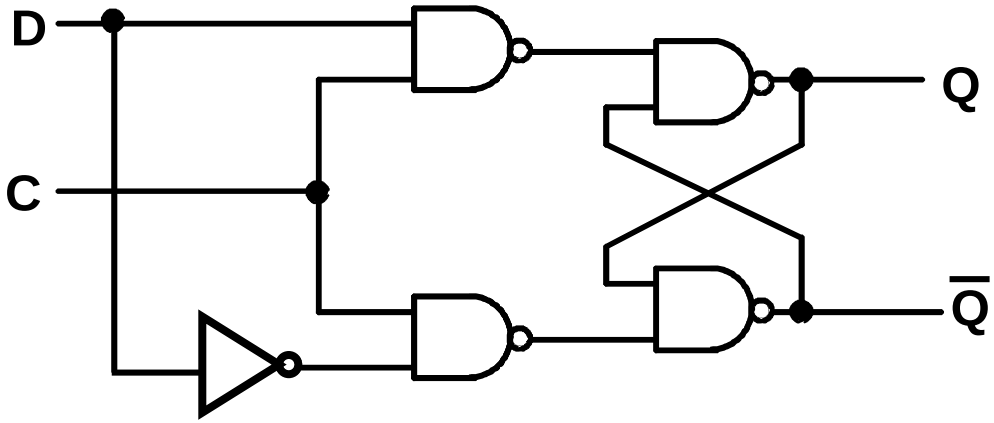


Q(t)	S	R	Q(t+1)	Comment
0	0	0	0	No change
0	0	1	0	Clear Q
0	1	0	1	Set Q
0	1	1	???	Indeterminate
1	0	0	1	No change
1	0	1	0	Clear Q
1	1	0	1	Set Q
1	1	1	???	Indeterminate

- The table describes what happens after the clock [at time (t+1)] based on:
  - current inputs (S,R) and
  - current state Q(t).

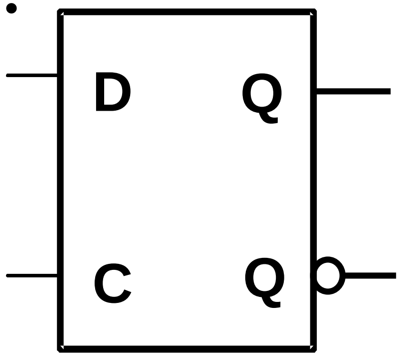
D Latch

- Adding an inverter to the S-R Latch, gives the D Latch:
- Note that there are no “indeterminate” states!



Q	D	Q(t+1)	Comment
0	0	0	No change
0	1	1	Set Q
1	0	0	Clear Q
1	1	1	No Change

The graphic symbol for a D Latch is:



Flip-Flops

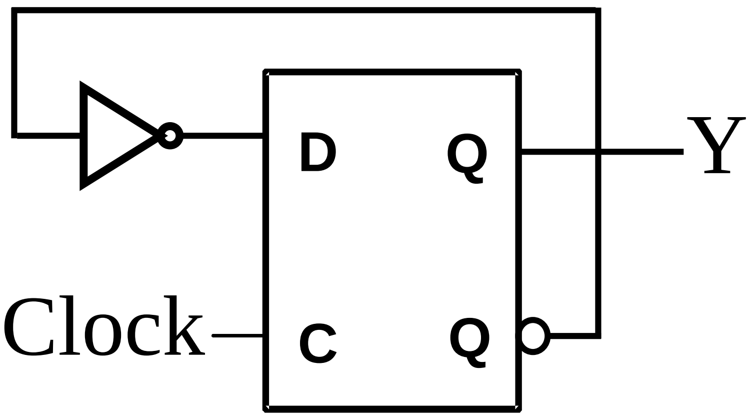
- The latch timing problem
- Master-slave flip-flop
- Edge-triggered flip-flop
- Standard symbols for storage elements
- Direct inputs to flip-flops

The Latch Timing Problem

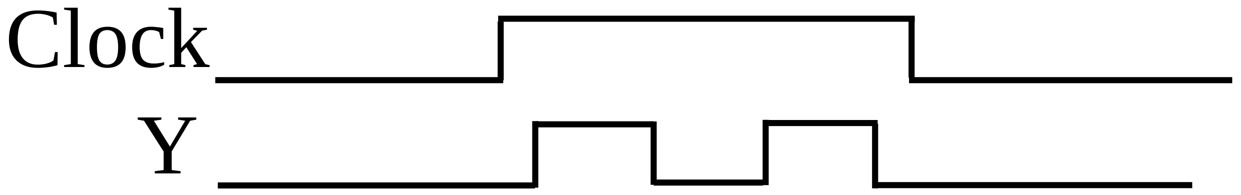
- In a sequential circuit, paths may exist through combinational logic:
  - From one storage element to another
  - From a storage element back to the same storage element
- The combinational logic between a latch output and a latch input may be as simple as an interconnect
- For a clocked D-latch, the output Q depends on the input D whenever the clock input C has value 1

The Latch Timing Problem (continued)

- Consider the following circuit:



- Suppose that initially Y = 0.



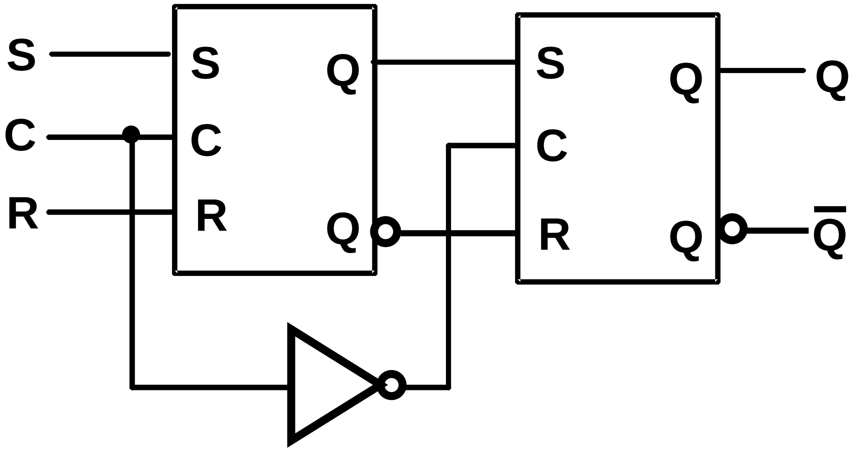
- As long as C = 1, the value of Y continues to change!
- The changes are based on the delay present on the loop through the connection from Y back to Y.
- This behavior is clearly unacceptable.
- <u>Desired behavior</u>: Y changes <u>only once</u> per clock pulse

The Latch Timing Problem (continued)

- A solution to the latch timing problem is to break the closed path from Y to Y within the storage element
- The commonly-used, path-breaking solutions replace the clocked D-latch with:
  - a master-slave flip-flop
  - an edge-triggered flip-flop

S-R Master-Slave Flip-Flop

- Consists of two clocked S-R latches in series with the clock on the second latch inverted
- The input is observed by the first latch with C = 1



- The output is changed by the second latch with C = 0
- The path from input to output is broken by the difference in clocking values (C = 1 and C = 0).
- The behavior demonstrated by the example with D driven by Y given previously is prevented since the clock must change from 1 to 0 before a change in Y based on D can occur.

Logic and Computer Design Fundamentals, 4e
PowerPoint® Slides

Flip-Flop Problem

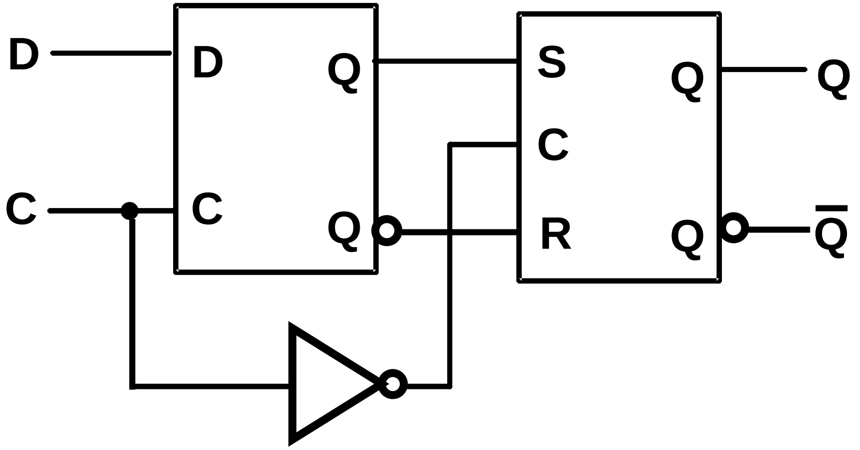
- The change in the flip-flop output is delayed by the pulse width which makes the circuit slower or
- S and/or R are permitted to change while C = 1
  - Suppose Q = 0 and S goes to 1 and then back to 0 with R remaining at 0
    - The master latch sets to 1
    - A 1 is transferred to the slave
  - Suppose Q = 0 and S goes to 1 and back to 0 and R goes to 1 and back to 0
    - The master latch sets and then resets
    - A 0 is transferred to the slave
  - This behavior is called *1s catching*

Flip-Flop Solution

- Use edge-triggering instead of master-slave
- An *edge-triggered* flip-flop ignores the pulse while it is at a constant level and triggers only during a transition of the clock signal
- Edge-triggered flip-flops can be built directly at the electronic circuit level, or
- A master-slave D flip-flop which also exhibits edge-triggered behavior can be used.

Edge-Triggered D Flip-Flop

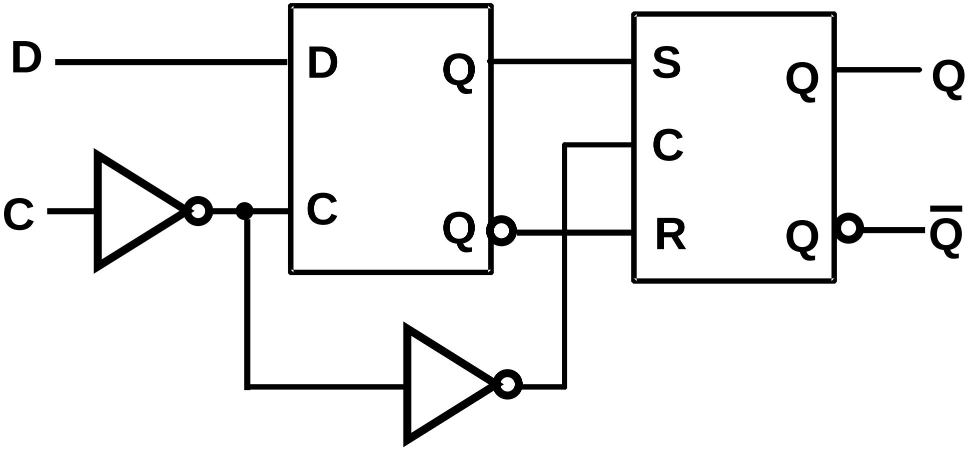
- The edge-triggered D flip-flop is the same as the master-slave D flip-flop



- It can be formed by:
  - Replacing the first clocked S-R latch with a clocked D latch or
  - Adding a D input and inverter to a master-slave S-R flip-flop
- The delay of the S-R master-slave flip-flop can be avoided since the 1s-catching behavior is not present with D replacing S and R inputs
- The change of the D flip-flop output is associated with the negative edge at the end of the pulse
- It is called a *negative-edge triggered* flip-flop

Positive-Edge Triggered D Flip-Flop

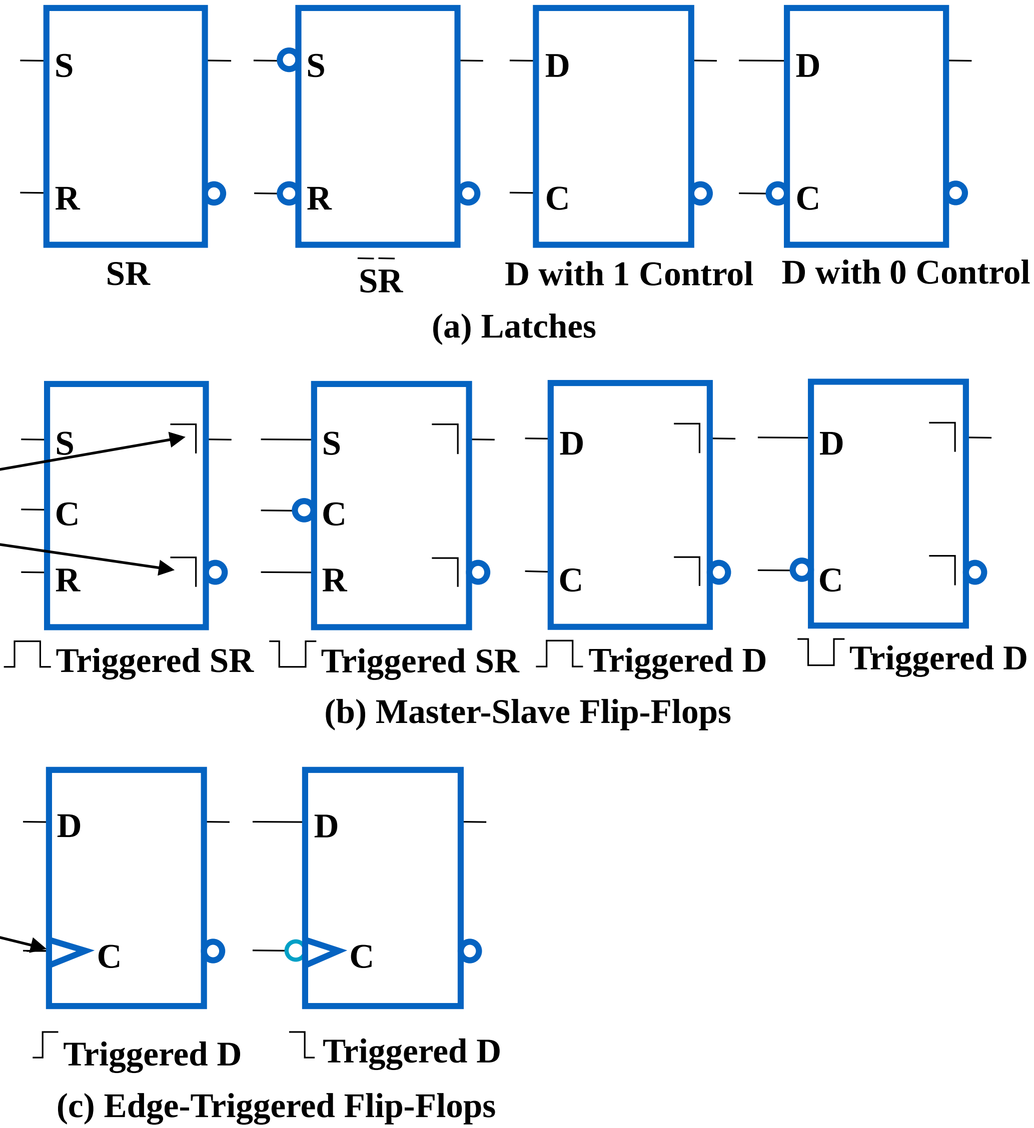
- Formed by adding inverter to clock input



- Q changes to the value on D applied at the positive clock edge within timing constraints to be specified
- Our choice as the standard flip-flop for most sequential circuits

Standard Symbols for Storage Elements

- Master-Slave: Postponed output indicators
- Edge-Triggered: Dynamic indicator



SR $\quad \overline{S}\,\overline{R}\quad$ D with 1 Control $\quad$ D with 0 Control

(a) Latches

Triggered SR $\quad$ Triggered SR $\quad$ Triggered D $\quad$ Triggered D

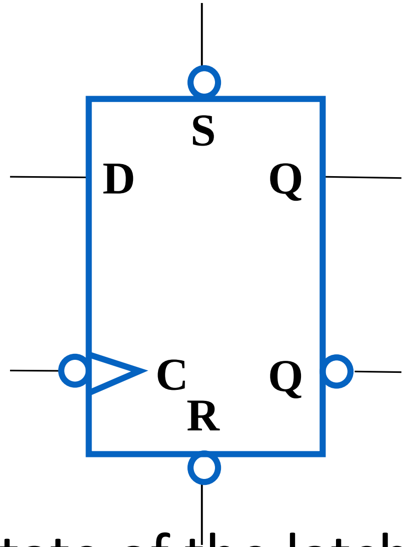
(b) Master-Slave Flip-Flops

Triggered D $\quad$ Triggered D

(c) Edge-Triggered Flip-Flops

Direct Inputs

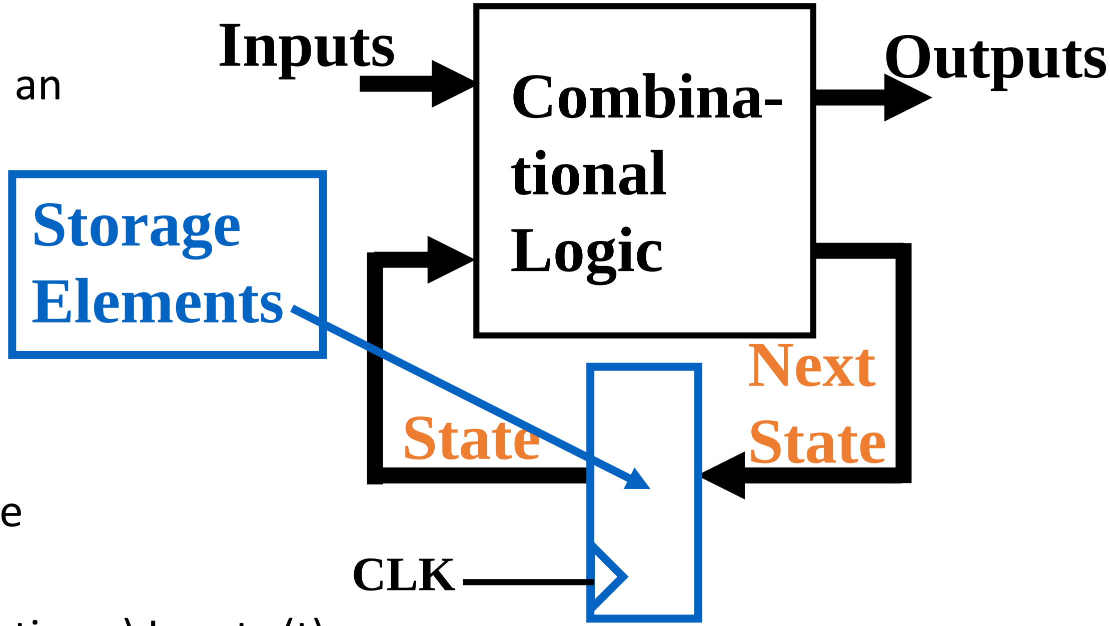
- At power up or at reset, all or part of a sequential circuit usually is initialized to a known state before it begins operation
- This initialization is often done outside of the clocked behavior of the circuit, i.e., asynchronously.



- Direct R and/or S inputs that control the state of the latches within the flip-flops are used for this initialization.
- For the example flip-flop shown
  - 0 applied to R resets the flip-flop to the 0 state
  - 0 applied to S sets the flip-flop to the 1 state

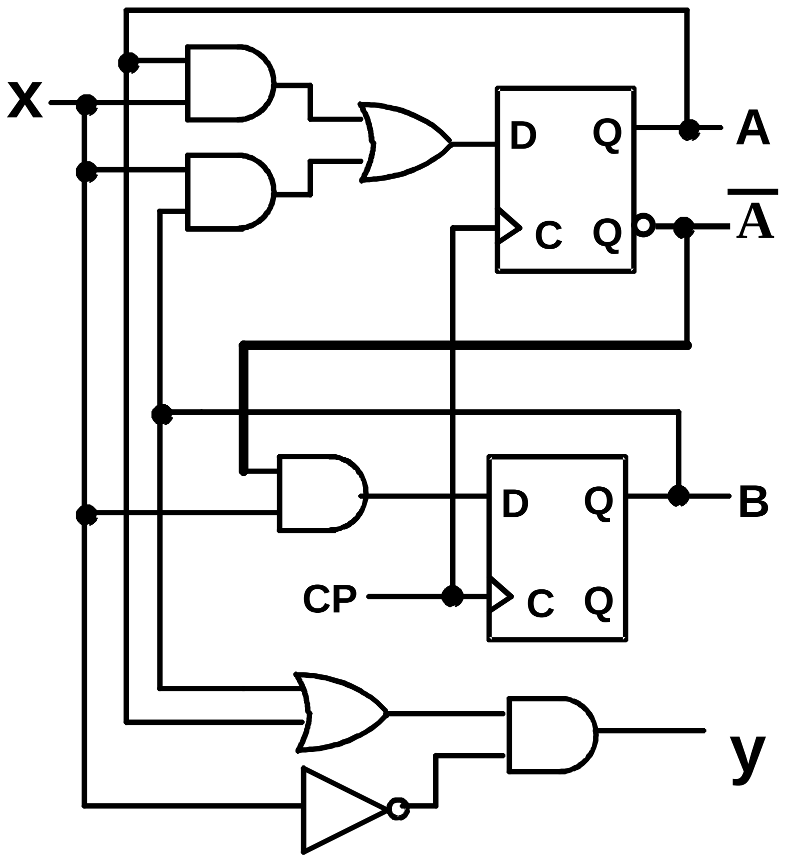
Sequential Circuit Analysis

- General Model
  - Current State at time (t) is stored in an array of flip-flops.
  - Next State at time (t+1) is a Boolean function Of State and Inputs.
  - Outputs at time (t) are a Boolean function of State (t) and (sometimes) Inputs (t).



Example 1

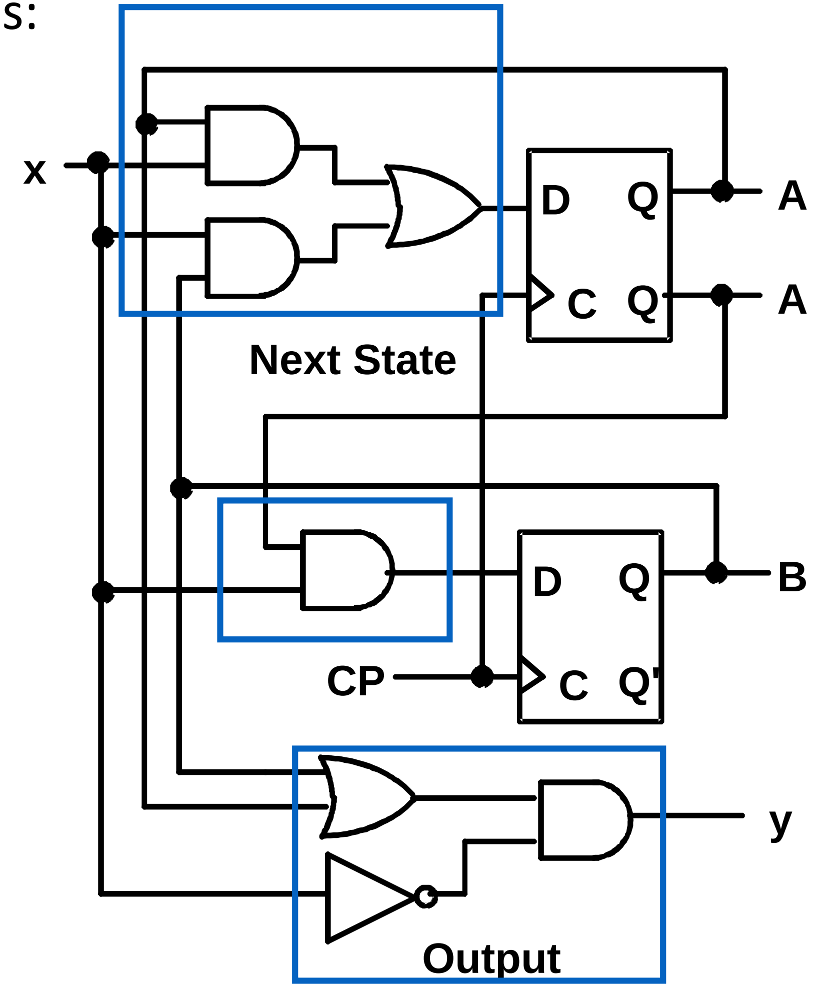
- Input: x(t)
- Output: y(t)
- State: (A(t), B(t))
- What is the Output Function?
- What is the Next State Function?



Example 1 (continued)

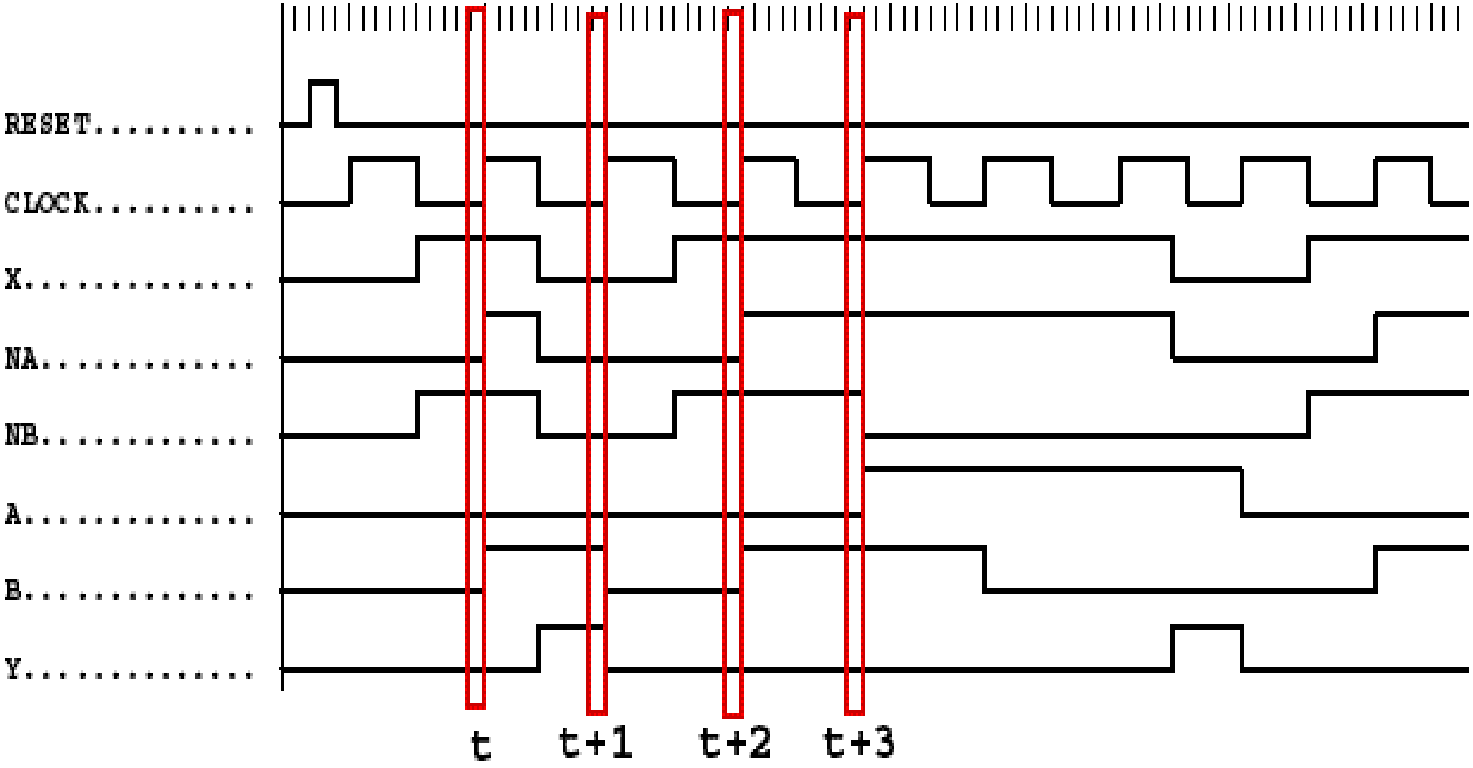
- Boolean equations for the functions:

$$A(t+1) = A(t)x(t) + B(t)x(t)$$
$$B(t+1) = \overline{A}(t)x(t)$$
$$y(t) = \overline{x}(t)(B(t) + A(t))$$



Example 1 (continued)

- Where in time are inputs, outputs and states defined?



State Table Characteristics

- *State table* – a multiple variable table with the following four sections:
  - *Present State* – the values of the state variables for each allowed state.
  - *Input* – the input combinations allowed.
  - *Next-state* – the value of the state at time (t+1) based on the present state and the input.
  - *Output* – the value of the output as a function of the present state and (sometimes) the input.
- From the viewpoint of a truth table:
  - the inputs are Input, Present State
  - and the outputs are Output, Next State

Example 1: State Table

- The state table can be filled in using the next state and output equations: $A(t+1) = A(t)x(t) + B(t)x(t)$ $B(t+1) = \overline{A}(t)x(t)$ $y(t) = \overline{x}(t)(B(t) + A(t))$

Present State		Input	Next State		Output
A(t)	B(t)	x(t)	A(t+1)	B(t+1)	y(t)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Example 1: Alternate State Table

- 2-dimensional table that matches well to a K-map. Present state rows and input columns in Gray code order.
  - $A(t+1) = A(t)x(t) + B(t)x(t)$
  - $B(t+1) = \overline{A}(t)x(t)$
  - $y(t) = \overline{x}(t)(B(t) + A(t))$

Present	**Next State**		**Output**	
State	**x(t)=0**	**x(t)=1**	**x(t)=0**	**x(t)=1**
A(t) B(t)	**A(t+1)B(t+1)**	**A(t+1)B(t+1)**	**y(t)**	**y(t)**
0 0	**0 0**	**0 1**	**0**	**0**
0 1	**0 0**	**1 1**	**1**	**0**
1 0	**0 0**	**1 0**	**1**	**0**
1 1	**0 0**	**1 0**	**1**	**0**

State Diagrams

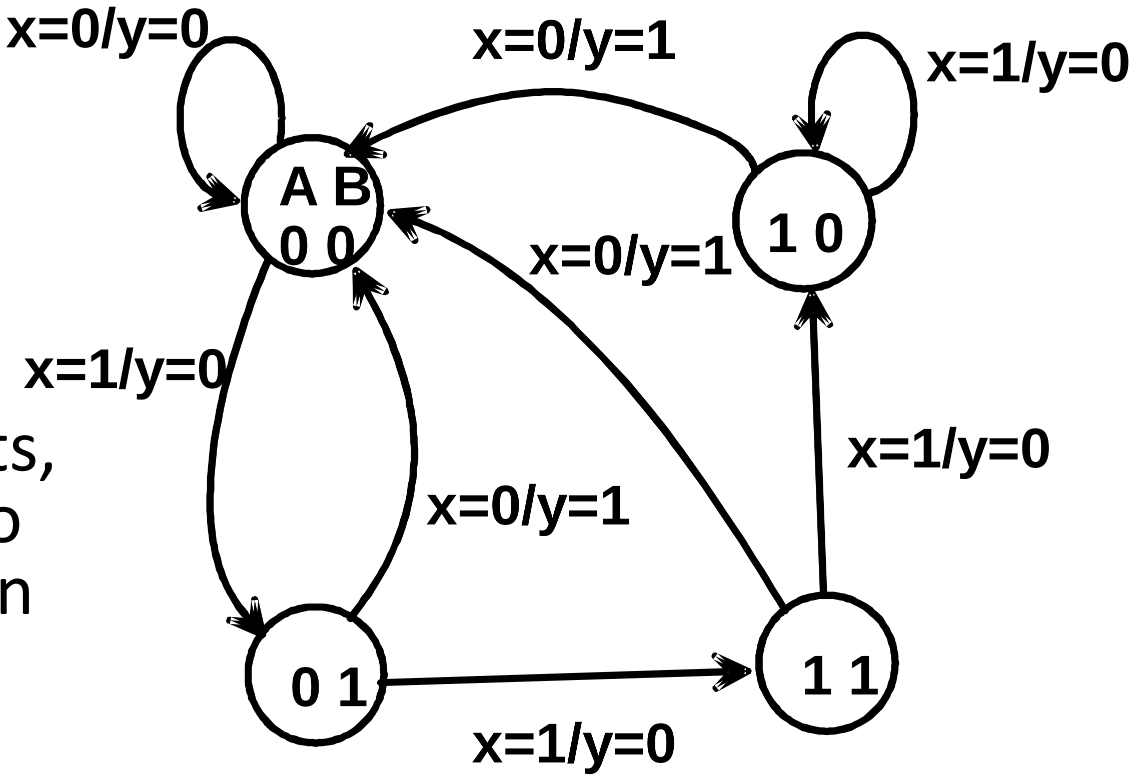
- The sequential circuit function can be represented in graphical form as a state diagram with the following components:
  - A circle with the state name in it for each state
  - A directed arc from the Present State to the Next State for each state transition
  - A label on each directed arc with the Input values which causes the state transition, and
  - A label:
    - On each circle with the output value produced, or
    - On each directed arc with the output value produced.

State Diagrams

- Label form:
  - On circle with output included:
    - state/output
    - Moore type output depends only on state
  - On directed arc with the output included:
    - input/output
    - Mealy type output depends on state and input

Example 1: State Diagram

- Which type?
- Diagram gets confusing for large circuits
- For small circuits, usually easier to understand than the state table

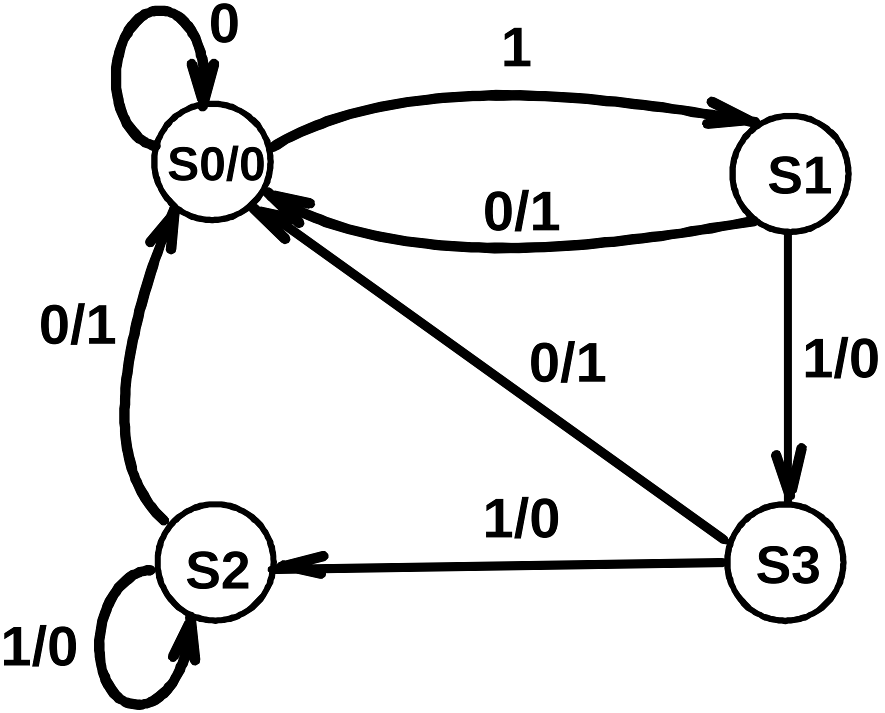


Equivalent State Definitions

- Two states are *equivalent* if their response for each possible input sequence is an identical output sequence.
- Alternatively, two states are *equivalent* if their outputs produced for each input symbol is identical and their next states for each input symbol are the same or equivalent.

Equivalent State Example

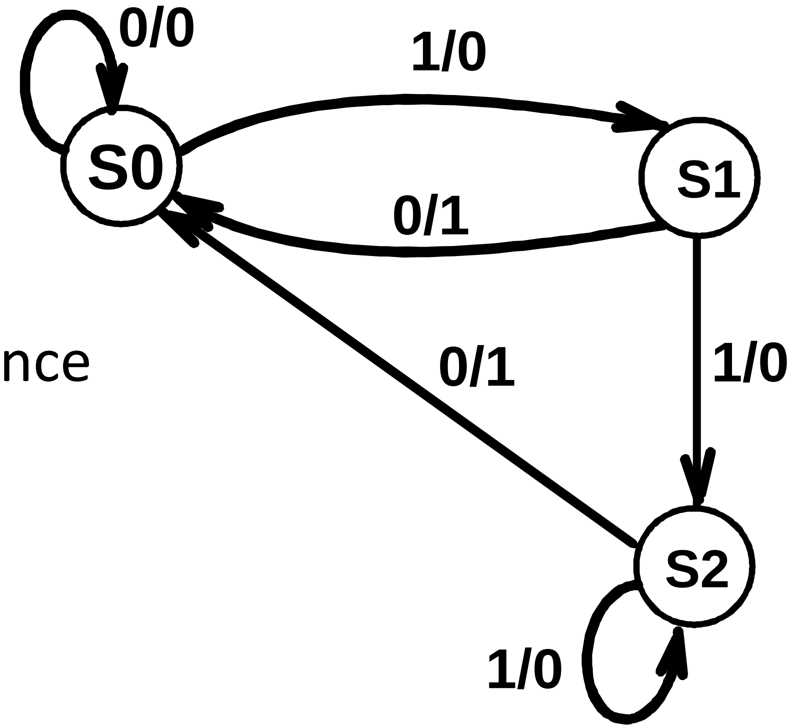
- Text Figure 5-17(a):
- For states S3 and S2,
  - the output for input 0 is 1 and input 1 is 0, and
  - the next state for input 0 is S0 and for input 1 is S2.
  - By the alternative definition, states S3 and S2 are equivalent.

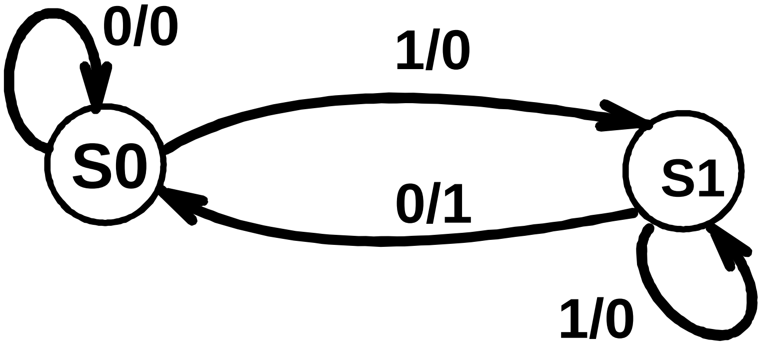


Logic and Computer Design Fundamentals, 4e
PowerPoint® Slides

Equivalent State Example

- Replacing S3 and S2 by a single state gives state diagram:
- Examining the new diagram, states S1 and S2 are equivalent since
  - their outputs for input 0 is 1 and input 1 is 0, and
  - their next state for input 0 is S0 and for input 1 is S2,
- Replacing S1 and S2 by a single state gives state diagram:





Moore and Mealy Models

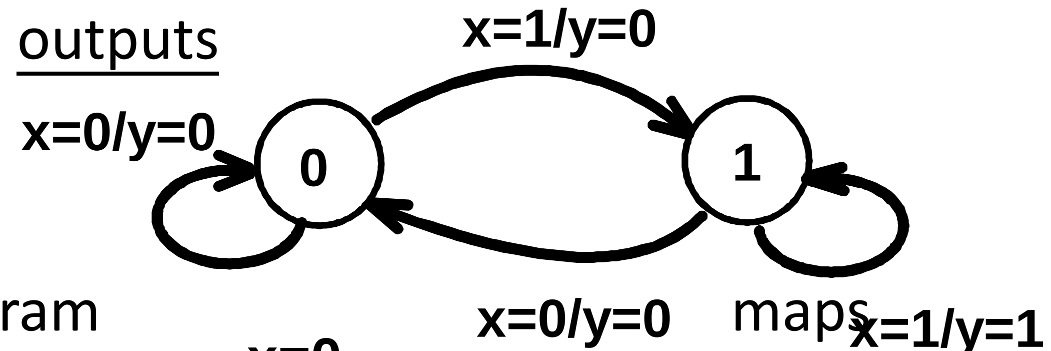
- Sequential Circuits or Sequential Machines are also called *Finite State Machines* (FSMs). Two formal models exist:

- **Moore Model**
  - **Named after E.F. Moore**
  - **Outputs are a function ONLY of states**
  - **Usually specified on the states.**

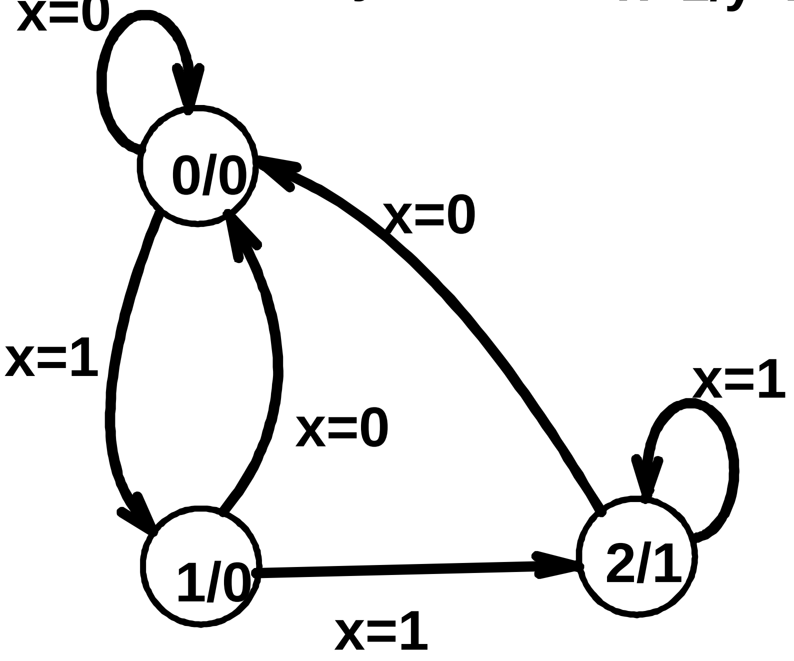
- **Mealy Model**
  - **Named after G. Mealy**
  - **Outputs are a function of inputs and states**
  - **Usually specified on the state transition arcs.**

Moore and Mealy Example Diagrams

- Mealy Model State Diagram maps inputs and state to outputs



- Moore Model State Diagram states to outputs



Moore and Mealy Example Tables

- Moore Model state table maps state to outputs

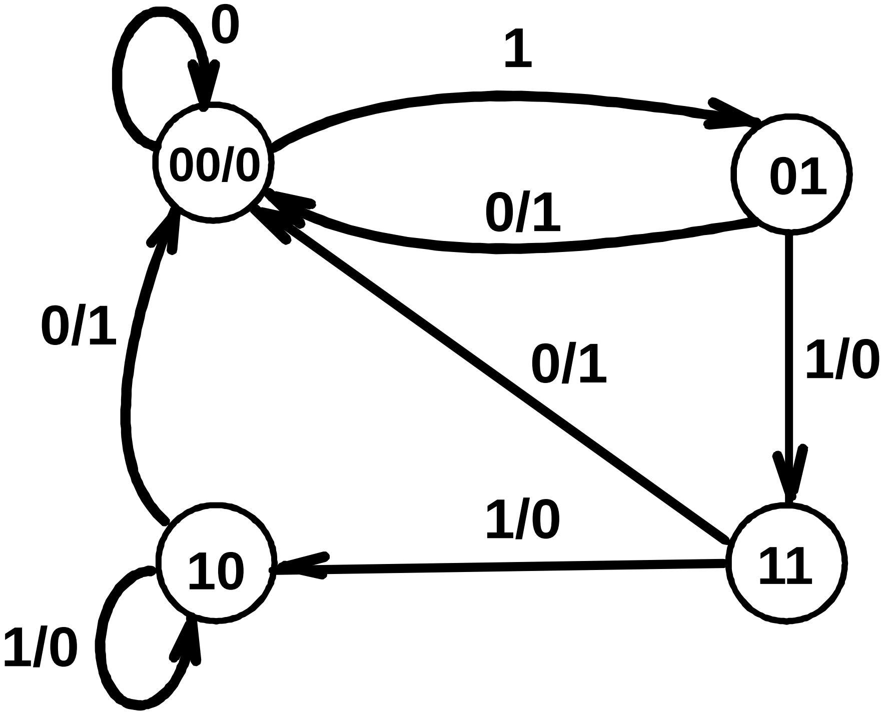
Present State	Next State		Output
	x=0	x=1	
0	0	1	0
1	0	2	0
2	0	2	1

- Mealy Model state table maps inputs and state to outputs

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
0	0	1	0	0
1	0	1	0	1

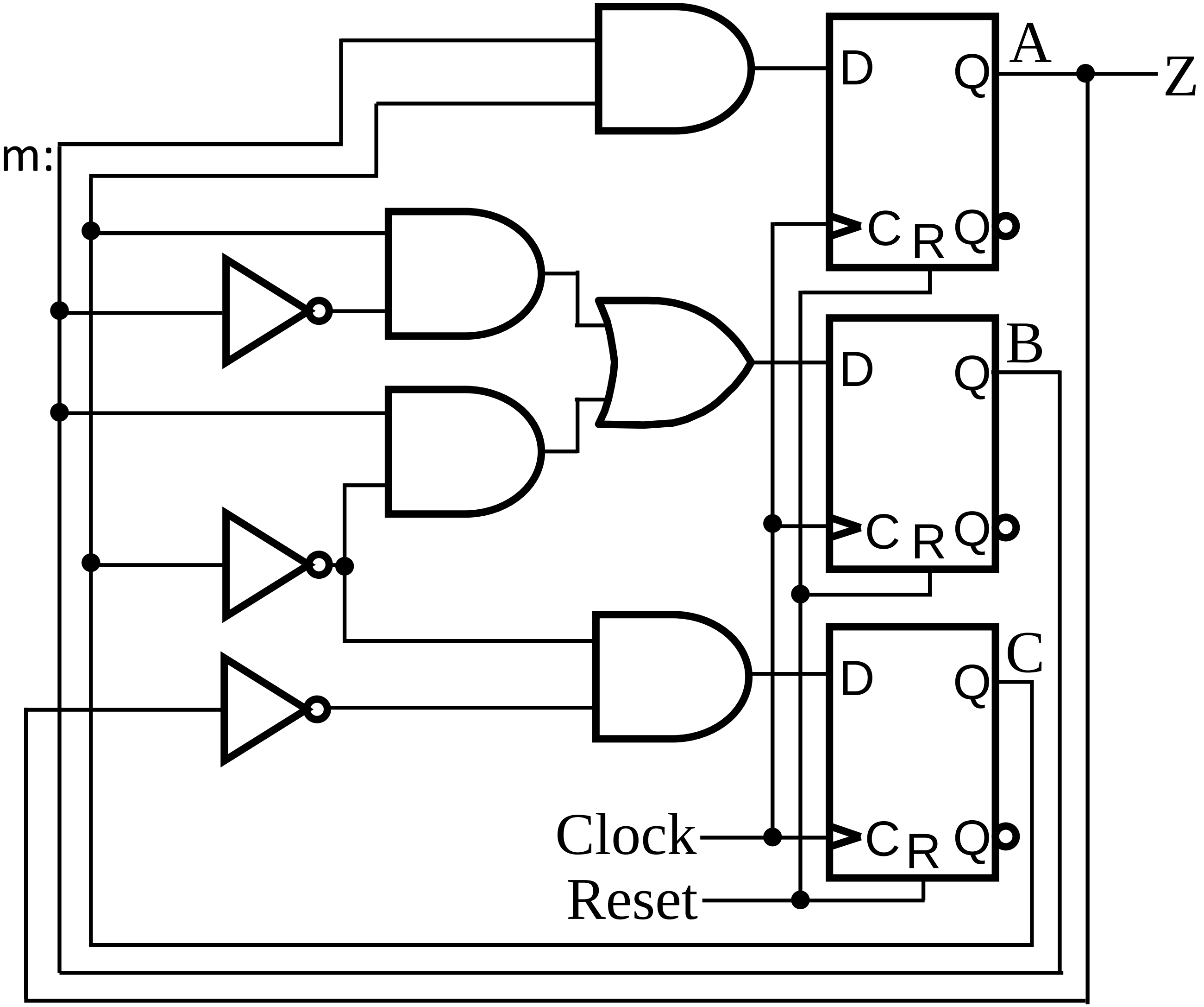
Mixed Moore and Mealy Outputs

- In real designs, some outputs may be Moore type and other outputs may be Mealy type.
- Example: Figure 5-17(a) can be modified to illustrate this
  - State 00: Moore
  - States 01, 10, and 11: Mealy
- Simplifies output specification



Example 2: Sequential Circuit Analysis

- Logic Diagram:



Example 2: Flip-Flop Input Equations

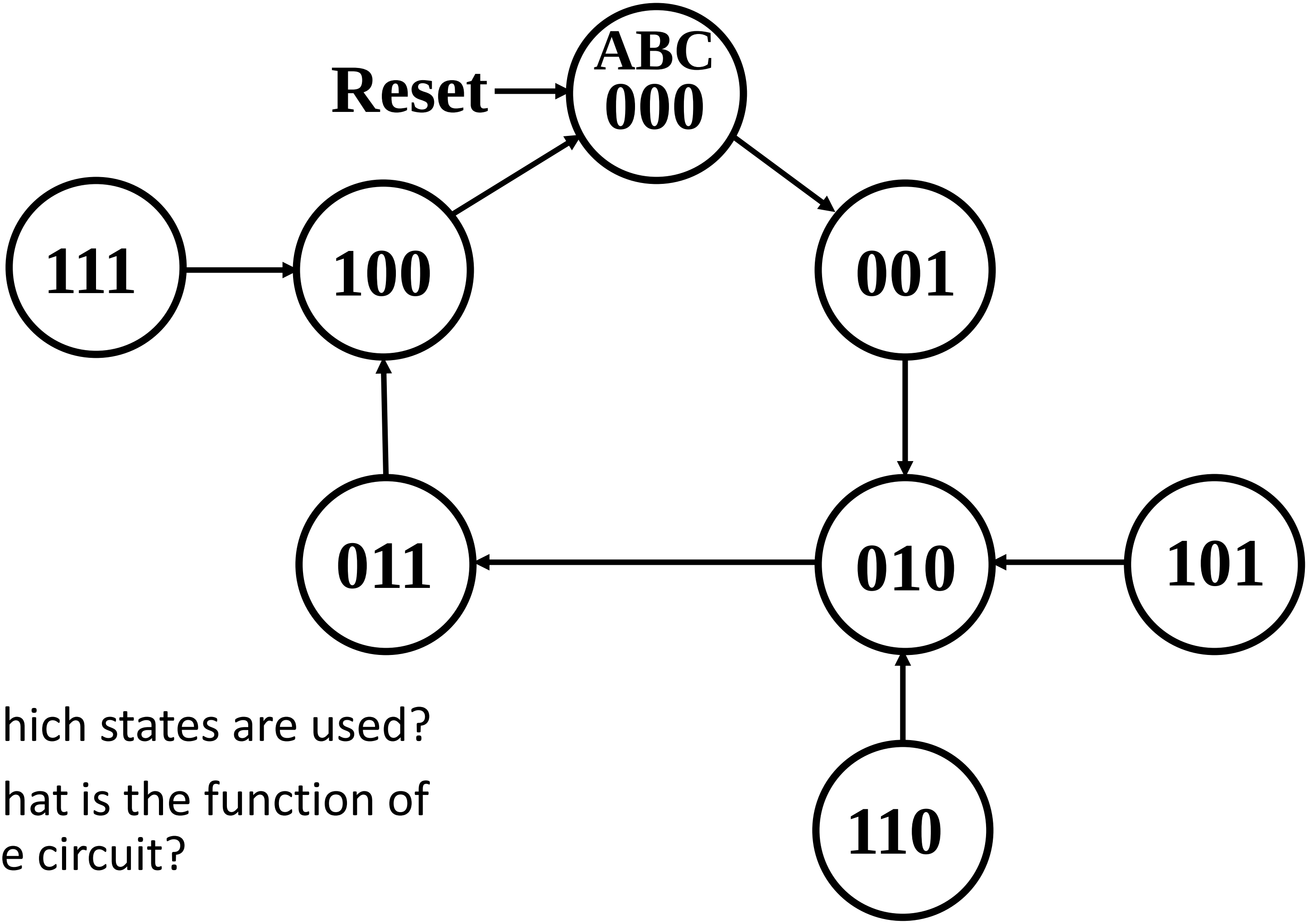
- Variables
  - Inputs: None
  - Outputs: Z
  - State Variables: A, B, C
- Initialization: Reset to (0,0,0)
- Equations
  - A(t+1) =   Z =
  - B(t+1) =
  - C(t+1) =

Example 2: State Table

$X' = X(t+1)$

A B C	A'B'C'	Z
0 0 0		
0 0 1		
0 1 0		
0 1 1		
1 0 0		
1 0 1		
1 1 0		
1 1 1		

Example 2: State Diagram



- Which states are used?
- What is the function of the circuit?

An FSM example

To introduce the design procedure for a FSM, we start by considering an example of a sequence detector. This circuit is designed to detect a particular binary sequence 101. Whenever there is a complete sequence 101, the output is equal to 1; otherwise, the output is equal to 0. In this design, each sequence 101 is considered as a separated one, *i.e.*, for the input 10101 only the first 101 is considered the detected sequence. However, for the input101101, there are two detected sequences.

The circuit has one input from which the binary stream is fed in and only one output. The state of the circuit changes at the leading edge of the clock signal. For example, consider the random binary sequence at the input as shown in Figure 5-66. The output becomes 1 only at t_7 when an exact 101 sequence is received, and at t_{10} when another 101 sequence is received. In this situation, it is necessary to keep track of the state change of the circuit since the output is not entirely dependent on the current input.

Clock	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
Input	1	1	0	0	1	0	1	1	0	1	0	1
Output	0	0	0	0	0	0	0	1	0	0	1	0

Figure 5-66 Sequence at input and corresponding output

The best way to start designing an FSM is to draw a state diagram, which transforms the conceptual specification to a pictorial representation. It can help the designer to be clearer about the state transition under different inputs. In this particular case, it requires four states to represent all states in the circuit. We call them S0 – S4, respectively. The complete state diagram is shown in Figure 5-67 in which the transitions in different states are labeled by directional arcs.

- ***State diagram***

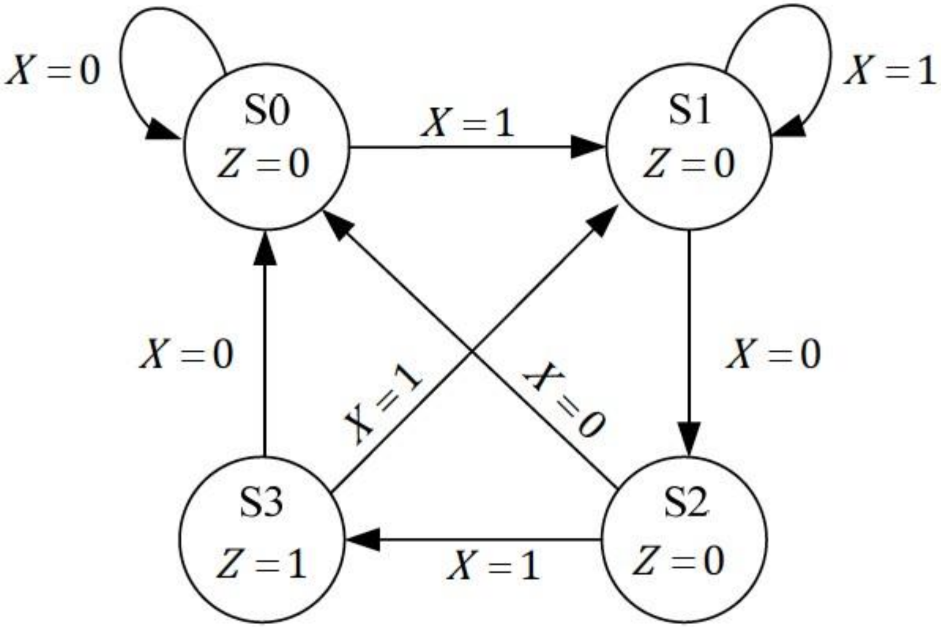
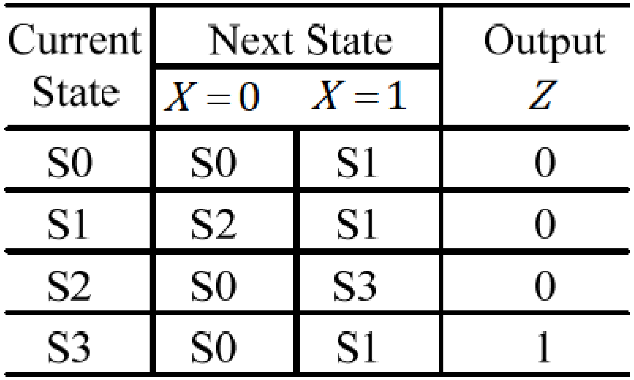


Figure 5-67 State diagram

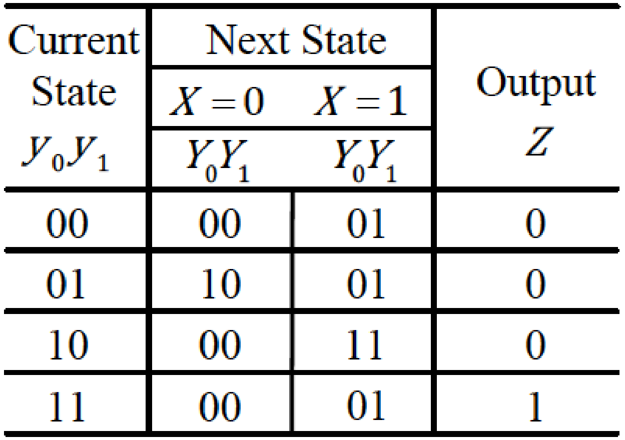
- ***State transition table***



Current State	Next State		Output
	$X = 0$	$X = 1$	Z
S0	S0	S1	0
S1	S2	S1	0
S2	S0	S3	0
S3	S0	S1	1

Figure 5-68 State table

- ***State coding***



Current State	Next State		Output
	$X=0$	$X=1$	
y_0y_1	Y_0Y_1	Y_0Y_1	Z
00	00	01	0
01	10	01	0
10	00	11	0
11	00	01	1

Figure 5-69 State assigned table

- ***Logic expression***

X \ y_0y_1	00	01	11	10
0	0	1	0	0
1	0	0	0	1

$$Y_0 = \overline{X}\,\overline{y_0}y_1 + Xy_0\overline{y_1}$$

X \ y_0y_1	00	01	11	10
0	0	0	0	0
1	1	1	1	1

$$Y_1 = X$$

y_1 \ y_0	0	1
0	0	0
1	0	1

$$Z = y_0y_1$$

Figure 5-70 Logic expression

- ***Circuit implementation***

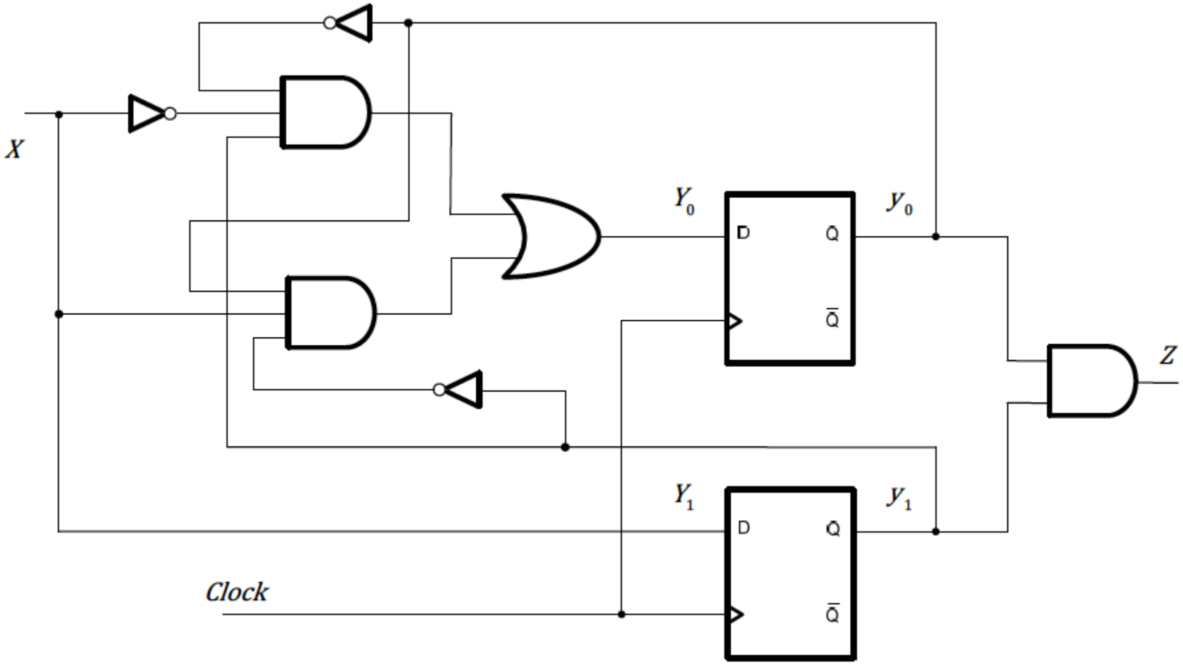


Figure 5-71 Circuit implementation of sequence detector

FSM Design Steps

Step1: Receive a design specification for the FSM and understand its requirement.

Step2: Create a state diagram according to the given specification and identify the required inputs, outputs, states and all possible state transitions.

Step3: Construct a state table and check for the redundancy states.

Step4: Determine the number of variables to represent all the needed states and make a state assignment.

Step5: Select the proper type of flip-flop and derive the logic expression.

Step6: Implement the logic circuit.

- ***FSM as the control unit for a datapath circuit***

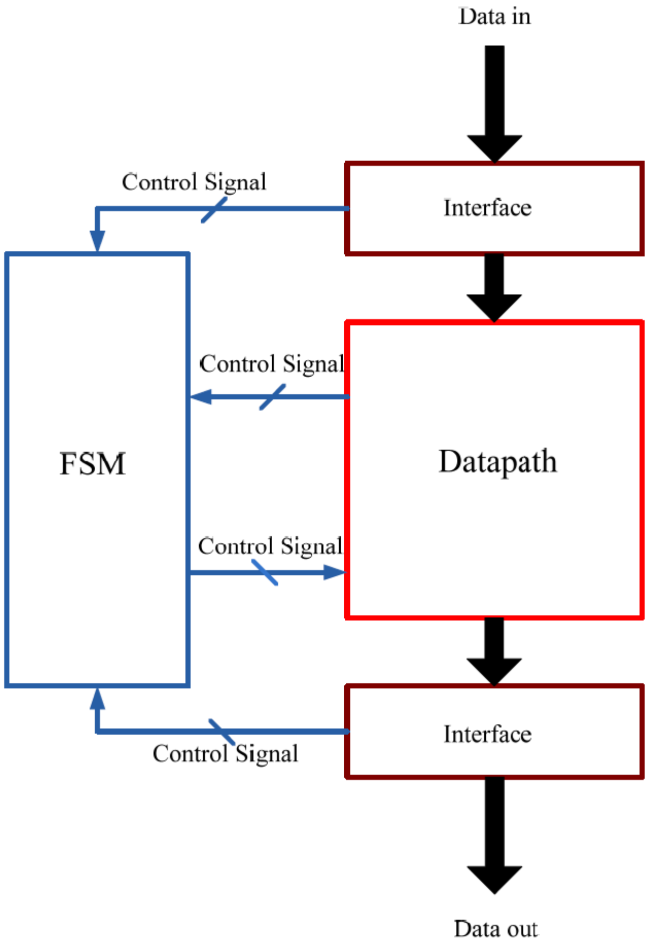


Figure 5-79 An architecture in a microprocessor

Datapath structure

- A digital processor usually consists of three parts: controller, ALU and interface.
  - As is shown in Figure 5-79, the control block is the brain of the processor and it supervises and instructs each unit working cooperatively.
  - The interface is responsible for the communication between the processor and external devices.
  - The ALU block is the core part of the processor where most of the operations are executed, and it takes large hardware resources and the most part of the area.

- ***Datapath of an FIR processor***

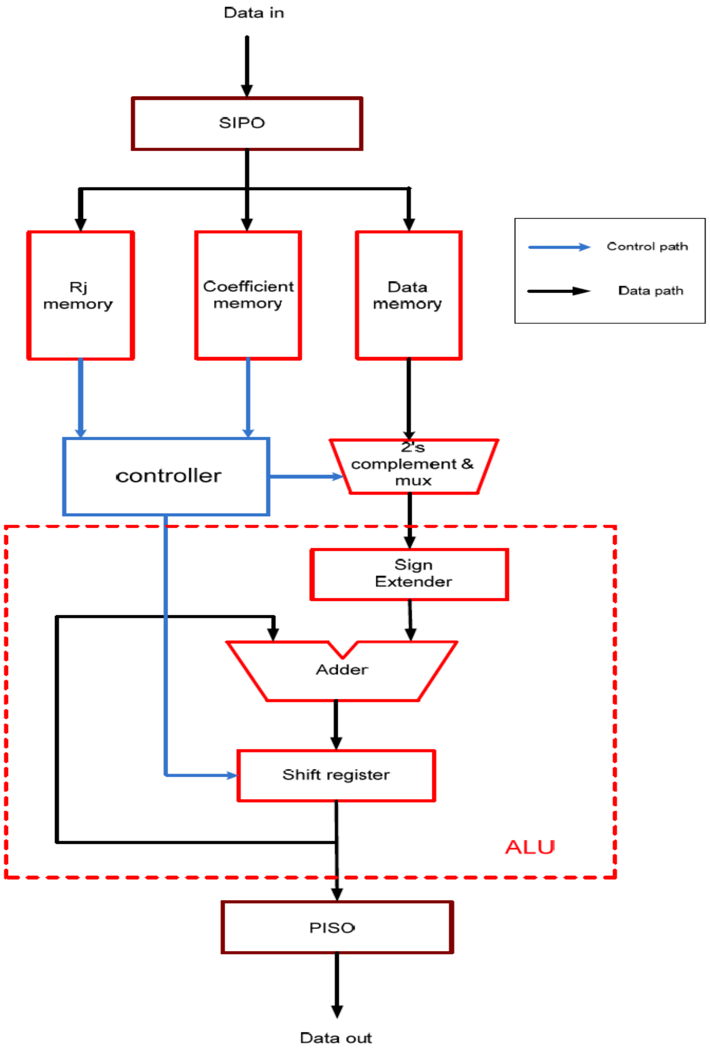


Figure 5-80 Datapath of a FIR processor

Summary

- In this lecture we summarized the basic combinational and sequential logics.
- In the discussion we have reviewed the most commonly used circuit blocks in terms of their function and structure.
- These circuits can be easily modified to fit most application needs.

– The intent of putting them here is for the self-completion of the discussion, and as a reference.