Review of Combinational and Sequential Logic Circuits and Design

9

This chapter is primarily dedicated to reviewing combinational and sequential logic circuits. Understanding and designing logic blocks are part of the integration effort to build the front-end electronics for microcontrollers.

The first part of this chapter starts with defining logic gates and the concept of truth table which then leads to the implementation of basic logic circuits. Later in the chapter, the concept of Karnaugh maps is introduced in order to minimize gate count, thereby completing the basic requirements of combinational circuit design. Following the minimization techniques, various fundamental logic blocks such as multiplexers, encoders, decoders and one-bit adders are introduced so that they can be used to construct larger scale combinational logic circuits. These include different types of adders such as ripple-carry adder, carry-look-ahead adder, carry-select adder, and the combination of all three types depending on gate count, circuit speed and power consumption. Subtractors, linear and barrel shifters, and array multipliers are also considered combinational mega cells and they will be examined in this chapter.

The definition of clock and system timing are the integral elements for sequential logic circuits. Data in a digital system moves from one storage device to the next by the virtue of a system clock. During its travel, data is routed in and out of different combinational logic blocks, and becomes modified to satisfy a specified functionality.

Therefore, in the second part of this chapter we will review the memory devices that store data, and explain the process of designing sequential circuits that require clock input. We will begin with the introduction of two basic memory elements, the latch and the flip-flop. We will explain how data travels between memory elements using timing diagrams, and analyze timing violations as a result of unexpected combinational logic delays on the data path or the clock line. Later in the chapter, we will design the basic sequential building blocks, such as registers, shift registers and counters, and show the Moore-type and the Mealy-type state machines that control data movement. We will study the advantages and disadvantages of these conventional controllers against counter-decoder type controllers in various design environments. We will introduce the concept of block memory and how it is used in a digital system at the end of this chapter. We will conclude the chapter with a comprehensive example of transferring data from one memory block to the next, the use of

A. Bindal, *Electronics for Embedded Systems*,
DOI 10.1007/978-3-319-39439-8_9

timing diagrams in the development of the design, and show how to incrementally build a data-path and a controller using timing diagrams.

9.1 Logic Gates

AND Gate

Assume that the output, OUT, in Fig. 9.1 is at logic 0 when both switches, A and B, are open. Unless both A and B close, the output stays at logic 0.

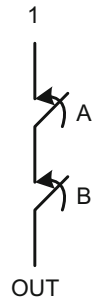


Fig. 9.1 Switch representation of a two-input AND gate

A two-input AND gate functions similarly to the circuit in Fig. 9.1. If any two inputs, A and B, of the AND gate in Fig. 9.2 are at logic 0, the gate produces an output, OUT, at logic 0. Both inputs of the gate must be equal to logic 1 in order to produce an output at logic 1. This behavior is tabulated in Table 9.1, which is called a "truth table".

Fig. 9.2 Two-input AND gate symbol

Table 9.1 Two-input AND gate truth table

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

The functional representation of the two-input AND gate is:

$$OUT = A \,.\, B$$

Here, the symbol "." between inputs A and B represents the AND-function.

OR Gate

Now, assume a parallel connectivity between switches A and B as shown in Fig. 9.3. OUT becomes to logic 1 if any of the switches close; otherwise the output will stay at logic 0.

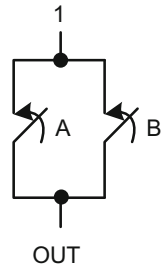


Fig. 9.3 Switch representation of two-input OR gate

A two-input OR gate shown in Fig. 9.4 also functions similarly to the circuit in Fig. 9.3. If any two inputs are at logic 1, the gate produces an output, OUT, at logic 1. Both inputs of the gate must be equal to logic 0 in order to produce an output at logic 0. This behavior is tabulated in the truth table, Table 9.2.

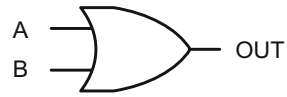


Fig. 9.4 Two-input OR gate symbol

Table 9.2 Two-input OR gate truth table

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

The functional representation of the two-input OR gate is:

$$\text{OUT} = \text{A} + \text{B}$$

where, the symbol "+" between inputs A and B signifies the OR-function.

Exclusive OR Gate

A two-input Exclusive OR gate, XOR gate, is shown in Fig. 9.5. The XOR gate produces a logic 0 output if both inputs are equal. Therefore, in many logic applications this gate is used to compare the input logic levels to see if they are equal. The functional behavior of the gate is tabulated in Table 9.3.

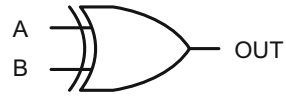


Fig. 9.5 Two-input XOR gate symbol

Table 9.3 Two-input XOR gate truth table

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

The functional representation of the two-input XOR gate is:

$$\text{OUT} = \text{A} \oplus \text{B}$$

where, the symbol "⊕" between inputs A and B signifies the XOR function.

Buffer

A buffer is a single input device whose output is logically equal to its input. The only use of this gate is to be able to supply current to the cumulative capacitive loading of logic gates at the output node. The logical representation of this gate is shown in Fig. 9.6.

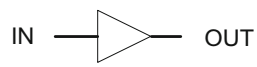


Fig. 9.6 Buffer symbol

Complementary Logic Gates

All basic logic gates need to have complemented forms. If a single input needs to be complemented, an inverter shown in Fig. 9.7 is used. The inverter truth table is shown in Table 9.4.

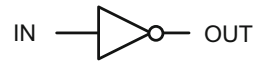


Fig. 9.7 Inverter symbol

Table 9.4 Inverter truth table

IN	OUT
0	1
1	0

The functional representation of the inverter is:

$$\text{OUT} = \overline{\text{IN}}$$

where, the symbol "−" on top of the input, IN, represents the complement-function.

The complemented form of two-input AND gate is called two-input NAND gate, where "N" signifies negation. The logic representation is shown in Fig. 9.8, where a circle at the output of the gate means complemented output. The truth table of this gate is shown in Table 9.5. Note that all output values in this table are exact opposites of the values given in Table 9.1.

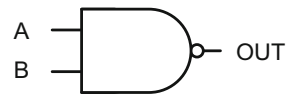


Fig. 9.8 Two-input NAND gate symbol

Table 9.5 Two-input NAND gate truth table

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

The functional representation of the two-input NAND gate is:

$$\text{OUT} = \overline{\text{A} \cdot \text{B}}$$

Similar to the NAND gate, two-input OR and XOR gates have complemented configurations, called two-input NOR and XNOR gates, respectively.

The symbolic representation and truth table of a two-input NOR gate is shown in Fig. 9.9 and Table 9.6, respectively. Again, all the outputs in Table 9.6 are exact complements of Table 9.2.

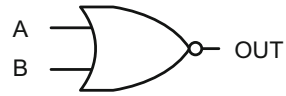


Fig. 9.9 Two-input NOR gate symbol

Table 9.6 Two-input NOR gate truth table

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

The functional representation of the two-input NOR gate is:

$$\text{OUT} = \overline{\text{A} + \text{B}}$$

The symbolic representation and truth table of a two-input XNOR gate is shown in Fig. 9.10 and Table 9.7, respectively. This gate, like its counterpart two-input XOR gate, is often used to detect if input logic levels are equal.

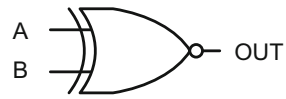


Fig. 9.10 Two-input XNOR gate symbol

Table 9.7 Two-input XNOR gate truth table

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

The functional representation of the two-input XNOR gate is:

$$\text{OUT} = \overline{\text{A} \oplus \text{B}}$$

Tri-State Buffer and Inverter

It is often necessary to create an open circuit between the input and the output of a logic gate if the gate is not enabled. This need creates two more basic logic gates, the tri-state buffer and tri-state inverter.

The tri-state buffer is shown in Fig. 9.11. Its truth table in Table 9.8 indicates continuity between the input and the output terminals if the control input, EN, is at logic 1; when EN is lowered to logic 0, an open circuit exists between IN and OUT, which is defined as a high impedance, HiZ, condition at the output terminal.

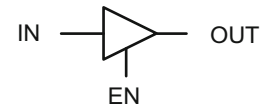


Fig. 9.11 Tri-state buffer symbol

Table 9.8 Tri-state buffer truth table

EN	IN	OUT
0	0	HiZ
0	1	HiZ
1	0	0
1	1	1

The tri-state inverter is shown in Fig. 9.12 along with its truth table in Table 9.9. This gate behaves like an inverter when EN input is at logic 1; however, if EN is lowered to logic 0, its output disconnects from its input.

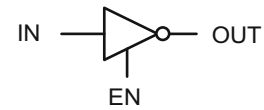


Fig. 9.12 Tri-state inverter symbol

Table 9.9 Tri-state inverter truth table

EN	IN	OUT
0	0	HiZ
0	1	HiZ
1	0	1
1	1	0

The control input, EN, to tri-state buffer and inverter can also be complemented in order to produce an active-low enabling scheme.

The tri-state buffer with the active-low enable input in Fig. 9.13 creates continuity when EN = 0.

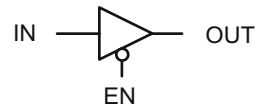


Fig. 9.13 Tri-state buffer symbol with complemented enable input

The tri-state inverter with the active-low input in Fig. 9.14 also functions like an inverter when EN is at logic 0, but its output becomes HiZ when EN is changed to logic 1.

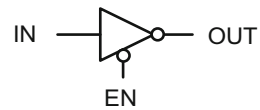


Fig. 9.14 Tri-state inverter symbol with complemented enable input

9.2 Boolean Algebra

It is essential to be able to reconfigure logic gates to suit our design goals. Logical reconfigurations may be as simple as re-grouping the inputs of a single gate or complementing the inputs of several gates to reach a design objective.

Identity, commutative, associative, distributive laws and DeMorgan's negation rules are used to perform logical manipulations. Table 9.10 tabulates these laws.

Table 9.10 Identity, commutative, associative, distributive and DeMorgan's rules

$A \cdot 1 = A$ $A \cdot 0 = 0$ $A \cdot A = A$ $A \cdot \overline{A} = 0$ $A + 1 = 1$ $A + 0 = A$ $A + A = A$ $A + \overline{A} = 1$ $\overline{\overline{A}} = A$	Identity
$A \cdot B = B \cdot A$ $A + B = B + A$	Commutative
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$ $A + (B + C) = (A + B) + C$	Associative
$A \cdot (B + C) = A \cdot B + A \cdot C$ $A + B \cdot C = (A + B) \cdot (A + C)$	Distributive
$\overline{A \cdot B} = \overline{A} + \overline{B}$ $\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's

Example 9.1 Reduce $\mathrm{OUT} = \mathrm{A} \cdot \overline{\mathrm{B}} \cdot \mathrm{C} + \mathrm{A} \cdot \mathrm{B} \cdot \mathrm{C} + \mathrm{A} \cdot \overline{\mathrm{B}}$ using algebraic rules.

$$\begin{aligned}\mathrm{OUT} &= \mathrm{A} \cdot \overline{\mathrm{B}} \cdot \mathrm{C} + \mathrm{A} \cdot \mathrm{B} \cdot \mathrm{C} + \mathrm{A} \cdot \overline{\mathrm{B}} \\ &= \mathrm{A} \cdot \mathrm{C} \cdot (\overline{\mathrm{B}} + \mathrm{B}) + \mathrm{A} \cdot \overline{\mathrm{B}} \\ &= \mathrm{A} \cdot (\mathrm{C} + \overline{\mathrm{B}})\end{aligned}$$

Example 9.2 Reduce $\mathrm{OUT} = \mathrm{A} + \overline{\mathrm{A}} \cdot \mathrm{B}$ using algebraic rules.

$$\begin{aligned}\mathrm{OUT} &= \mathrm{A} + \overline{\mathrm{A}} \cdot \mathrm{B} \\ &= (\mathrm{A} + \overline{\mathrm{A}}) \cdot (\mathrm{A} + \mathrm{B}) \\ &= \mathrm{A} + \mathrm{B}\end{aligned}$$

Example 9.3 Reduce $\mathrm{OUT} = \mathrm{A}\,.\,\mathrm{B} + \overline{\mathrm{A}}\,.\,\mathrm{C} + \mathrm{B}\,.\,\mathrm{C}$ using algebraic rules.

$$\begin{aligned}\mathrm{OUT} &= \mathrm{A}\,.\,\mathrm{B} + \overline{\mathrm{A}}\,.\,\mathrm{C} + \mathrm{B}\,.\,\mathrm{C} \\ &= \mathrm{A}\,.\,\mathrm{B} + \overline{\mathrm{A}}\,.\,\mathrm{C} + \mathrm{B}\,.\,\mathrm{C}\,.\,(\mathrm{A}\,.\,\overline{\mathrm{A}}) \\ &= \mathrm{A}\,.\,\mathrm{B} + \overline{\mathrm{A}}\,.\,\mathrm{C} + \mathrm{A}\,.\,\mathrm{B}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C} \\ &= \mathrm{A}\,.\,\mathrm{B}\,.\,(1 + \mathrm{C}) + \overline{\mathrm{A}}\,.\,\mathrm{C}\,.\,(1 + \mathrm{B}) \\ &= \mathrm{A}\,.\,\mathrm{B} + \overline{\mathrm{A}}\,.\,\mathrm{C}\end{aligned}$$

Example 9.4 Reduce $\mathrm{OUT} = (\mathrm{A} + \mathrm{B})\,.\,(\overline{\mathrm{A}} + \mathrm{C})$ using algebraic rules.

$$\begin{aligned}\mathrm{OUT} &= (\mathrm{A} + \mathrm{B})\,.\,(\overline{\mathrm{A}} + \mathrm{C}) \\ &= \mathrm{A}\,.\,\overline{\mathrm{A}} + \mathrm{A}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B} + \mathrm{B}\,.\,\mathrm{C} \\ &= \mathrm{A}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B} + \mathrm{B}\,.\,\mathrm{C} \\ &= \mathrm{A}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B} + \mathrm{B}\,.\,\mathrm{C}\,.\,(\mathrm{A} + \overline{\mathrm{A}}) \\ &= \mathrm{A}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B} + \mathrm{A}\,.\,\mathrm{B}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C} \\ &= \mathrm{A}\,.\,\mathrm{C}\,.\,(1 + \mathrm{B}) + \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,(1 + \mathrm{C}) \\ &= \mathrm{A}\,.\,\mathrm{C} + \overline{\mathrm{A}}\,.\,\mathrm{B}\end{aligned}$$

Example 9.5 Convert OUT = (A + B) . $\overline{\mathrm{C}\,.\,\mathrm{D}}$ into an OR-combination of two-input AND gates using algebraic laws and DeMorgan's theorem.

$$\begin{aligned}\mathrm{OUT} &= (\mathrm{A} + \mathrm{B})\,.\,\overline{\mathrm{C}\,.\,\mathrm{D}} \\ &= (\mathrm{A} + \mathrm{B})\,.\,(\overline{\mathrm{C}} + \overline{\mathrm{D}}) \\ &= \mathrm{A}\,.\,\overline{\mathrm{C}} + \mathrm{A}\,.\,\overline{\mathrm{D}} + \mathrm{B}\,.\,\overline{\mathrm{C}} + \mathrm{B}\,.\,\overline{\mathrm{D}}\end{aligned}$$

Example 9.6 Convert OUT = A . B + C . D into an AND-combination of two-input OR gates using algebraic laws and DeMorgan's theorem.

$$\begin{aligned}\mathrm{OUT} &= \mathrm{A}\,.\,\mathrm{B} + \mathrm{C}\,.\,\mathrm{D} \\ &= \overline{\overline{\mathrm{A}\,.\,\mathrm{B} + \mathrm{C}\,.\,\mathrm{D}}} \\ &= \left(\overline{\overline{\mathrm{A}} + \overline{\mathrm{B}})\,.\,(\overline{\mathrm{C}} + \overline{\mathrm{D}}}\right)\end{aligned}$$

9.3 Designing Combinational Circuits Using Truth Tables

A combinational circuit is cascaded form of basic logic gates without any feedback from the output to any input. The logic function is obtained from a truth table that specifies the complete functionality of the digital circuit.

Example 9.7 Using the truth table given in Table 9.11 determine the output function of the digital circuit.

Table 9.11 An arbitrary truth table with four inputs

A	B	C	D	OUT
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

The output function can be expressed either as the OR combination of AND gates or the AND combination of OR gates.

If the output is expressed in terms of AND gates, all output entries that are equal to one in the truth table must be grouped together as a single OR gate.

$$\begin{aligned}\text{OUT} = {} & \overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\overline{\text{C}}\,.\,\overline{\text{D}} + \overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\overline{\text{C}}\,.\,\text{D} + \overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\text{C}\,.\,\overline{\text{D}} + \overline{\text{A}}\,.\,\text{B}\,.\,\overline{\text{C}}\,.\,\text{D} \\ & + \text{A}\,.\,\overline{\text{B}}\,.\,\overline{\text{C}}\,.\,\overline{\text{D}} + \text{A}\,.\,\overline{\text{B}}\,.\,\overline{\text{C}}\,.\,\text{D} + \text{A}\,.\,\overline{\text{B}}\,.\,\text{C}\,.\,\overline{\text{D}}\end{aligned}$$

This expression is called the Sum Of Products (SOP), and it contains seven terms each of which is called a "minterm". In the first minterm, $\overline{A} \,.\, \overline{B} \,.\, \overline{C} \,.\, \overline{D}$, each A, B, C and D input is complemented to produce OUT = 1 for the A = B = C = D = 0 entry of the truth table. Each of the remaining six minterms also complies with producing OUT = 1 for their respective input entries.

The resulting combinational circuit is shown in Fig. 9.15.

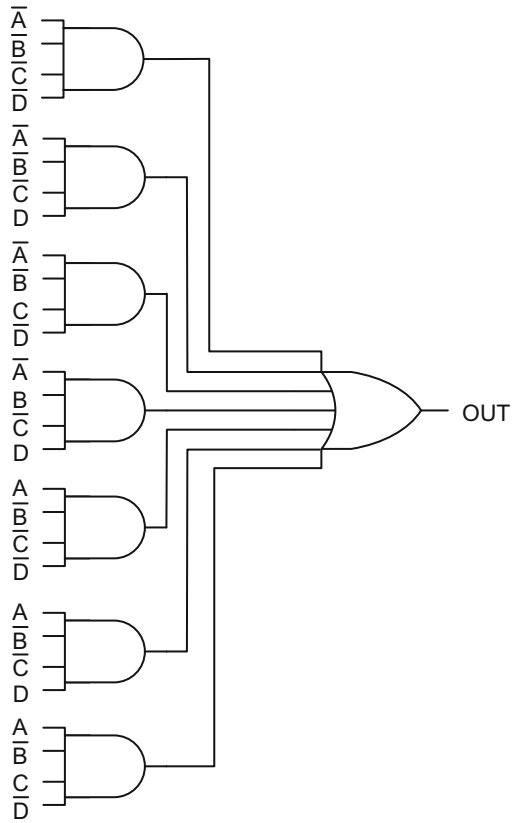


Fig. 9.15 AND-OR logic representation of the truth table in Table 9.11

If the output function needs to be expressed in terms of OR gates, all the output entries that are equal to zero in the truth table must be grouped as a single AND gate.

$$\begin{aligned} \text{OUT} = {} & (A + B + \overline{C} + \overline{D}) \,.\, (A + \overline{B} + C + D) \,.\, (A + \overline{B} + \overline{C} + D) \\ & .\, (A + \overline{B} + \overline{C} + \overline{D}) \,.\, (\overline{A} + B + \overline{C} + \overline{D}) \,.\, (\overline{A} + \overline{B} + C + D) \\ & .\, (\overline{A} + \overline{B} + C + \overline{D}) \,.\, (\overline{A} + \overline{B} + \overline{C} + D) \,.\, (\overline{A} + \overline{B} + \overline{C} + \overline{D}) \end{aligned}$$

This expression is called the Product Of Sums (POS), and it contains nine terms each of which is called a "maxterm". The first maxterm, $A + B + \overline{C} + \overline{D}$, produces OUT = 0 for the

ABCD = 0011 entry of the truth table. Since the output is formed with a nine-input AND gate, the values of the other maxterms do not matter to produce OUT = 0. Each of the remaining eight maxterms generates OUT = 0 for their corresponding truth table input entries.

The resulting combinational circuit is shown in Fig. 9.16.

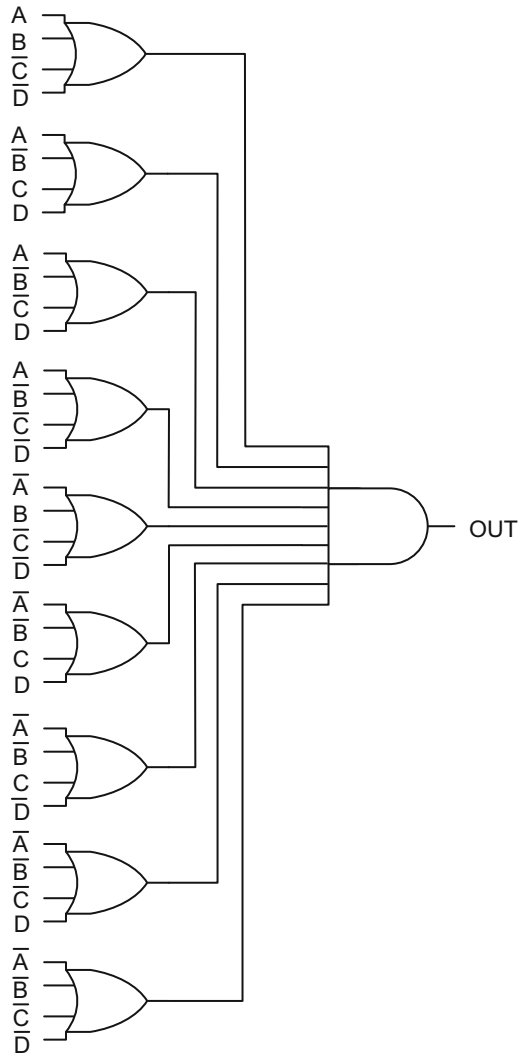


Fig. 9.16 OR-AND logic representation of the truth table in Table 9.11

9.4 Combinational Logic Minimization—Karnaugh Maps

One of the most useful tools in logic design is the use of Karnaugh maps (K-map) to minimize combinational logic functions.

Minimization can be performed in two ways. To obtain SOP form of a minimized logic function, the entries with logic 1 in the truth table must be grouped together in the K-map. To obtain POS form of a minimized logic function, the entries with logic 0 must be grouped together in the K-map.

Example 9.8 Using the truth table in Table 9.12, determine the minimized SOP and POS output functions. Prove them to be identical.

Table 9.12 An arbitrary truth table with three inputs

A	B	C	OUT
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

The K-map formed according to the truth table groups 1s to obtain the minimized output function, OUT, in SOP form in Fig. 9.17.

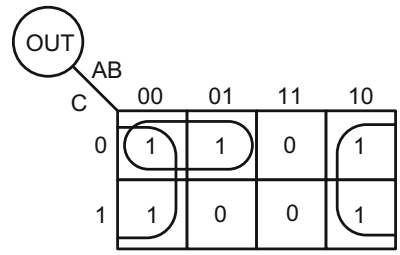


Fig. 9.17 K-map of the truth table in Table 9.12 to determine SOP

Grouping 1s takes place among neighboring boxes in the K-map where only one variable is allowed to change at a time. For instance, the first grouping of 1s combines ABC = 000 and ABC = 010 as they are in neighboring boxes. Only B changes from logic 0 to logic 1 while A and C stay constant at logic 0. To obtain OUT = 1, both A and C need to be complemented; this produces the first term, $\overline{A}.\overline{C}$, for the output function. Similarly, the second grouping of 1s combines the neighboring boxes, ABC = 000, 001, 100 and 101, where both A and C change while B stays constant at logic 0. To obtain OUT = 1, B needs to be complemented; this generates the second term, $\overline{B}$, for the output function.

This means that either the term $\overline{A}.\overline{C}$ or $\overline{B}$ makes OUT equal to logic 1. Therefore, the minimized output function, OUT, in the SOP form is:

$$OUT = \overline{B} + \overline{A}.\overline{C}$$

Grouping 0s produces the minimized POS output function as shown in Fig. 9.18.

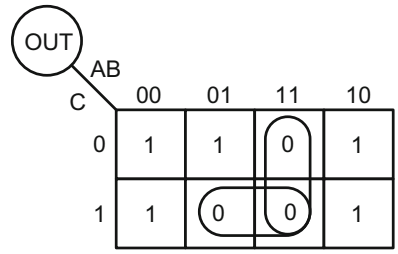


Fig. 9.18 K-map of the truth table in Table 9.12 to determine POS

This time, the first grouping of 0s combines the boxes, ABC = 011 and 111, where A changes from logic 0 to logic 1 while B and C stay constant at logic 1. This grouping targets OUT = 0, which requires both B and C to be complemented. As a result, the first term of the output function, $\overline{B} + \overline{C}$, is generated. The second grouping combines ABC = 110 and 111 where C changes value while A and B are equal to logic 1. To obtain OUT = 0, both A and B need to be complemented. Consequently, the second term, $\overline{A} + \overline{B}$, is generated.

Either of the terms $\overline{B} + \overline{C}$ or $\overline{A} + \overline{B}$ makes OUT = 0. Therefore, the minimized output function in the POS form becomes:

$$OUT = (\overline{B} + \overline{C}).(\overline{A} + \overline{B})$$

To find out if the SOP and POS forms are identical, one can manipulate the POS using the algebraic rules given earlier.

$$\begin{aligned} OUT &= (\overline{B} + \overline{C}).(\overline{A} + \overline{B}) \\ &= \overline{A}.\overline{B} + \overline{B}.\overline{B} + \overline{A}.\overline{C} + \overline{B}.\overline{C} \\ &= \overline{A}.\overline{B} + \overline{B} + \overline{A}.\overline{C} + \overline{B}.\overline{C} \\ &= \overline{B}.(\overline{A} + 1 + \overline{C}) + \overline{A}.\overline{C} \\ &= \overline{B} + \overline{A}.\overline{C} \end{aligned}$$

However, this is the SOP form of the output function derived above.

Example 9.9 Using the truth table in Example 9.7 determine the minimized SOP and POS output functions.

To obtain an output function in SOP form, 1s in the K-map in Fig. 9.19 is grouped together as shown below.

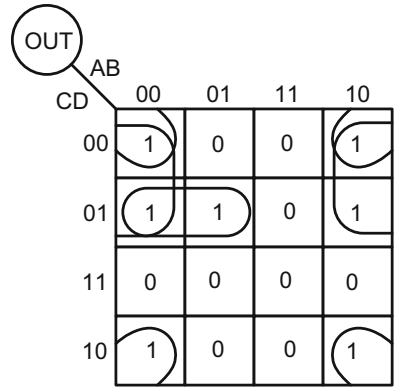


Fig. 9.19 K-map of the truth table in Table 9.11 to determine SOP

The minimized output function contains only three minterms compared to seven minterms in Example 9.7. Also, the minterms are reduced to groups of two or three inputs instead of four.

$$\text{OUT} = \overline{\text{B}} \,.\, \overline{\text{C}} + \overline{\text{A}} \,.\, \overline{\text{C}} \,.\, \text{D} + \overline{\text{B}} \,.\, \overline{\text{D}}$$

The resultant combinational circuit is shown in Fig. 9.20.

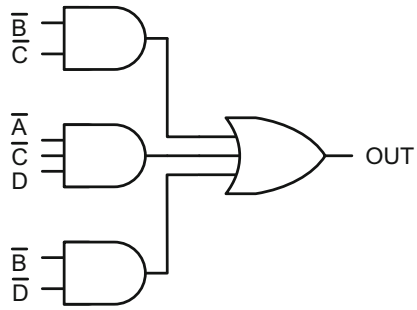


Fig. 9.20 Minimized logic circuit in SOP form from the K-map in Fig. 9.19

Further minimization can be achieved algebraically, which then reduces the number of terms from three to two.

$$\text{OUT} = \overline{\text{B}} \,.\, (\overline{\text{C}} + \overline{\text{D}}) \; + \; \overline{\text{A}} \,.\, \overline{\text{C}} \,.\, \text{D}$$

The corresponding combinational circuit is shown in Fig. 9.21.

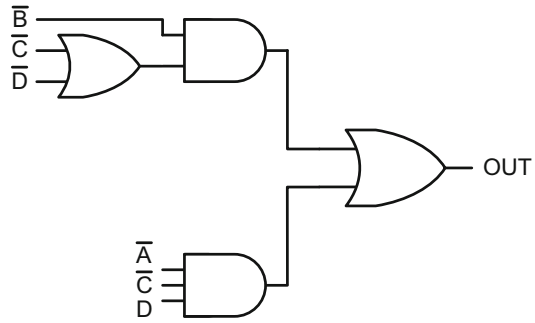


Fig. 9.21 Logic circuit in Fig. 9.20 after algebraic minimizations are applied

To obtain a POS output function, 0s are grouped together as shown in Fig. 9.22.

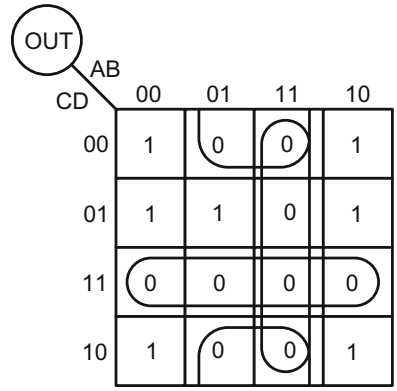


Fig. 9.22 K-map of the truth table in Table 9.11 to determine POS

The minimized output function contains only three maxterms compared to nine in Example 9.7. Also, the maxterms are reduced to groups of two inputs instead of four.

$$\text{OUT} = (\overline{\text{C}} + \overline{\text{D}}) \,.\, (\overline{\text{A}} + \overline{\text{B}}) \,.\, (\overline{\text{B}} + \text{D})$$

The resultant combinational circuit is shown in Fig. 9.23.

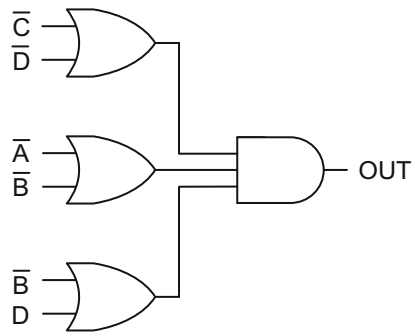


Fig. 9.23 Minimized logic circuit in POS form from the K-map in Fig. 9.22

Example 9.10 Determine if the minimized SOP and POS output functions in Example 9.9 are identical to each other.

Rewriting the POS form of OUT from Example 9.9 and using the algebraic laws shown earlier, this expression can be re-written as:

$$\begin{aligned}\text{OUT} &= (\overline{\text{C}}+\overline{\text{D}})\,.\,(\overline{\text{A}}+\overline{\text{B}})\,.\,(\overline{\text{B}}+\text{D}) \\ &= (\overline{\text{A}}\,.\,\overline{\text{C}}+\overline{\text{A}}\,.\,\overline{\text{D}}+\overline{\text{B}}\,.\,\overline{\text{C}}+\overline{\text{B}}\,.\,\overline{\text{D}})\,.\,(\overline{\text{B}}+\text{D}) \\ &= \overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\overline{\text{C}}+\overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\overline{\text{D}}+\overline{\text{B}}\,.\,\overline{\text{C}}+\overline{\text{B}}\,.\,\overline{\text{D}} \\ &+\overline{\text{A}}\,.\,\overline{\text{C}}\,.\,\text{D}+\overline{\text{B}}\,.\,\overline{\text{C}}\,.\,\text{D} \\ &= \overline{\text{B}}\,.\,\overline{\text{C}}+\overline{\text{B}}\,.\,\overline{\text{D}}+\overline{\text{A}}\,.\,\overline{\text{C}}\,.\,\text{D}\end{aligned}$$

The result is identical to the SOP expression given in Example 9.9.

Example 9.11 Determine the minimal SOP and POS forms of the output function, OUT, from the K-map in Fig. 9.24. Note that the "X" sign corresponds to a "don't care" condition that represents either logic 0 or logic 1.

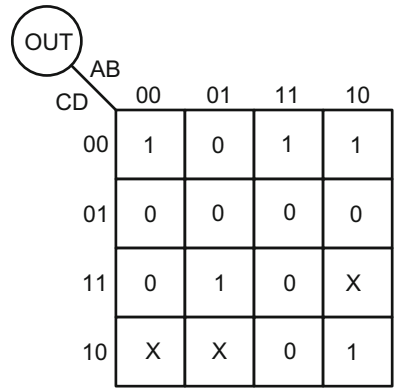


Fig. 9.24 An arbitrary K-map with "don't care" entries

For SOP, we group 1s in the K-map in Fig. 9.25. Boxes with "don't care" are used as 1s to achieve a minimal SOP expression.

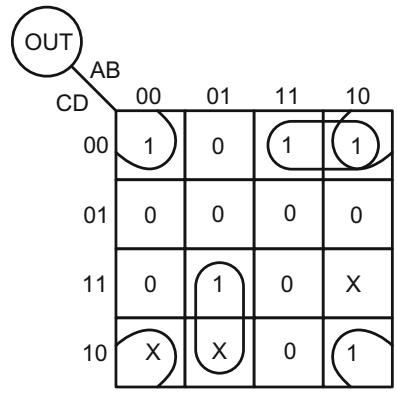


Fig. 9.25 Grouping to determine SOP form for the K-map in Fig. 9.24

The SOP functional expression for OUT is:

$$\text{OUT} = \text{A} \,.\, \overline{\text{C}} \,.\, \overline{\text{D}} + \overline{\text{A}} \,.\, \text{B} \,.\, \text{C} + \overline{\text{B}} \,.\, \overline{\text{D}}$$

For POS, we group 0s in the K-map in Fig. 9.26. Boxes with "don't care" symbols are used as 0s to achieve a minimal POS expression.

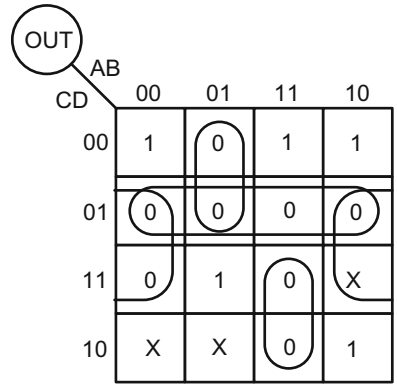


Fig. 9.26 Grouping to determine POS form for the K-map in Fig. 9.24

The POS functional expression for OUT is:

$$\text{OUT} = (\text{C} + \overline{\text{D}}) \,.\, (\text{B} + \overline{\text{D}}) \,.\, (\text{A} + \overline{\text{B}} + \text{C}) \,.\, (\overline{\text{A}} + \overline{\text{B}} + \overline{\text{C}})$$

To show that the SOP and POS expressions are identical, we start with the POS expression using the algebraic manipulations described earlier in Table 9.10.

$$
\begin{aligned}
\mathrm{OUT} &= (\mathrm{C}+\overline{\mathrm{D}})\,.\,(\mathrm{B}+\overline{\mathrm{D}})\,.\,(\mathrm{A}+\overline{\mathrm{B}}+\mathrm{C})\,.\,(\overline{\mathrm{A}}+\overline{\mathrm{B}}+\overline{\mathrm{C}}) \\
&= (\mathrm{B}\,.\,\mathrm{C}\ +\ \mathrm{C}\,.\,\overline{\mathrm{D}}+\mathrm{B}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{D}})\,.\,(\mathrm{A}\,.\,\overline{\mathrm{B}}+\mathrm{A}\,.\,\overline{\mathrm{C}}+\overline{\mathrm{A}}\,.\,\overline{\mathrm{B}}+\overline{\mathrm{B}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{C}}+\overline{\mathrm{A}}\,.\,\mathrm{C}+\overline{\mathrm{B}}\,.\,\mathrm{C}) \\
&= (\mathrm{B}\,.\,\mathrm{C}\ +\overline{\mathrm{D}})\,.\,(\mathrm{A}\,.\,\overline{\mathrm{C}}+\overline{\mathrm{A}}\,.\,\mathrm{C}+\overline{\mathrm{B}}) \\
&= \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\ +\ \mathrm{A}\,.\,\overline{\mathrm{C}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{A}}\,.\,\mathrm{C}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{D}} \\
&= \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\ +\ \mathrm{A}\,.\,\overline{\mathrm{C}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{A}}\,.\,\mathrm{C}\,.\,\overline{\mathrm{D}}\,.\,(\mathrm{B}+\overline{\mathrm{B}}) \\
&= \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\ +\ \mathrm{A}\,.\,\overline{\mathrm{C}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{A}}\,.\,\overline{\mathrm{B}}\,.\,\mathrm{C}\,.\,\overline{\mathrm{D}} \\
&= \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\,.\,(1+\overline{\mathrm{D}})+\mathrm{A}\,.\,\overline{\mathrm{C}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{D}}\,.\,(1+\overline{\mathrm{A}}\,.\,\mathrm{C}) \\
&= \overline{\mathrm{A}}\,.\,\mathrm{B}\,.\,\mathrm{C}\ +\ \mathrm{A}\,.\,\overline{\mathrm{C}}\,.\,\overline{\mathrm{D}}+\overline{\mathrm{B}}\,.\,\overline{\mathrm{D}}
\end{aligned}
$$

The result is identical to the minimal SOP expression for OUT shown above.

9.5 Basic Logic Blocks

2-1 Multiplexer

A 2-1 multiplexer (MUX) is one of the most versatile logic elements in logic design. It is defined as follows:

$$
\mathrm{OUT} = \begin{bmatrix} \mathrm{A} & \text{if} & \text{sel = 1} \\ \mathrm{B} & \text{else} & \end{bmatrix}
$$

A functional diagram of the 2-1 MUX is given in Fig. 9.27. According to the functional description of this device, when sel = 1 input A is passed through the device to become its output. When sel = 0 input B is passed through the device to become its output.

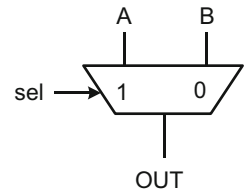


Fig. 9.27 2-1 MUX symbol

According to this definition, the truth table in Table 9.13 can be formed:

Table 9.13 2-1 MUX truth table

sel	A	B	OUT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Now, let us transfer the output values from the truth table to the K-map in Fig. 9.28.

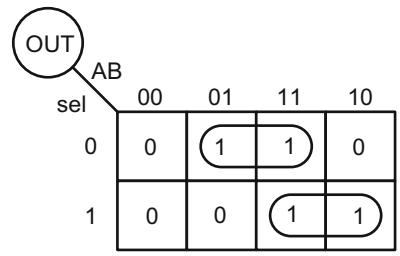


Fig. 9.28 2-1 MUX K-map

Grouping 1s in the K-map reveals the minimal output function of the 2-1 MUX in SOP form:

$$\text{OUT} = \text{sel} \,.\, \text{A} + \overline{\text{sel}} \,.\, \text{B}$$

The corresponding combinational circuit is shown in Fig. 9.29.

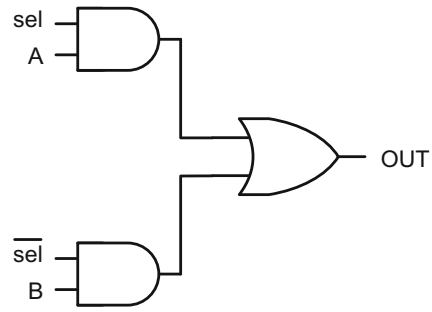


Fig. 9.29 2-1 MUX logic circuit

4-1 Multiplexer

Considering A, B, C and D are the inputs, the functional description of a 4-1 MUX becomes as follows:

$$\text{OUT} = \begin{cases} \text{A} & \text{if sel1} = 0 \quad \text{and} \quad \text{sel2} = 0 \\ \text{B} & \text{if sel1} = 0 \quad \text{and} \quad \text{sel2} = 1 \\ \text{C} & \text{if sel1} = 1 \quad \text{and} \quad \text{sel2} = 0 \\ \text{D} & \text{else} \end{cases}$$

According to this description, we can form a truth table and obtain the minimal SOP or POS expression for OUT. However, it is quite easy to decipher the SOP expression for OUT from the description above. The AND-combination of A, complemented sel1 and complemented sel2 inputs constitute the first minterm of our SOP. The second minterm should contain B, complemented sel1 and uncomplemented sel2 according to the description above. Similarly, the third minterm contains C, uncomplemented sel1 and complemented sel2. Finally, the last minterm contains D, uncomplemented sel1 and uncomplemented sel2 control inputs. Therefore, the SOP expression for the 4-1 MUX becomes equal to the logic expression below and is implemented in Fig. 9.30.

$$\text{OUT} = \overline{\text{sel1}} \,.\, \overline{\text{sel2}} \,.\, \text{A} + \overline{\text{sel1}} \,.\, \text{sel2} \,.\, \text{B} + \text{sel1} \,.\, \overline{\text{sel2}} \,.\, \text{C} \; + \; \text{sel1} \,.\, \text{sel2} \,.\, \text{D}$$

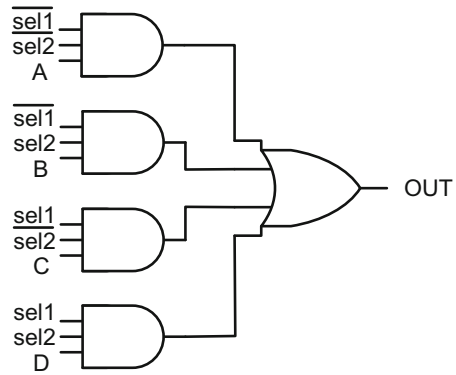


Fig. 9.30 4-1 MUX logic circuit in SOP form

However, implementing 4-1 MUX this way is not advantageous due to the amount of gate delays; a three-input AND gate is a serial combination of a three-input NAND and an inverter, and similarly a four-input OR connects a four-input NOR to an inverter. Therefore, we obtain a minimum of four gate delays instead of two according to this circuit.

Logic translations are possible to reduce the gate delay. The first stage of this process is to complement the outputs of all four three-input AND gates. This necessitates complementing the inputs of the four-input OR gate, and results in a circuit in Fig. 9.31.

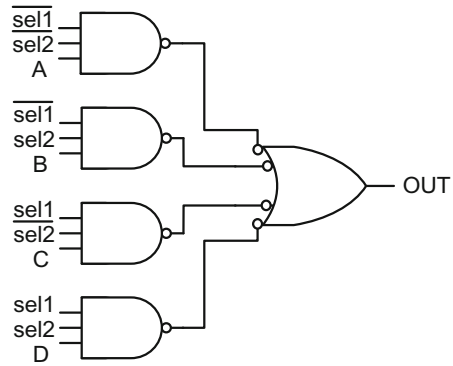


Fig. 9.31 Logic conversion of 4-1 MUX in Fig. 9.30

However, an OR gate with complemented inputs is equivalent to a NAND gate. Therefore, the circuit in Fig. 9.32 becomes optimal for implementation purposes because the total MUX delay is only the sum of a three-input NAND and a four-input NAND gate delays instead of the earlier four gate delays.

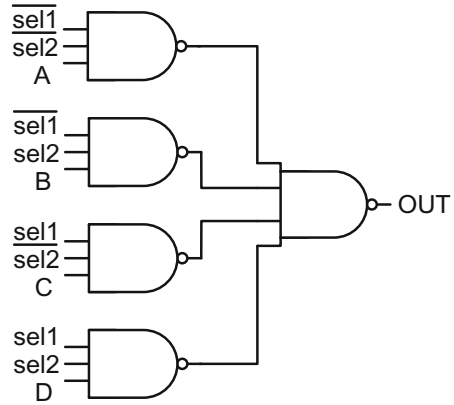


Fig. 9.32 4-1 MUX logic circuit in NAND-NAND form

The symbolic diagram of the 4-1 MUX is shown in Fig. 9.33.

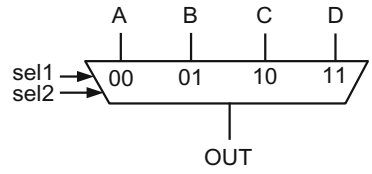


Fig. 9.33 4-1 MUX symbol

Encoders

Encoders are combinational logic blocks that receive 2^N number of inputs and produce N number of encoded outputs.

Example 9.12 Generate an encoding logic from the truth table given in Table 9.14.

Table 9.14 An arbitrary encoder truth table with four inputs

IN1	IN2	IN3	IN4	OUT1	OUT2
0	0	0	0	0	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	0

The K-maps shown in Fig. 9.34 groups 1s and produces the SOP expressions for OUT1 and OUT2.

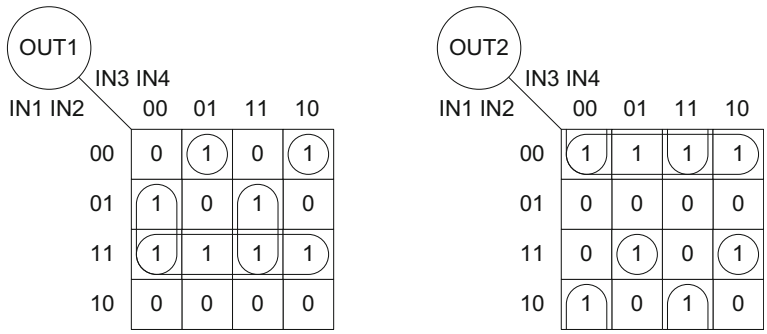


Fig. 9.34 K-maps of the truth table in Table 9.14

$$\begin{aligned}\text{OUT1} &= \text{IN1}\,.\,\text{IN2} + \overline{\text{IN1}}\,.\,\overline{\text{IN2}}\,.\,\text{IN3}\,.\,\overline{\text{IN4}} + \overline{\text{IN1}}\,.\,\overline{\text{IN2}}\,.\,\overline{\text{IN3}}\,.\,\text{IN4} \\ &\quad + \text{IN2}\,.\,\overline{\text{IN3}}\,.\,\overline{\text{IN4}} + \text{IN2}\,.\,\text{IN3}\,.\,\text{IN4} \\ &= \text{IN1}\,.\,\text{IN2} + \overline{\text{IN1}}\,.\,\overline{\text{IN2}}\,.\,(\text{IN3} \oplus \text{IN4}) + \text{IN2}\,.\,(\overline{\text{IN3} \oplus \text{IN4}})\end{aligned}$$

$$\begin{aligned}\text{OUT2} &= \overline{\text{IN1}}\,.\,\overline{\text{IN2}} + \overline{\text{IN2}}\,.\,\overline{\text{IN3}}\,.\,\overline{\text{IN4}} + \overline{\text{IN2}}\,.\,\text{IN3}\,.\,\text{IN4} \\ &\quad + \text{IN1}\,.\,\text{IN2}\,.\,\overline{\text{IN3}}\,.\,\text{IN4} + \text{IN1}\,.\,\text{IN2}\,.\,\text{IN3}\,.\,\overline{\text{IN4}} \\ &= \overline{\text{IN1}}\,.\,\overline{\text{IN2}} + \overline{\text{IN2}}\,.\,(\overline{\text{IN3} \oplus \text{IN4}}) + \text{IN1}\,.\,\text{IN2}\,.\,(\text{IN3} \oplus \text{IN4})\end{aligned}$$

Decoders

Decoders are combinational logic blocks used to decode encoded inputs. An ordinary decoder takes N inputs and produces 2^N outputs.

Example 9.13 Design a line decoder in which an active high enable signal activates only one of eight independent outputs according to truth table in Table 9.15. When the enable signal is lowered to logic 0, all eight outputs are disabled and stay at logic 0.

Table 9.15 Truth table of a line decoder with three inputs with enable

EN	IN[2]	IN[1]	IN[0]	OUT[7]	OUT[6]	OUT[5]	OUT[4]	OUT[3]	OUT[2]	OUT[1]	OUT[0]
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

In this table, all outputs become logic 0 when the enable signal, EN, is at logic 0. However, when EN = 1, activation of the output starts. For each three-bit input entry, there is always one output at logic 1. For example, when IN[2] = IN[1] = IN[0] = 0, OUT[0] becomes active and equals to logic 1 while all other outputs stay at logic 0. IN[2] = IN[1] = IN[0] = 1 activates OUT[7] and disables all other outputs..

We can produce each output expression from OUT[7] to OUT[0] simply by reading the input values from the truth table. The accompanying circuit is composed of eight AND gates, each with four inputs as shown in Fig. 9.35.

$$\text{OUT}[7] = \text{EN} \cdot \text{IN}[2] \cdot \text{IN}[1] \cdot \text{IN}[0]$$
$$\text{OUT}[6] = \text{EN} \cdot \text{IN}[2] \cdot \text{IN}[1] \cdot \overline{\text{IN}[0]}$$
$$\text{OUT}[5] = \text{EN} \cdot \text{IN}[2] \cdot \overline{\text{IN}[1]} \cdot \text{IN}[0]$$
$$\text{OUT}[4] = \text{EN} \cdot \text{IN}[2] \cdot \overline{\text{IN}[1]} \cdot \overline{\text{IN}[0]}$$
$$\text{OUT}[3] = \text{EN} \cdot \overline{\text{IN}[2]} \cdot \text{IN}[1] \cdot \text{IN}[0]$$
$$\text{OUT}[2] = \text{EN} \cdot \overline{\text{IN}[2]} \cdot \text{IN}[1] \cdot \overline{\text{IN}[0]}$$
$$\text{OUT}[1] = \text{EN} \cdot \overline{\text{IN}[2]} \cdot \overline{\text{IN}[1]} \cdot \text{IN}[0]$$
$$\text{OUT}[0] = \text{EN} \cdot \overline{\text{IN}[2]} \cdot \overline{\text{IN}[1]} \cdot \overline{\text{IN}[0]}$$

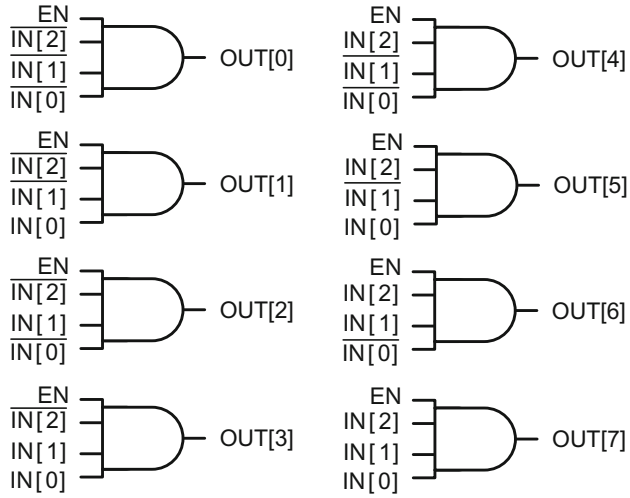


Fig. 9.35 Logic circuit of a line decoder in Table 9.15

One-Bit Full Adder

A one-bit full adder has three inputs: A, B, and carry-in (CIN), and two outputs: sum (SUM) and carry-out (COUT). The symbolic representation of a full adder is shown in Fig. 9.36.

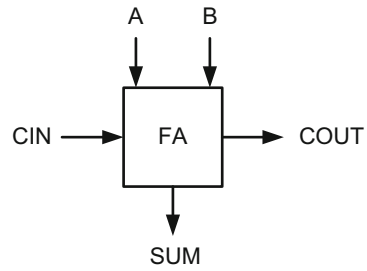


Fig. 9.36 One-bit full adder symbol

A one-bit full adder simply adds the contents of its two inputs, A and B, to the contents of CIN, and forms the truth table given in Table 9.16.

Table 9.16 One-bit full adder truth table

CIN	A	B	SUM	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

We can obtain the minimized SOP expressions for SUM and COUT from the K-maps in Figs. 9.37 and 9.38.

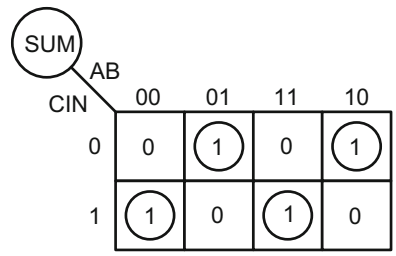


Fig. 9.37 SUM output of a one-bit full adder Consequently

Consequently,

$$\begin{aligned}\text{SUM} &= \overline{\text{A}}\,.\,\overline{\text{B}}\,.\,\text{CIN} + \overline{\text{A}}\,.\,\text{B}\,.\,\overline{\text{CIN}} + \text{A}\,.\,\text{B}\,.\,\text{CIN} + \text{A}\,.\,\overline{\text{B}}\,.\,\overline{\text{CIN}} \\ &= \text{CIN}\,.\,(\overline{\text{A}}\,.\,\overline{\text{B}} + \text{A}\,.\,\text{B}) + \overline{\text{CIN}}\,.\,(\overline{\text{A}}\,.\,\text{B} + \text{A}\,.\,\overline{\text{B}}) \\ &= \text{CIN}\,.\,(\overline{\text{A} \oplus \text{B}}) + \overline{\text{CIN}}\,.\,(\text{A} \oplus \text{B}) \\ &= \text{A} \oplus \text{B} \oplus \text{CIN}\end{aligned}$$

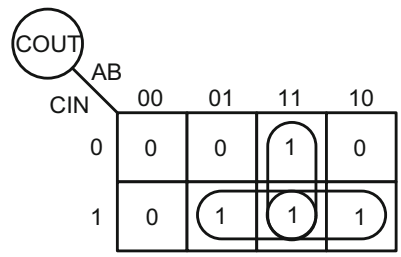


Fig. 9.38 COUT output of a one-bit full adder

Thus,

$$\begin{aligned} \text{COUT} &= \text{CIN} \,.\, \text{B} + \text{A} \,.\, \text{B} + \text{A} \,.\, \text{CIN} \\ &= \text{CIN} \,.\, (\text{A} + \text{B}) + \text{A} \,.\, \text{B} \end{aligned}$$

The resultant logic circuits for SUM and COUT are shown in Fig. 9.39.

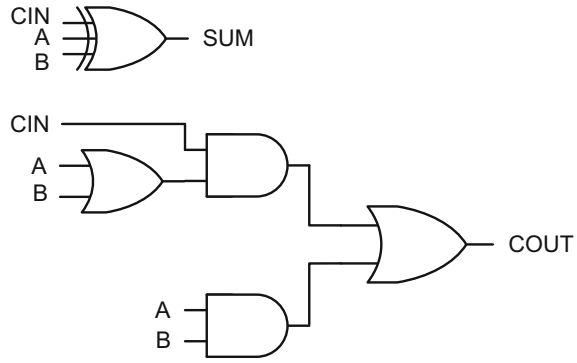


Fig. 9.39 One-bit full adder logic circuit

One-Bit Half Adder

A one-bit half adder has only two inputs, A and B with no CIN. A and B inputs are added to generate SUM and COUT outputs. The symbolic representation of a half-adder is shown in Fig. 9.40.

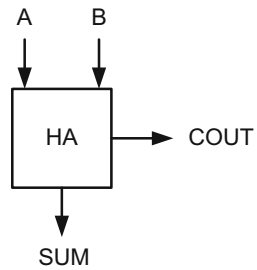


Fig. 9.40 One-bit half-adder symbol

The truth table given in Table 9.17 describes the functionality of the half adder.

Table 9.17 One-bit half-adder truth table

A	B	SUM	COUT
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From the truth table, the POS expressions for SUM and COUT can be written as:

$$\text{SUM} = \text{A} \oplus \text{B}$$

$$\text{COUT} = \text{A} \,.\, \text{B}$$

Therefore, we can produce SUM and COUT circuits as shown in Fig. 9.41.

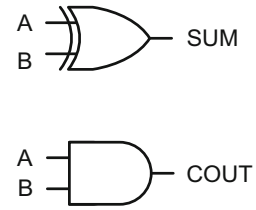


Fig. 9.41 One-bit half adder logic circuit

9.6 Adders

One-bit full-adders can be cascaded serially to produce multiple-bit adder configurations. There are three basic adder types:

Ripple-Carry Adder
Carry-Look-Ahead (CLA) Adder
Carry-Select Adder

However, different hybrid adder topologies can be designed by combining these configurations. For the sake of simplicity, we will limit the number of bits to four and explain each topology in detail.

Ripple-Carry Adder

The ripple-carry adder is a cascaded configuration of multiple one-bit full adders. The circuit topology of a four-bit ripple carry adder is shown in Fig. 9.42. In this figure, the carry-out output of a one-bit full adder is connected to the carry-in input of the next full adder to propagate carry from CIN[0] to higher bits.

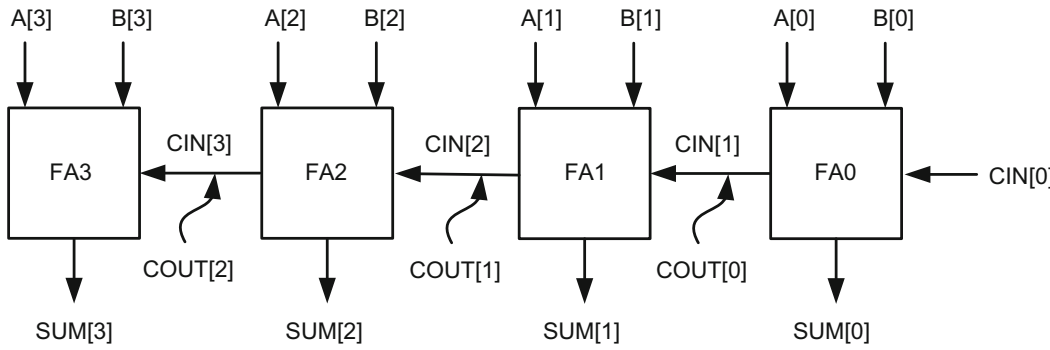


Fig. 9.42 Four-bit ripple-carry adder

For the 0-th bit of this adder, we have:

$$\text{SUM}[0] = \text{A}[0] \oplus \text{B}[0] \oplus \text{CIN}[0]$$
$$\text{COUT}[0] = \text{CIN}[1] = \text{A}[0] \,.\, \text{B}[0] + \text{CIN}[0] \,.\, (\text{A}[0] + \text{B}[0]) = \text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0]$$

where,

G[0] = A[0] . B[0] as the zeroth order generation term
P[0] = A[0] + B[0] as the zeroth order propagation term

For the first bit:

$$\text{SUM}[1] = \text{A}[1] \oplus \text{B}[1] \oplus \text{CIN}[1] = \text{A}[1] \oplus \text{B}[1] \oplus (\text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0])$$
$$\begin{aligned}\text{COUT}[1] &= \text{CIN}[2] = \text{G}[1] + \text{P}[1] \,.\, \text{CIN}[1] \\ &= \text{G}[1] + \text{P}[1] \,.\, (\text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0]) = \text{G}[1] + \text{P}[1] \,.\, \text{G}[0] + \text{P}[1] \,.\, \text{P}[0] \,.\, \text{CIN}[0]\end{aligned}$$

where,

G[1] = A[1] . B[1] as the first order generation term
P[1] = A[1] + B[1] as the first order propagation term

For the second bit:

$$\begin{aligned}\text{SUM}[2] &= \text{A}[2] \oplus \text{B}[2] \oplus \text{CIN}[2] = \text{A}[2] \oplus \text{B}[2] \oplus \{\text{G}[1] + \text{P}[1] \,.\, (\text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0])\} \\ &= \text{A}[2] \oplus \text{B}[2] \oplus (\text{G}[1] + \text{P}[1] \,.\, \text{G}[0] + \text{P}[1] \,.\, \text{P}[0] \,.\, \text{CIN}[0])\end{aligned}$$
$$\begin{aligned}\text{COUT}[2] &= \text{CIN}[3] = \text{G}[2] + \text{P}[2] \,.\, \text{CIN}[2] \\ &= \text{G}[2] + \text{P}[2] \,.\, \{\text{G}[1] + \text{P}[1] \,.\, (\text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0])\} \\ &= \text{G}[2] + \text{P}[2] \,.\, \text{G}[1] + \text{P}[2] \,.\, \text{P}[1] \,.\, \text{G}[0] + \text{P}[2] \,.\, \text{P}[1] \,.\, \text{P}[0] \,.\, \text{CIN}[0]\end{aligned}$$

where,

G[2] = A[2] . B[2] as the second order generation term
P[2] = A[2] + B[2] as the second order propagation term

And for the third bit:

$$\begin{aligned}\text{SUM}[3] &= \text{A}[3] \oplus \text{B}[3] \oplus \text{CIN}[3] = \text{A}[3] \oplus \text{B}[3] \oplus \{\text{G}[2] + \text{P}[2] \,.\, \{\text{G}[1] \\ &\qquad + \text{P}[1] \,.\, (\text{G}[0] + \text{P}[0] \,.\, \text{CIN}[0])\}\} \\ &= \text{A}[3] \oplus \text{B}[3] \oplus (\text{G}[2] + \text{P}[2] \,.\, \text{G}[1] + \text{P}[2] \,.\, \text{P}[1] \,.\, \text{G}[0] + \text{P}[2] \,.\, \text{P}[1] \,.\, \text{P}[0] \,.\, \text{CIN}[0])\end{aligned}$$

These functional expressions of SUM and COUT also serve to estimate the maximum gate delays for each bit of the adder.

The circuit diagram in Fig. 9.43 explains the maximum delay path through each bit.

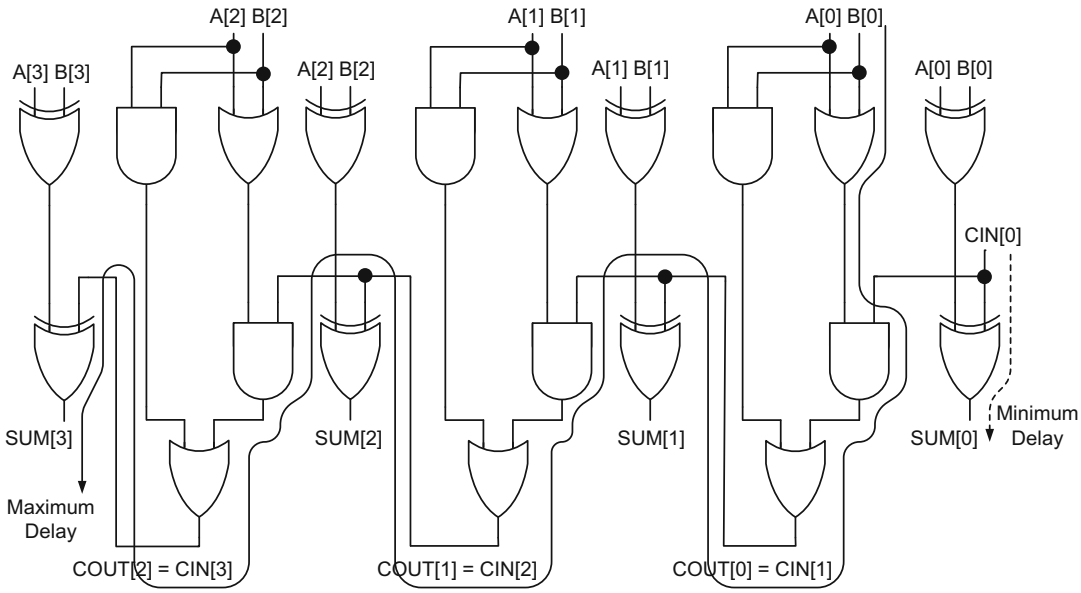


Fig. 9.43 Logic circuit of the four-bit adder with the maximum and minimum delays

The maximum gate delay from A[0] or B[0] inputs to SUM[0] is $2T_{XOR2}$, where T_{XOR2} is a single two-input XOR gate delay.

The maximum gate delay from A[0] or B[0] to COUT[0] is $2T_{OR2} + T_{AND2}$ where T_{OR2} and T_{AND2} are two-input OR gate and two-input AND gate delays, respectively.

The gate delay from A[1] or B[1] to SUM[1] is still $2T_{XOR2}$; however, the delay from A[0] or B[0] to SUM[1] is $2T_{OR2} + T_{AND2} + T_{XOR2}$, which is more than $2T_{XOR2}$ and must be considered the maximum gate delay for this particular bit position and for more significant bits.

The maximum gate delay from A[0] or B[0] to COUT[1] is $3T_{OR2} + 2T_{AND2}$. It may make more sense to expand the expression for COUT[1] as COUT[1] = G[1] + P[1] . G[0] + P[1] . P[0] . CIN[0], and figure out if the overall gate delay, $T_{OR2} + T_{AND3} + T_{OR3}$, is smaller compared to $3T_{OR2} + 2T_{AND2}$. Here, T_{AND3} and T_{OR3} are single three-input AND and OR gate delays, respectively.

The maximum gate delay from A[0] or B[0] to SUM[2] is $3T_{OR2} + 2T_{AND2} + T_{XOR2}$. When the expression for SUM[2] is expanded as SUM[2] = A[2] ⊕ B[2] ⊕ (G[1] + P[1] . G[0] + P[1] . P[0] . CIN[0]), we see that this delay becomes $T_{OR2} + T_{AND3} + T_{OR3} + T_{XOR2}$, and may be smaller than the original delay if $T_{AND3} < 2T_{AND2}$ and $T_{OR3} < 2T_{OR2}$.

The maximum gate delay from A[0] or B[0] to COUT[2] is $4T_{OR2} + 3T_{AND2}$. When COUT[2] is expanded as COUT[2] = G[2] + P[2] . G[1] + P[2] . P[1] . G[0] + P[2] . P[1] . P[0] . CIN[0], the maximum delay becomes $T_{OR2} + T_{AND4} + T_{OR4}$, and may be smaller than the original delay if $T_{AND4} < 3T_{AND2}$ and $T_{OR4} < 3T_{OR2}$. Here, T_{AND4} and T_{OR4} are single four-input AND and OR gate delays, respectively.

Finally, the maximum delay from A[0] or B[0] to SUM[3] is $4T_{OR2} + 3T_{AND2} + T_{XOR2}$, which is also the maximum propagation delay for this adder. When the functional expression for SUM[3] is expanded as SUM[3] = A[3] ⊕ B[3] ⊕ (G[2] + P[2] . G[1] + P[2] . P[1] . G[0] + P[2] . P[1] . P[0] . CIN[0]), the total propagation delay becomes $T_{OR2} + T_{AND4} + T_{OR4} + T_{XOR2}$, and again it may be smaller compared to the original delay if $T_{AND4} < 3T_{AND2}$ and $T_{OR4} < 3T_{OR2}$.

Carry-Look-Ahead Adder

The idea behind carry-look-ahead (CLA) adders is to create a topology where carry-in bits to all one-bit full adders are available simultaneously. A four-bit CLA circuit topology is shown in Fig. 9.44. In this figure, the SUM output of a more significant bit does not have to wait until the carry bit ripples from the least significant bit position, but it gets computed after some logic delay. In reality, all carry-in signals are generated by complex combinational logic blocks called Carry-Look-Ahead (CLA) hook-ups, as shown in Fig. 9.44. Each CLA block adds a certain propagation delay on top of the two-input XOR gate delay to produce a SUM output.

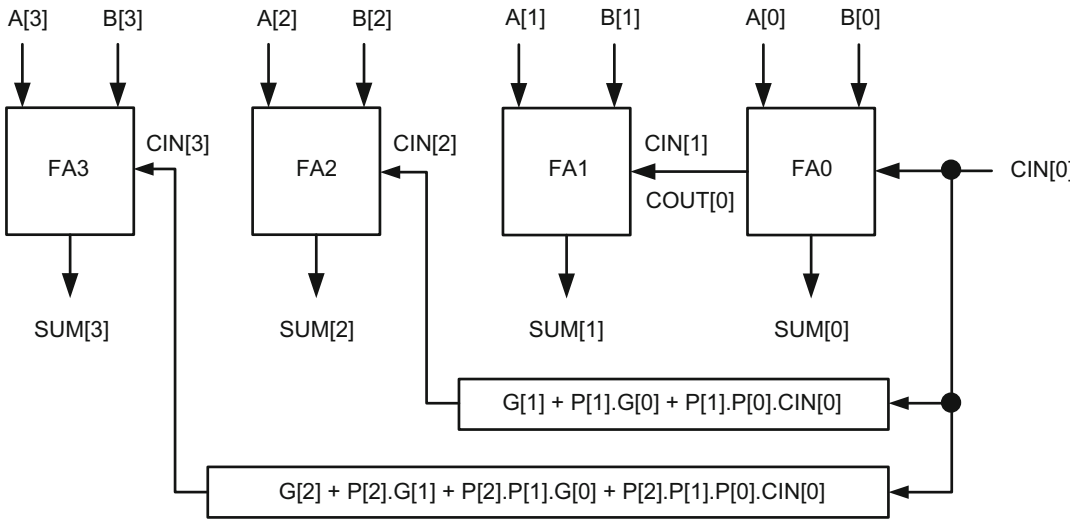


Fig. 9.44 A four-bit carry-look-ahead adder

The earlier SUM and CIN expressions derived for the ripple carry adder can be applied to the CLA adder to generate its functional equations.

Therefore,

$$SUM[0] = A[0] \oplus B[0] \oplus CIN[0]$$
$$SUM[1] = A[1] \oplus B[1] \oplus CIN[1]$$
$$SUM[2] = A[2] \oplus B[2] \oplus CIN[2]$$
$$SUM[3] = A[3] \oplus B[3] \oplus CIN[3]$$

where,

$$CIN[1] = G[0] + P[0] \,.\, CIN[0]$$
$$CIN[2] = G[1] + P[1] \,.\, G[0] + P[1] \,.\, P[0] \,.\, CIN[0]$$
$$CIN[3] = G[2] + P[2] \,.\, G[1] + P[2] \,.\, P[1] \,.\, G[0] + P[2] \,.\, P[1] \,.\, P[0] \,.\, CIN[0]$$

Therefore, CIN[1] is generated by the COUT[0] function within FA0. However, CIN[2] and CIN[3] have to be produced by separate logic blocks in order to provide CIN signals for FA2 and FA3.

According to Fig. 9.44, once a valid CIN[0] becomes available, it takes successively longer times to generate valid signals for higher order SUM outputs due to the increasing complexity in CLA hook-ups.

Assume that T_{SUM0}, T_{SUM1}, T_{SUM2} and T_{SUM3} are the propagation delays of bits 0, 1, 2 and 3 with respect to the CIN[0] signal. We can approximate $T_{SUM0} = T_{XOR2}$. To compute T_{SUM1}, we need to examine the expression for CIN[1]. In this expression, P[0] . CIN[0] produces a two-input AND gate delay, and G[0] + (P[0] . CIN[0]) produces a two-input OR gate delay to be added on top of T_{XOR2}. Therefore, T_{SUM1} becomes $T_{SUM1} = T_{AND2} + T_{OR2} + T_{XOR2}$. Similarly, the expressions for CIN[2] and CIN[3] produce $T_{SUM2} = T_{AND3} + T_{OR3} + T_{XOR2}$ and $T_{SUM3} = T_{AND4} + T_{OR4} + T_{XOR2}$, respectively.

The maximum propagation delay for this adder is, therefore, $T_{SUM3} = T_{AND4} + T_{OR4} + T_{XOR2}$.

Despite the CLA adder's advantage of being faster than the ripple-carry adder, in most cases the extra CLA logic blocks make this adder topology occupy a larger chip area if the number of adder bits is above eight.

Carry-Select Adder

Carry-Select Adders require two rows of identical adders. The identical adders can be as simple as two rows of identical ripple-carry adders or CLA adders depending on the design requirements. Figure 9.45 shows the circuit topology of a four-bit carry-select adder composed of two rows of ripple carry adders.

In this figure, the full adder at the least significant bit position operates normally and generates a value for COUT[0]. As this value is generated, the two one-bit full adders, one with CINA[1] = 0 and the other with CINB[1] = 1, simultaneously generate SUMA[1] and SUMB[1]. If COUT[0] becomes equal to one, SUMB[1] gets selected and becomes SUM[1]; otherwise, SUMA[1] becomes the SUM[1] output. Whichever value ends up being SUM[1], it is produced after a 2-1 MUX propagation delay.

However, we cannot say the same in generating SUM[2] and SUM[3] outputs in this figure. After producing SUM[1], carry ripples through both adders normally to generate SUM[2] and SUM[3]; hence, the speed advantage of having two rows of adders becomes negligible. Therefore, we must be careful when employing a carry-select scheme before designing an adder, as this method practically doubles the chip area.

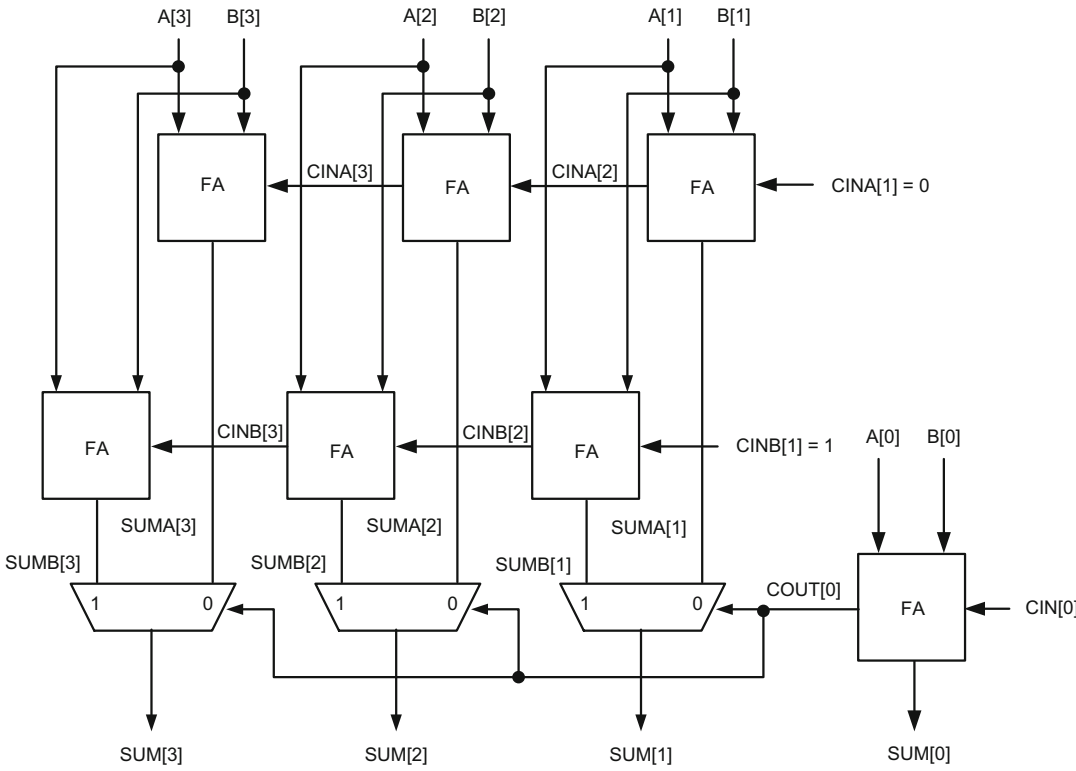


Fig. 9.45 A four-bit carry-select adder

Even though carry-select topology is ineffective in speeding up this particular four-bit adder, it may be advantageous if employed to an adder with greater number of bits in conjunction with another adder topology such as the CLA.

Example 9.14 Design a 32-bit carry-look-ahead adder. Compute the worst-case propagation delay in the circuit.

We need to be careful in dealing with the CLA hook-ups when generating higher order terms because the complexity of these logic blocks can "grow" exponentially in size while they may only provide marginal speed gain when compared to ripple-carry scheme.

Therefore, the first step of the design process is to separate the adder into eight-bit segments with full CLA hook-ups. The proposed topology is shown in Fig. 9.46.

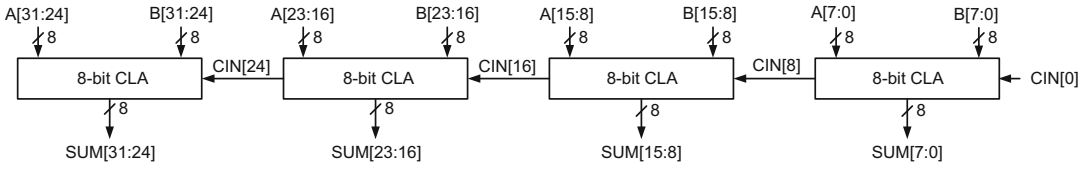


Fig. 9.46 A 32-bit carry-look-ahead topology

Each eight-bit CLA segment contains six CLA hook-ups from CLA0 to CLA5 as shown in Fig. 9.47.

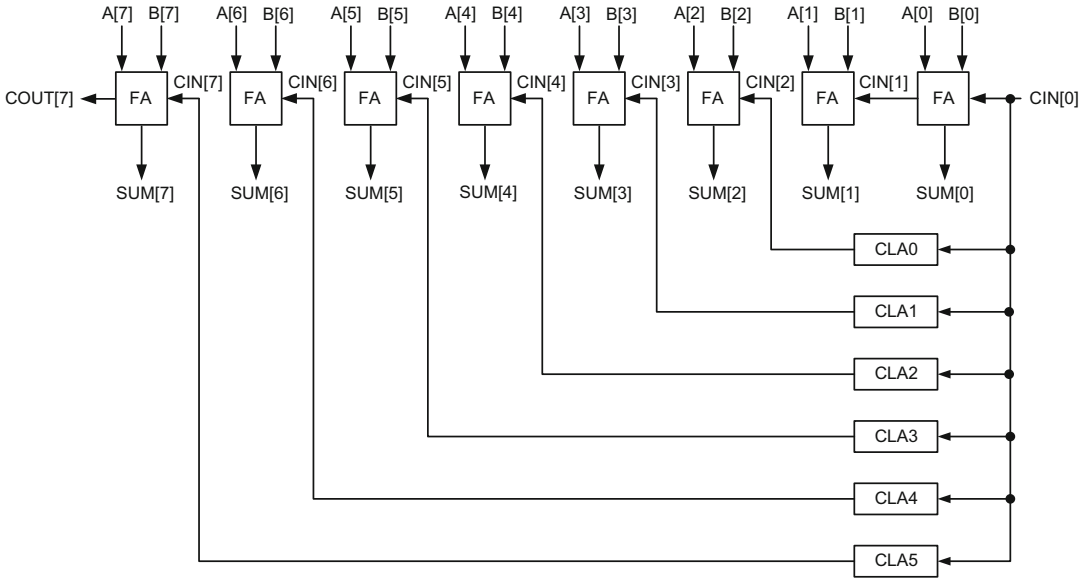


Fig. 9.47 An eight-bit segment of the carry-look ahead adder in Fig. 9.46

CIN and SUM expressions from bit 0 through bit 7 are given below.

$SUM[0] = A[0] \oplus B[0] \oplus CIN[0]$

$SUM[1] = A[1] \oplus B[1] \oplus CIN[1]$
where, $CIN[1] = G[0] + P[0] \cdot CIN[0]$

$SUM[2] = A[2] \oplus B[2] \oplus CIN[2]$
where, $CIN[2] = G[1] + P[1] \cdot G[0] + P[1] \cdot P[0] \cdot CIN[0]$

$SUM[3] = A[3] \oplus B[3] \oplus CIN[3]$
where, $CIN[3] = G[2] + P[2] \cdot G[1] + P[2] \cdot P[1] \cdot G[0] + P[2] \cdot P[1] \cdot P[0] \cdot CIN[0]$

$SUM[4] = A[4] \oplus B[4] \oplus CIN[4]$
where, $CIN[4] = G[3] + P[3] \cdot (G[2] + P[2] \cdot G[1] + P[2] \cdot P[1] \cdot G[0] + P[2] \cdot P[1] \cdot P[0] \cdot CIN[0])$

$$\begin{aligned}
&\text{SUM[5]} = \text{A[5]} \oplus \text{B[5]} \oplus \text{CIN[5]}\\
&\text{where, CIN[5]} = \text{G[4]} + \text{P[4]} \,.\, \{\text{G[3]} + \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]}\\
&\qquad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})\}\\
&\qquad = \text{G[4]} + \text{P[4]} \,.\, \text{G[3]} + \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]}\\
&\qquad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})
\end{aligned}$$

$$\begin{aligned}
&\text{SUM[6]} = \text{A[6]} \oplus \text{B[6]} \oplus \text{CIN[6]}\\
&\text{where, CIN[6]} = \text{G[5]} + \text{P[5]} \,.\, \{\text{G[4]} + \text{P[4]} \,.\, \text{G[3]} + \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]}\\
&\qquad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})\}\\
&\qquad = \text{G[5]} + \text{P[5]} \,.\, \text{G[4]} + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{G[3]} + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]}\\
&\qquad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})
\end{aligned}$$

$$\begin{aligned}
&\text{SUM[7]} = \text{A[7]} \oplus \text{B[7]} \oplus \text{CIN[7]}\\
&\text{where, CIN[7]} = \text{G[6]} + \text{P[6]} \,.\, \{\text{G[5]} + \text{P[5]} \,.\, \text{G[4]} + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{G[3]}\\
&\qquad + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]} + P[2] \,.\, \text{P[1]} \,.\, \text{G[0]}\\
&\qquad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})\}
\end{aligned}$$

And finally,

$$\begin{aligned}
\text{COUT[7]} = \text{CIN[8]} &= \text{G[7]} + \text{P[7]} \,.\, \{\text{G[6]} + \text{P[6]} \,.\, \{\text{G[5]} + \text{P[5]} \,.\, \text{G[4]} + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{G[3]}\\
&\quad + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]}\\
&\quad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})\}\}\\
&= \text{G[7]} + \text{P[7]} \,.\, \text{G[6]} + \text{P[7]} \,.\, \text{P[6]} \,.\, \{\text{G[5]} + \text{P[5]} \,.\, \text{G[4]}\\
&\quad + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{G[3]} + \text{P[5]} \,.\, \text{P[4]} \,.\, \text{P[3]} \,.\, (\text{G[2]} + \text{P[2]} \,.\, \text{G[1]}\\
&\quad + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{G[0]} + \text{P[2]} \,.\, \text{P[1]} \,.\, \text{P[0]} \,.\, \text{CIN[0]})\}
\end{aligned}$$

In these derivations, particular attention was paid to limit the number of inputs to four in all AND and OR gates since larger gate inputs are counterproductive in reducing the overall propagation delay.

From these functional expressions, maximum propagation delays for SUM[7] and COUT[7] are Estimated using the longest logic strings in Fig. 9.48.

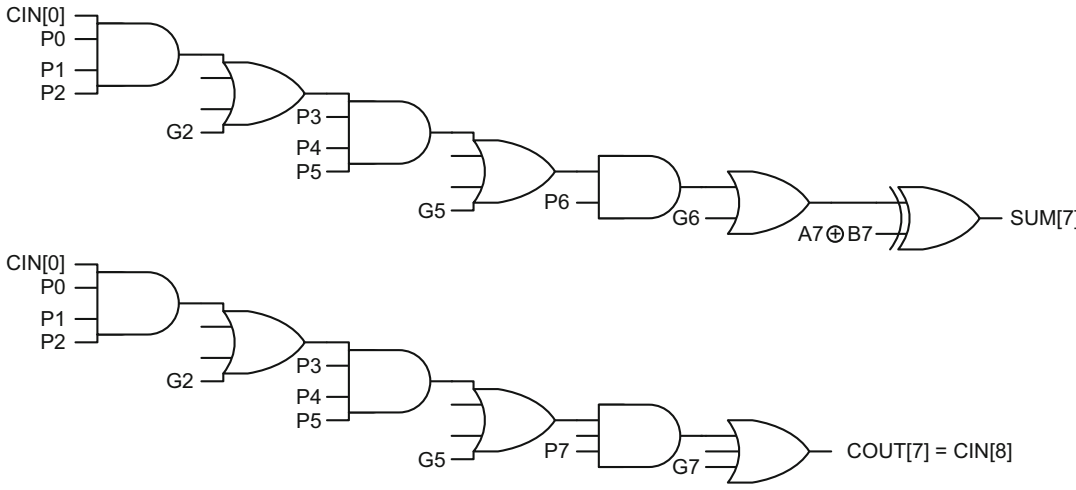


Fig. 9.48 Propagation delay estimation of the eight-bit carry-look-ahead adder in Fig. 9.47

For SUM[7], the minterm, P[2] . P[1] . P[0] . CIN[0], generates the first four-input AND gate. This is followed by a four-input OR-gate whose minterms are G[2], P[2] . G[1], P[2] . P[1] . G[0], and P[2] . P[1] . P[0] . CIN[0]. This gate is followed by a four-input AND, four-input OR, two-input AND, two-input OR and two-input XOR-gates in successive order. The entire string creates a propagation delay of $T_{SUM7} = 2(T_{AND4} + T_{OR4}) + T_{AND2} + T_{OR2} + T_{XOR2}$ from CIN[0] to SUM[7].

For COUT[7], the longest propagation delay is between CIN[0] to COUT[7] as shown in Fig. 9.48, and it is equal to $T_{COUT7} = 2(T_{AND4} + T_{OR4}) + T_{AND3} + T_{OR3}$. The delays for the rest of the circuit in Fig. 9.46 become easy to determine since the longest propagation delays have already been evaluated.

The delay from CIN[0] to SUM[15], T_{SUM15}, simply becomes equal to the sum of T_{COUT7} and T_{SUM7}. In other words, $T_{SUM15} = 4(T_{AND4} + T_{OR4}) + T_{AND3} + T_{OR3} + T_{AND2} + T_{OR2} + T_{XOR2}$. Similarly, the delay from CIN[0] to COUT[15], T_{COUT15}, is equal to $T_{COUT15} = 4(T_{AND4} + T_{OR4}) + 2(T_{AND3} + T_{OR3})$.

The remaining delays are evaluated in the same way and lead to the longest propagation delay in this circuit, which is from CIN[0] to SUM[31], T_{SUM31}.

Thus,

$$T_{SUM31} = 3[2(T_{AND4} + T_{OR4}) + T_{AND3} + T_{OR3}] + 2(T_{AND4} + T_{OR4}) + T_{AND2} + T_{OR2} + T_{XOR2}$$

or

$$T_{SUM31} = 8(T_{AND4} + T_{OR4}) + 3(T_{AND3} + T_{OR3}) + T_{AND2} + T_{OR2} + T_{XOR2}$$

Example 9.15 Design a 32-bit hybrid carry-select/carry-look-ahead adder. Compute the worst-case propagation delay in the circuit.

Large adders are where the carry-select scheme shines! This is a classical example in which the maximum propagation delay is reduced considerably compared to the CLA scheme examined in Example 9.14.

As mentioned earlier, a twin set of an adder configuration is required by the carry-select scheme. The adders can be ripple-carry, carry-look-ahead or the combination of the two.

In this example, the 32-bit adder is again divided in eight-bit segments where each segment consists of a full CLA adder as shown in Fig. 9.49. The first segment, CLA-0, is a single unit which produces COUT[7] with full CLA hook-ups. The rest of the eight-bit segments are mirror images of each other and they are either named A-segments (A1, A2 and A3) or B-segments (B1, B2 and B3).

As the CLA-0 generates a valid COUT[7], the CLA-A1 and CLA-B1 simultaneously generate COUTA[15] and COUTB[15]. When COUT[7] finally forms, it selects either COUTA[15] or COUTB[15] depending on the value of CIN[0]. This segment produces COUT[15] and SUM[15:8].

COUT[15], on the other hand, is used to select between COUTA[23] and COUTB[23], both of which have already been formed when COUT[15] arrives at the 2-1 MUX as a control input. COUT[15] also selects between SUMA[23:16] and SUMB[23:16] to determine the correct SUM[23:16].

Similarly, COUT[23] is used as a control input to select between SUMA[31:24] and SUMB[31:24]. If there is a need for COUT[31], COUT[23] can serve to determine the value of COUT[31].

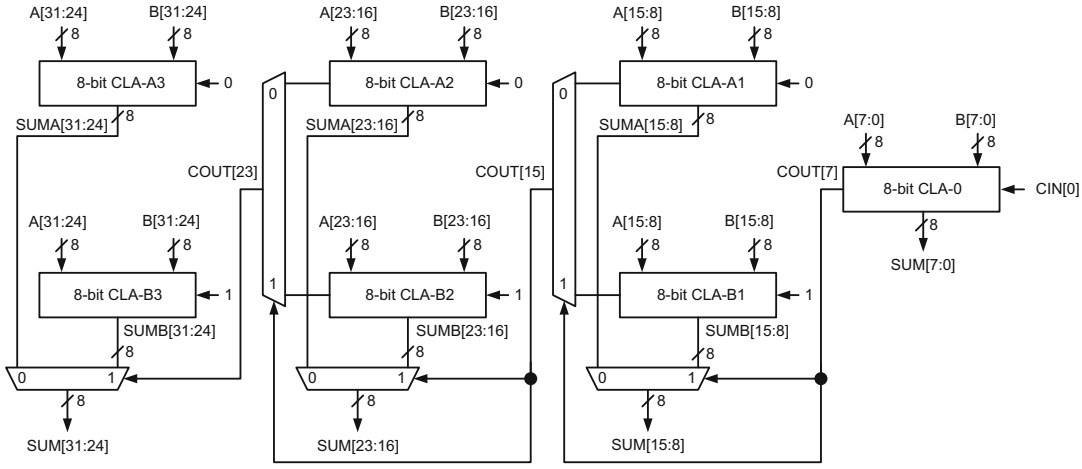


Fig. 9.49 A 32-bit carry-look-ahead/carry-select adder

The maximum propagation delay for the 32-bit carry-select/CLA adder can be found using the logic string in Fig. 9.50. The first section of this string from CIN[0] to COUT[7] is identical to the eight-bit CLA carry delay in Fig. 9.48. There are three cascaded MUX stages which correspond to the selection of COUT[15], COUT[23], and SUM[31].

Considering that a 2-1 MUX propagation delay consists of a two-input AND gate followed by a two-input OR gate, we obtain the following maximum delay for this 32-bit adder:

$T_{SUM31} = 2(T_{AND4} + T_{OR4}) + T_{AND3} + T_{OR3} + 3(T_{AND2} + T_{OR2})$

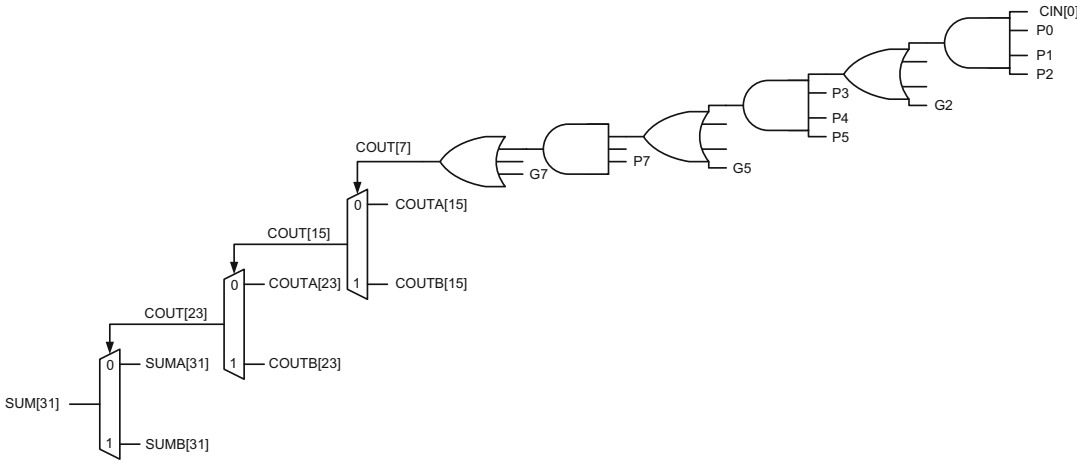


Fig. 9.50 Maximum delay propagation of the 32-bit adder in Fig. 9.49

Considering the maximum propagation delay in Example 9.14, this delay is shorter by at least $6(T_{AND4} + T_{OR4})$. Larger carry-select/carry-look-ahead adder schemes provide greater speed benefits at the cost of approximately doubling the adder area.

9.7 Subtractors

Subtraction is performed by a technique called Twos (2s) complement addition. Twos complement addition requires complementing one of the adder inputs (1s complement), and then adding one to the least significant bit.

Example 9.16 Form −4 using 2s complement addition using four bits.

A negative number is created by first inverting every bit of +4 (1s complement representation) and then adding 1 to it. +4 is equal to 0100 in four-bit binary form.

Its 1s complement is 1011. Its 2s complement is 1011 + 0001 = 1100.

Therefore, logic 0 signifies a positive sign, and logic 1 signifies a negative sign at the most significant bit position of a signed number.

Example 9.17 Add +4 to −4 using 2s complement addition

+4 = 0100 in binary form
−4 = 1100 in 2s complement form of +4.

Perform +4–4 = 0100 + 1100 = 1 0000
where, logic 1 at the overflow bit position is neglected in a four-bit binary format.

Therefore, we obtain 0000 = 0 as expected.

Subtractors function according to 2s complement addition. We need to form 1s complement of the adder input to be subtracted and use CIN[0] = 1 at the adder's least significant bit position to perform subtraction.

Figure 9.51 illustrates the topology of a 32-bit subtractor where input B is complemented and CIN[0] is tied to logic 1 to satisfy 2s complement addition and produce A–B.

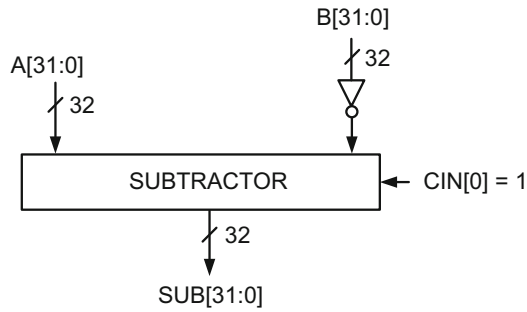


Fig. 9.51 A symbolic representation of a 32-bit subtractor

9.8 Shifters

There are two commonly used shifters in logic design:

Linear shifters
Barrel shifters

Linear Shifters

A linear shifter shifts its inputs by a number of bits to the right or left and routes the result to its output.

Example 9.18 Design a four-bit linear shifter that shifts its inputs to the left by one bit and produces logic 0 at the least significant output bit when SHIFT = 1. When SHIFT = 0, the shifter routes each input directly to the corresponding output.

The logic diagram for this shifter is given in Fig. 9.52. In this figure, each input is connected to the port 0 terminal of the 2-1 MUX as well as the port 1 terminal of the next MUX at the higher bit position. Therefore, when SHIFT = 1, logic 0, IN[0], IN[1], and IN[2] are routed through port 1 terminal of each 2-1 MUX and become OUT[0], OUT[1], OUT[2], and OUT[3], respectively. When SHIFT = 0, each input goes through port 0 terminal of the corresponding 2-1 MUX and becomes the shifter output.

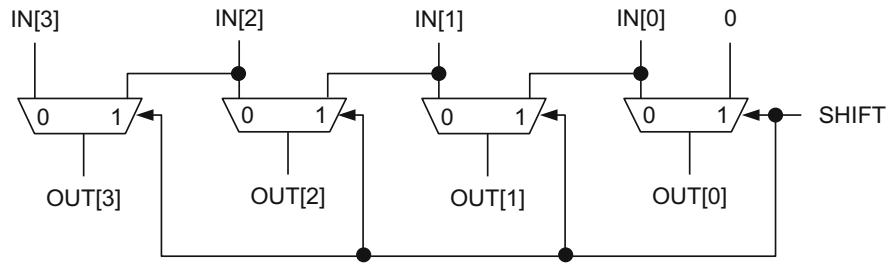


Fig. 9.52 Four-bit linear shifter

Barrel Shifters

Barrel shifters rotate their inputs either in clockwise or counterclockwise direction by a number of bits but preserve all their inputs when generating an output.

Example 9.19 Design a four-bit barrel shifter that rotates its inputs in a clockwise direction by one bit when SHIFT = 1. When SHIFT = 0, the shifter routes each one of its four inputs to its corresponding output.

The logic diagram for this shifter is given in Fig. 9.53. The only difference between this circuit and the linear shifter in Fig. 9.52 is the removal of logic 0 from the least significant bit, and connecting this input to the IN[3] pin instead. Consequently, this leads to OUT[0] = IN[3], OUT[1] = IN[0], OUT[2] = IN[1] and OUT[3] = IN[2] when SHIFT = 1, and OUT[0] = IN[0], OUT[1] = IN[1], OUT[2] = IN[2] and OUT[3] = IN[3] when SHIFT = 0.

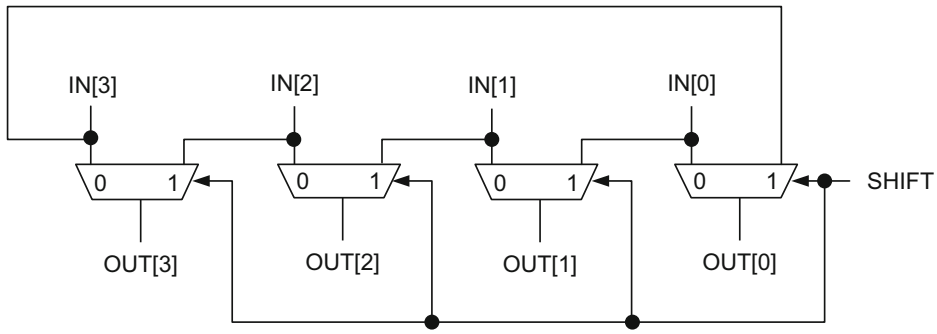


Fig. 9.53 Four-bit barrel shifter

Example 9.20 Design a four-bit barrel shifter that rotates its inputs clockwise by one or two bits.

First, there must be three control inputs specifying “no shift”, “shift 1 bit” and “shift 2 bits”. This requires a two control-bit input, SHIFT[1:0], as shown in Table 9.18. All

assignments in this table are done arbitrarily. However, it makes sense to assign a "No shift" to SHIFT[1:0] = 0, "Shift 1 bit" to SHIFT[1:0] = 1 and "Shift 2 bits" to SHIFT[1:0] = 2 for actual rotation amount.

Table 9.18 A four-bit barrel shifter truth table

SHIFT[1]	SHIFT[0]	OPERATION	OUT[3]	OUT[2]	OUT[1]	OUT[0]
0	0	No shift	IN[3]	IN[2]	IN[1]	IN[0]
0	1	Shift 1 bit	IN[2]	IN[1]	IN[0]	IN[3]
1	0	Shift 2 bits	IN[1]	IN[0]	IN[3]	IN[2]
1	1	No shift	IN[3]	IN[2]	IN[1]	IN[0]

According to this table, if there is no shift, each input bit is simply routed to its own output. If "shift 1 bit" input is active, then each input is routed to the neighboring output at the next significant bit position. In other words, IN[3] rotates clockwise and becomes OUT[0]; IN[0], IN[1] and IN[2] shift 1 bit to the left and become OUT[1], OUT[2] and OUT[3], respectively. If "shift 2 bits" becomes active, then each input is routed to the output of the neighboring bit which is two significant bits higher. This gives the impression that all input bits have been rotated twice before being routed to the output. Thus, OUT[0] = IN[2], OUT[1] = IN[3], OUT[2] = IN[0] and OUT[3] = IN[1].

Therefore, using Table 9.18, we can conclude the logic diagram in Fig. 9.54.

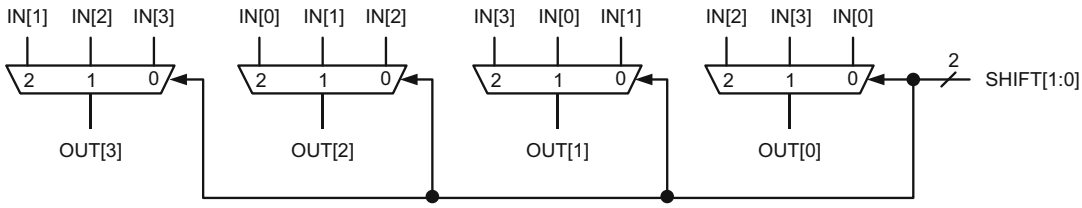


Fig. 9.54 Logic diagram of the barrel shifter in Table 9.18

A more detailed view of Fig. 9.54 is given in Fig. 9.55.

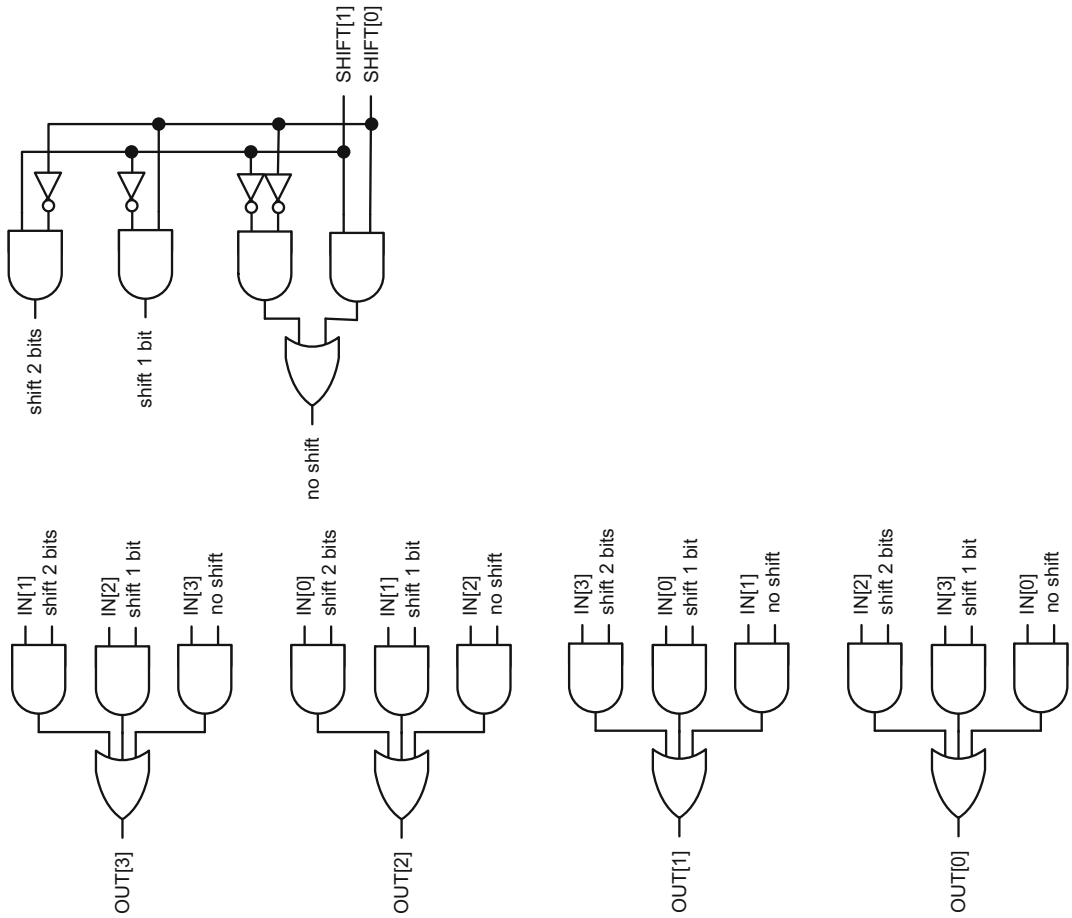


Fig. 9.55 Logic circuit of the barrel shifter in Fig. 9.54

9.9 Multipliers

Similar to our everyday hand multiplication method, an array multiplier generates all partial products before summing each column in the partial product tree to obtain a result. This scheme is explained in Fig. 9.56 for a four-bit array multiplier.

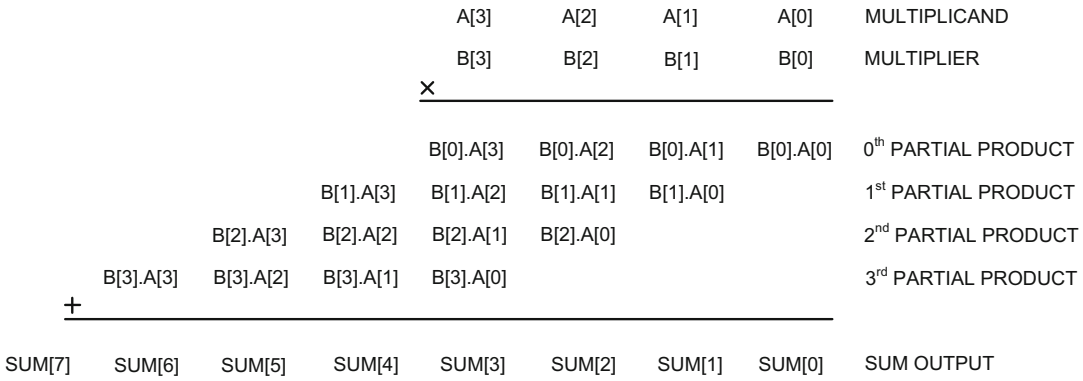


Fig. 9.56 4×4 array multiplier algorithm

The rules of partial product generation are as follows:

1. The zeroth partial product aligns with multiplicand and multiplier bit columns.
2. Each partial product is shifted one bit to the left with respect to the previous one once it is created.
3. Each partial product is the exact replica of the multiplicand if the multiplier bit is one. Otherwise, it is deleted.

Example 9.21 Multiply 1101 and 1001 according to the rules of array multiplication.

Suppose 1101 is a multiplicand and 1001 is a multiplier. Then, for a four-bit multiplier four partial products are formed. The bits in each column of the partial product are added successively. Carry bits are propagated to more significant bit positions. This process is illustrated in Fig. 9.57.

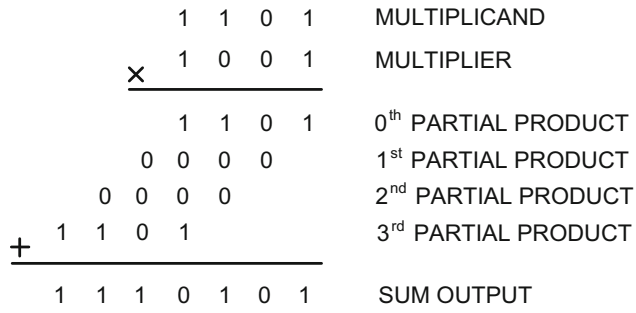


Fig. 9.57 4×4 array multiplier algorithm example

Example 9.22 Design the partial product tree for a four-bit array multiplier.

Following the convention in Fig. 9.56 and the rules of partial product generation for an array multiplier, we can implement the partial product tree as shown in Fig. 9.58.

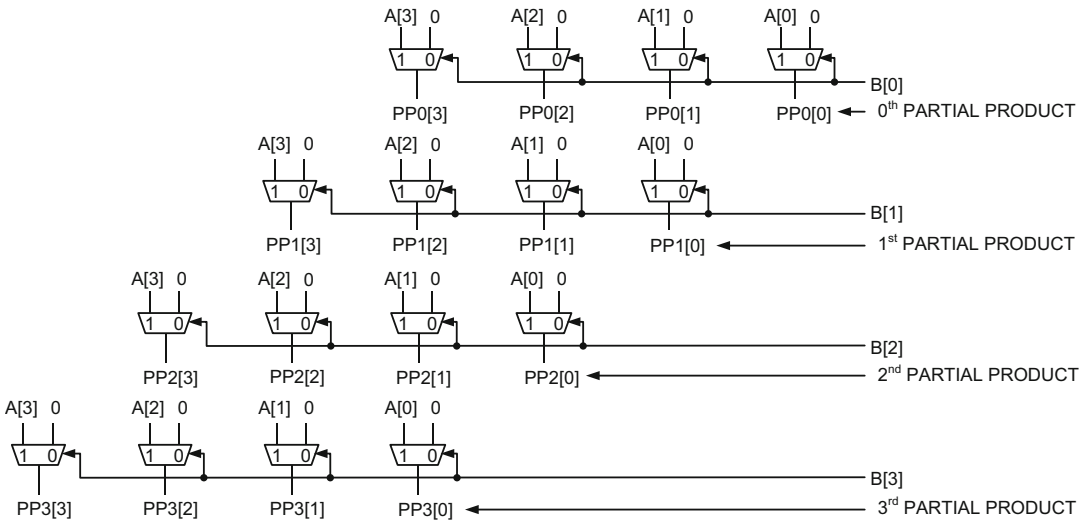


Fig. 9.58 4×4 array multiplier bit selector tree

In this figure, partial product elements of the zeroth partial product, B[0].A[3], B[0].A[2], B[0].A[1] and B[0].A[0], are replaced by PP0[3:0] for purposes of better illustration. Similarly, PP1[3:0], PP2[3:0] and PP3[3:0] are the new partial product outputs corresponding to the rows one, two and three.

Example 9.23 Design a full adder tree responsible for adding every partial product in the partial product tree for a four-bit array multiplier.

After generating partial products, the next design step is to add partial product elements column by column to generate SUM outputs, SUM[7:0], while propagating carry bits for higher order columns. Following the naming convention in Fig. 9.58, all 16 partial product elements are fed to the full adder tree in Fig. 9.59. The box outlined by dashed lines shows how the carry propagation takes place from one column to the next.

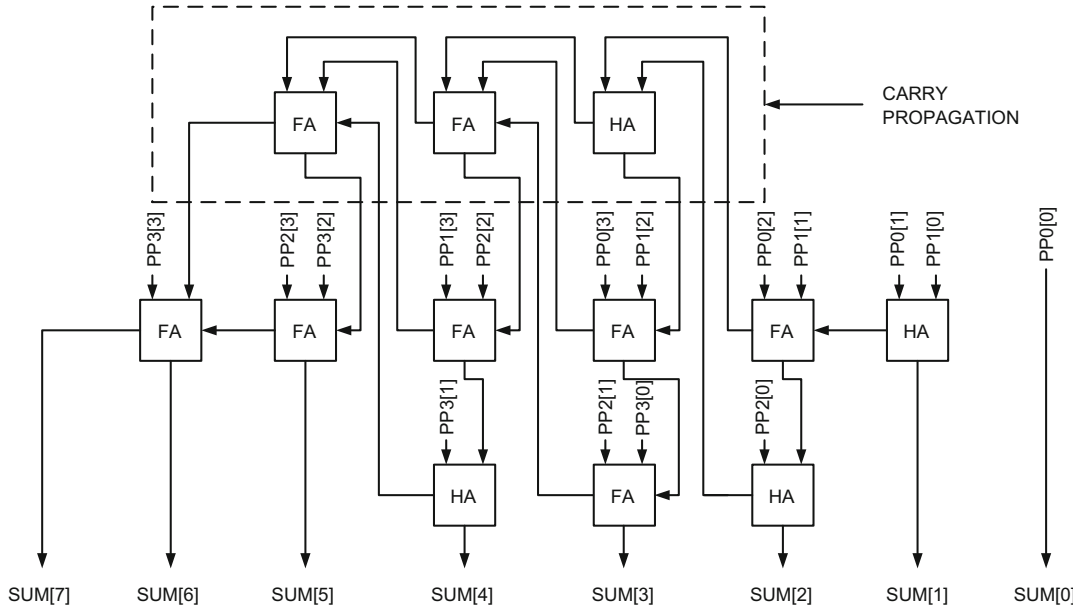


Fig. 9.59 An eight-bit propagate adder for the bit selector tree in Fig. 9.58

9.10 D Latch

The D Latch is the most basic memory element in logic design. It has a data input, D, a clock input and a data output, Q, as shown at the top portion of Fig. 9.60. It contains a tri-state inverter at its input stage followed by two back-to-back inverters connected in a loop configuration, which serves to store data.

The clock signal connected to the enable input of the tri-state inverter can be set either to active-high or active-low. In Fig. 9.60, the changes at the input transmit though the memory element and appear at the output during the low phase of the clock. In contrast, changes at the input are blocked during the high phase of the clock, and no data transmits to the output. Once the data is stored in the back-to-back inverter loop, it becomes stable and does not change until different data is introduced at the input. The buffer at the output stage of the latch is used to drive multiple gate inputs.

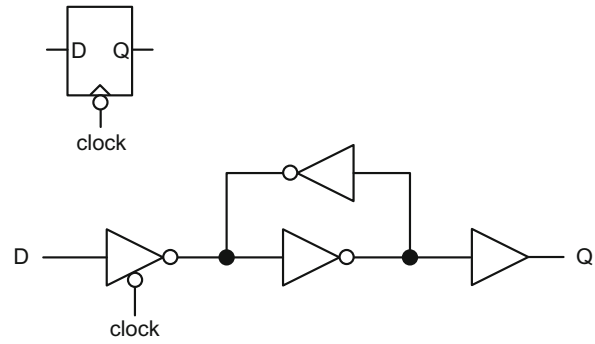


Fig. 9.60 Logic and circuit diagrams of a D latch

The operation of the D latch is shown in Fig. 9.61. During the low phase of the clock, the tri-state inverter is enabled. The new data transmits through the tri-state inverter, overwrites the old data in the back-to-back inverter stage, and reaches the output. When the clock switches to its high phase, the input-output data transmission stops because the tri-state buffer is disabled and blocks any new data transfer. Therefore, if certain data needs to be retained in the latch, it needs to be stored in the latch some time before the rising edge of the clock. This time interval is called the set-up time, t_S, and it is approximately equal to the sum of delays through the tri-state inverter and the inverter in the memory element. At the high phase of the clock, the data stored in the latch can no longer change as shown in Fig. 9.61.

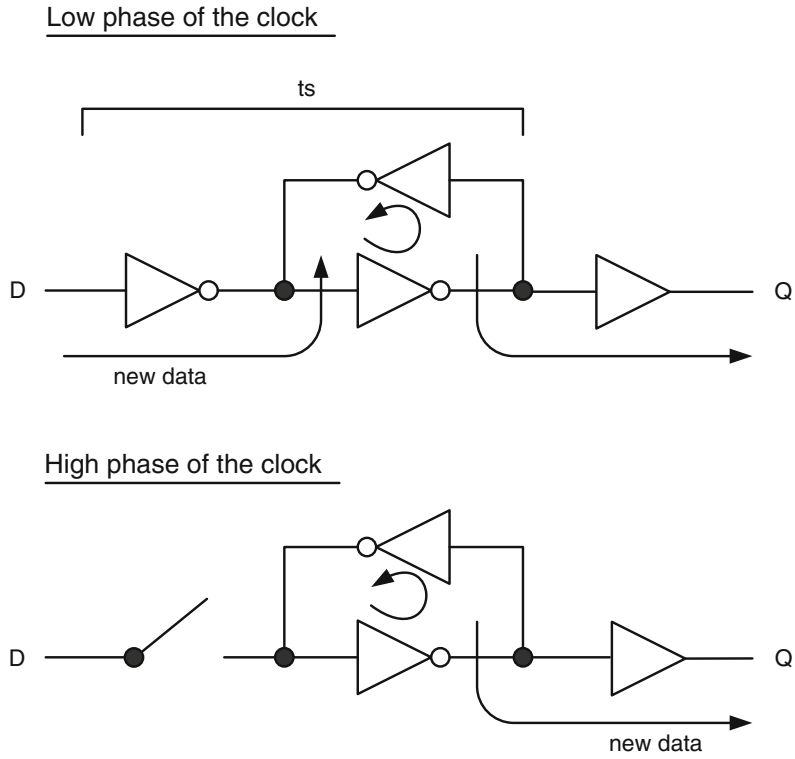


Fig. 9.61 Operation of D latch

9.11 Timing Methodology Using D Latches

Timing in logic systems is maintained by pipeline structures. A pipeline consists of combinational logic blocks bounded by memory elements as shown at the top portion of Fig. 9.62. The main purpose of pipelines is to process several data packets within the same clock cycle and maximize the data throughput.

To illustrate the concept of pipeline, latches are used as memory elements in the pipeline shown in Fig. 9.62. In every latch boundary, data propagates from one combinational logic stage to the next at the high and low phases of clock.

The bottom figure in Fig. 9.62 shows the timing diagram of a data transfer for a set of data packets ranging from D1 to D3 at the IN terminal. The first data packet, $D1^0$, retains its original value during the high phase of the clock (Cycle 1H) at the node A. $D1^0$ then propagates through the T1 stage and settles at the node B in a modified form, $D1^1$, sometime before the falling edge of the clock. Similarly, $D1^1$ at the node C retains its value during the low phase of the clock while its processed form, $D1^2$, propagates through the T2 stage and arrives at the node D before the rising edge of the clock. This data is processed further in the T3 stage and transforms into $D1^3$ before it becomes available at the OUT terminal at the falling edge of the clock in Cycle 2L.

Similarly, the next two data packets, $D2^0$ and $D3^0$, are also fed into the pipeline at the subsequent positive clock edges. Both of these data propagate through the combinational logic stages, T1, T2 and T3, and become available at the OUT terminal at the falling edge of Cycle 3L and Cycle 4L, respectively.

The total execution time for all three data packets takes four clock cycles according to the timing diagram in Fig. 9.62. If we were to remove all the latch boundaries between nodes A and F and wait until all three data packets, D1, D2 and D3, were processed through the sum of the three combinational logic stages, T1, T2 and T3, the total execution time would have been 3 x 3 = 9 clock cycles as each combinational logic stage requires one clock cycle to process data. Therefore, pipelining can be used advantageously to process data in a shorter amount of time and increase data throughput.

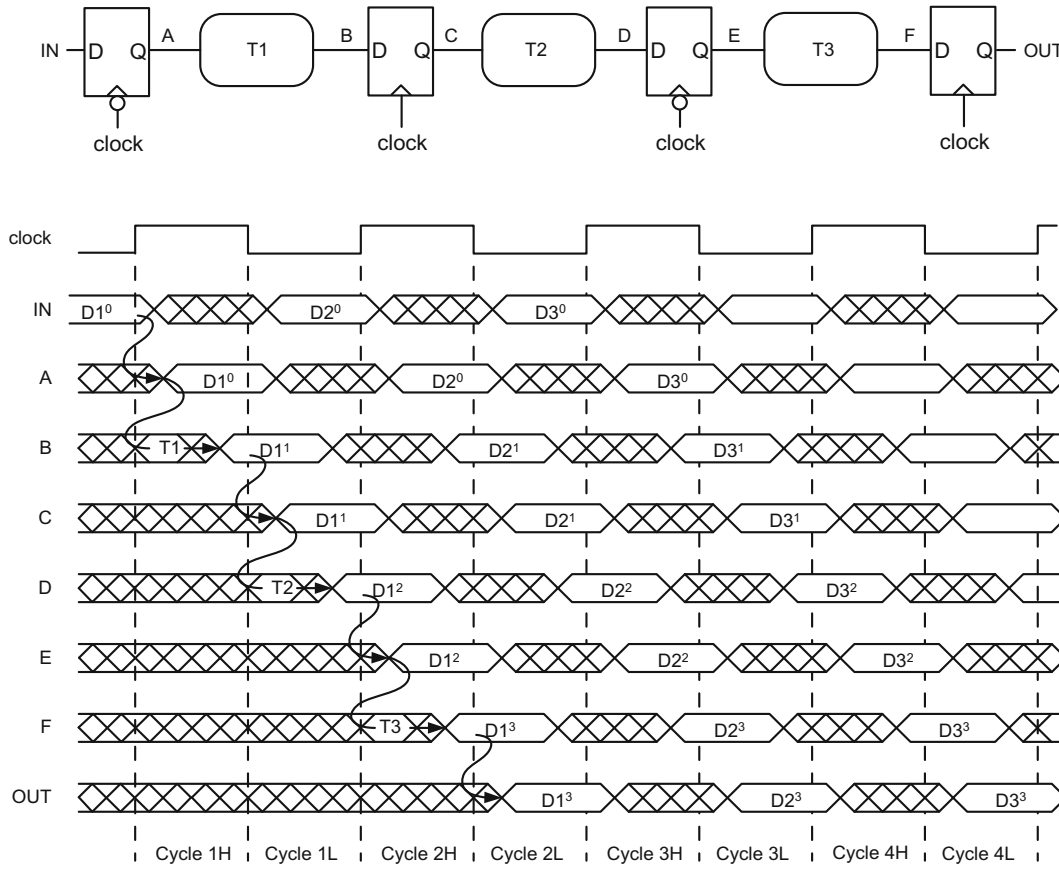


Fig. 9.62 Timing methodology using D latches

9.12 D Flip-Flop

D flip-flop is another important timing element in logic design to maintain timing while transferring data from one combinational logic block to the next.

Similar to a latch, a flip-flop also has a data input, D, a clock input and a data output, Q, as shown at the top portion of Fig. 9.63.

The bottom portion of Fig. 9.63 shows the circuit schematic of a typical flip-flop which contains two latches in series. The first latch has an active-low clock input, and it is called the master. The second latch has an active-high clock input, and it is called the slave. The master accepts new data during the low phase of the clock, and transfers this data to the slave during the high phase of the clock.

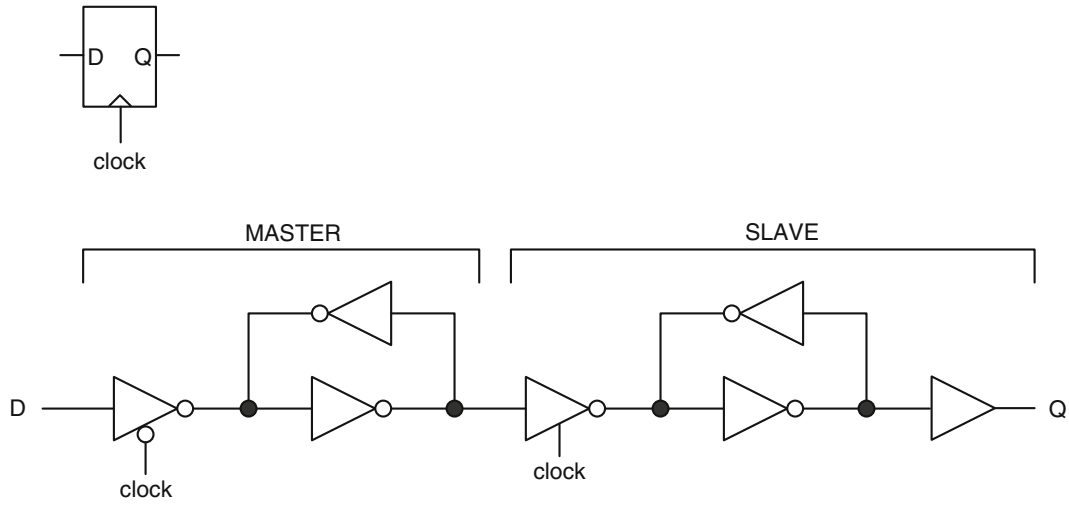


Fig. 9.63 Logic and circuit diagrams of a D flip-flop

Figure 9.64 shows the timing attributes of a flip-flop. The set-up time, t_S, is the time interval for the valid data to arrive and settle in the master latch before the rising edge of the clock. The hold time, t_H, is the time interval after the positive edge of the clock when the valid data needs to be kept steady and unchanged. The data stored in the master latch propagates through the slave latch and becomes the flip-flop output some time after the rising edge of the clock. This is called clock-to-q delay or t_{CLKQ}.

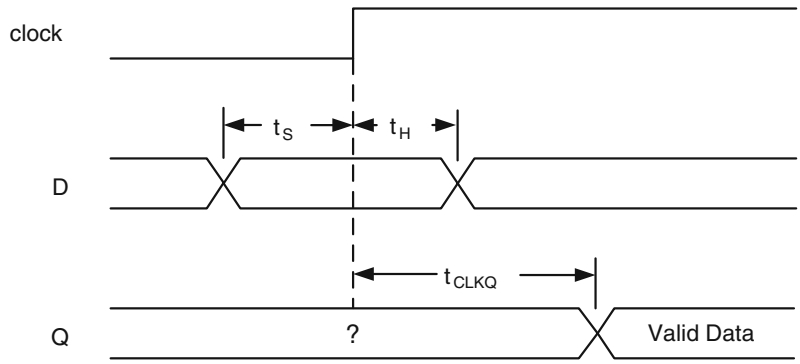


Fig. 9.64 Timing attributes of a D flip-flop

The operation of the flip-flop in two different phases of clock is shown in Fig. 9.65. During the low phase of the clock, new data enters the master latch, and it is stored. This data cannot propagate beyond the master latch because the tri-state inverter in the slave latch acts as an open circuit during the low phase of the clock. The flip-flop output reveals only the old data stored in the slave latch. When the clock goes high, however, the new data stored in the

master latch transmits through the slave and arrives at the output. One can calculate approximate values of t_S and t_{CLKQ} using the existing gate delays in the flip-flop.

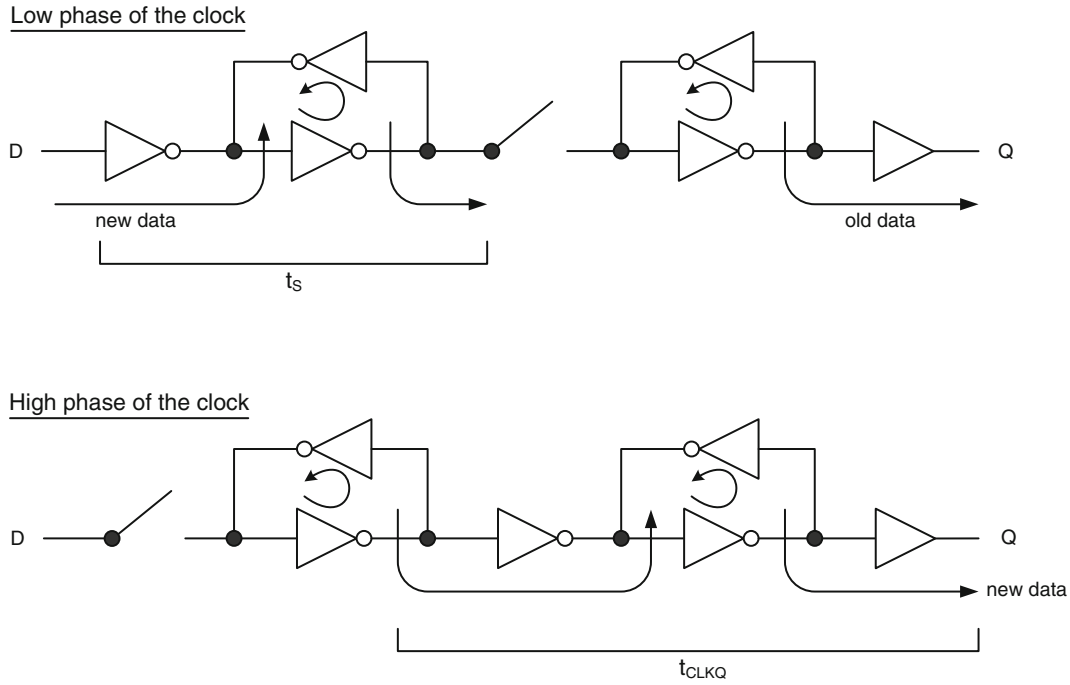


Fig. 9.65 Operation of D flip-flop

9.13 Timing Methodology Using D Flip-Flops

Data propagation through a pipeline with D flip-flops is shown in Fig. 9.66. The bottom figure in Fig. 9.66 shows the timing diagram of a data transfer for a set of data packets ranging from D1 to D3 at the IN terminal.

The first data packet, $D1^0$, at the IN terminal has to be steady and valid during the set-up and hold periods of the flip-flop, but it is free to change during the remaining part of the clock period as shown by oscillatory lines. Once the clock goes high, the valid $D1^0$ starts to propagate through the combinational logic block of T1 and reaches the second flip-flop boundary. The processed data, $D1^1$, has to arrive at the second flip-flop input, B, no later than the set-up time of the flip-flop. Otherwise, the correct data cannot be latched. $D1^1$ propagates through the second (T2) and third (T3) combinational logic stages, and becomes $D1^2$ and $D1^3$, respectively, before exiting at the OUT terminal as shown in the timing diagram in Fig. 9.66.

The subsequent data packets, $D2^0$ and $D3^0$ are similarly fed into the pipeline stage at consecutive positive clock edges following $D1^0$. They are processed and modified by the T1, T2 and T3 combinational logic stages as they propagate through the pipeline, and emerge at the OUT terminal.

The total execution time for three input data packets, $D1^0$, $D2^0$ and $D3^0$, takes six clock cycles, including the initial three clock cycle build-up period before $D1^3$ emerges at the OUT terminal. If we were to remove all the flip-flop boundaries between the nodes A and F, and wait for these three data packets to be processed without any pipeline structure, the total execution time would have been 3 x 3 = 9 clock cycles, assuming each T1, T2 or T3 logic stage imposes one clock cycle delay.

Once again, the pipelining technique considerably reduces the overall processing time and data throughput whether the timing methodology is latch-based or flip-flop-based.

The advantage of using latches as opposed to flip-flops is to be able to borrow time from neighboring stages. For example, the propagation delay in T1 stage can be extended at the expense of shortening the propagation delay in T2. This flexibility does not exist in a flip-flop based design in Fig. 9.66.

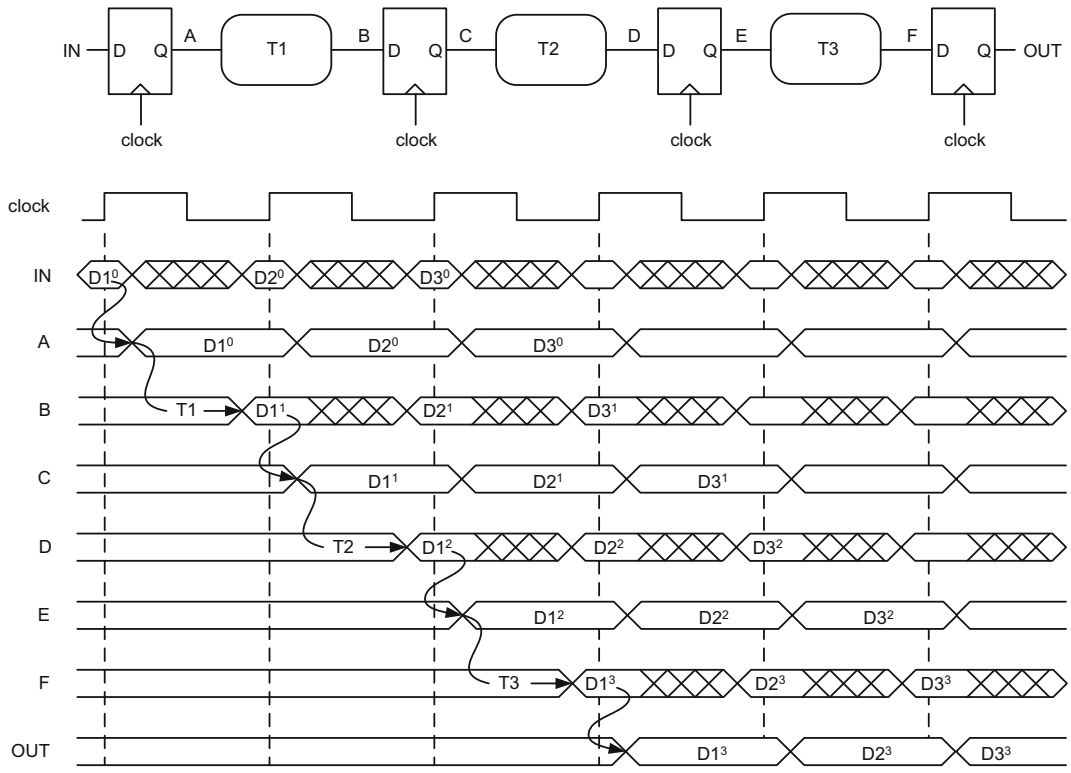


Fig. 9.66 Timing methodology using D flip-flops

9.14 Timing Violations

Although pipelining scheme helps to reduce the overall data processing time, we still need to watch out for possible timing violations because occasionally data does not comply with the requirements of flip-flop (or latch) set-up and hold times at pipeline boundaries.

This section examines the set-up and hold timing violations in a pipeline controlled by flip-flops. Figure 9.67 shows a section of a pipeline where a combinational logic block with a propagation delay of T_{COMB} is sandwiched between two flip-flops. At the rising edge of the clock, the valid and steady data that meets the set-up and hold times is fed into the pipeline from the IN terminal. After t_{CLKQ} delay at the node A, the data propagates through the combinational logic block and emerges at the node B as shown in the timing diagram. However, the data arrives at the node B past the allocated set-up time of the flip-flop and creates a set-up violation. The amount of violation is dependent on the clock period and is calculated as follows:

$$\text{Set-up violation} = t_S - [T_C - (t_{CLKQ} + T_{COMB})] \tag{9.1}$$

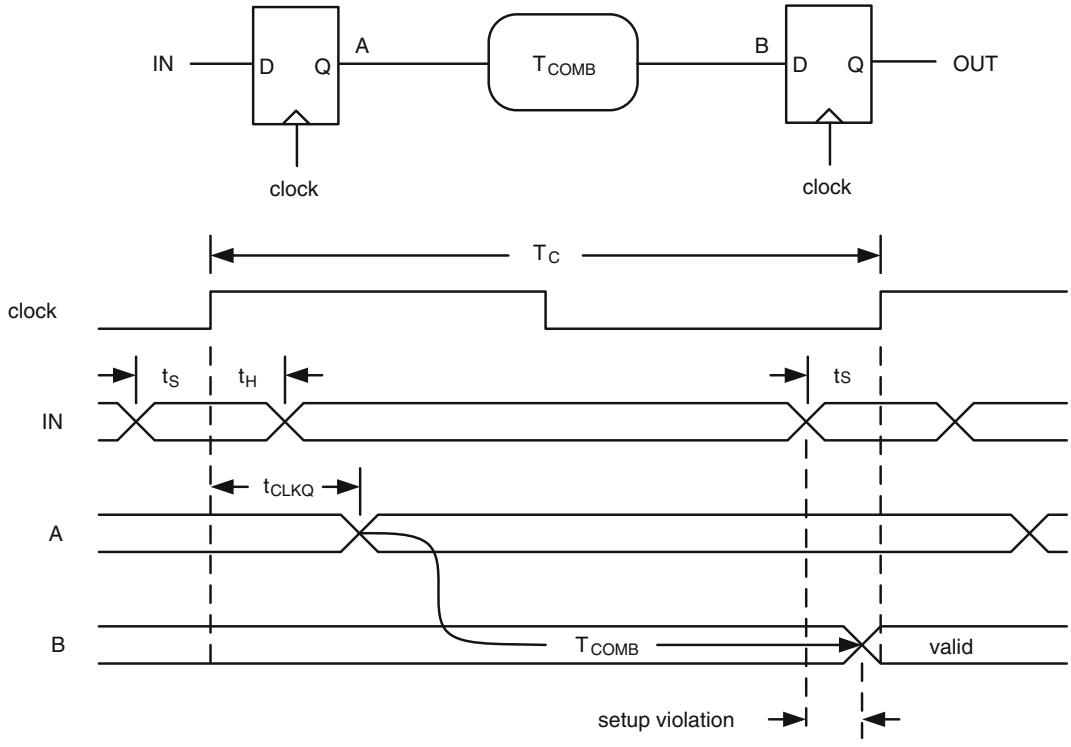


Fig. 9.67 Setup violation

Figure 9.68 describes the hold time violation where the clock shifts by T_{CLK} due to an unexpected delay in the clock line. In the timing diagram, the valid data is fed into the pipeline from the IN terminal, and arrives at the node B after t_{CLKQ} and T_{COMB} delays. Due to the shifted clock, the data arrives at the node B very early. This creates a substantial set-up time slack equal to $(T_C + T_{CLK} - t_S - t_{CLKQ} - T_{COMB})$ but produces a hold time violation at the delayed clock edge. The amount of violation is dependent on the clock delay and is calculated as follows:

$$\text{Hold violation} = (T_{CLK} + t_H) - (t_{CLKQ} + T_{COMB}) \tag{9.2}$$

Set-up violations can be recovered simply by increasing the clock period, T_C. However, there is no easy way to fix hold violations. We need to search for hold time violation at each flip-flop input, and add additional delay to the combinational logic block terminating at the particular flip-flop in order to avoid the violation.

The schematic in Fig. 9.69 examines the timing ramifications of two combinational logic blocks with different propagation delays merging into one block in a pipeline stage. The data

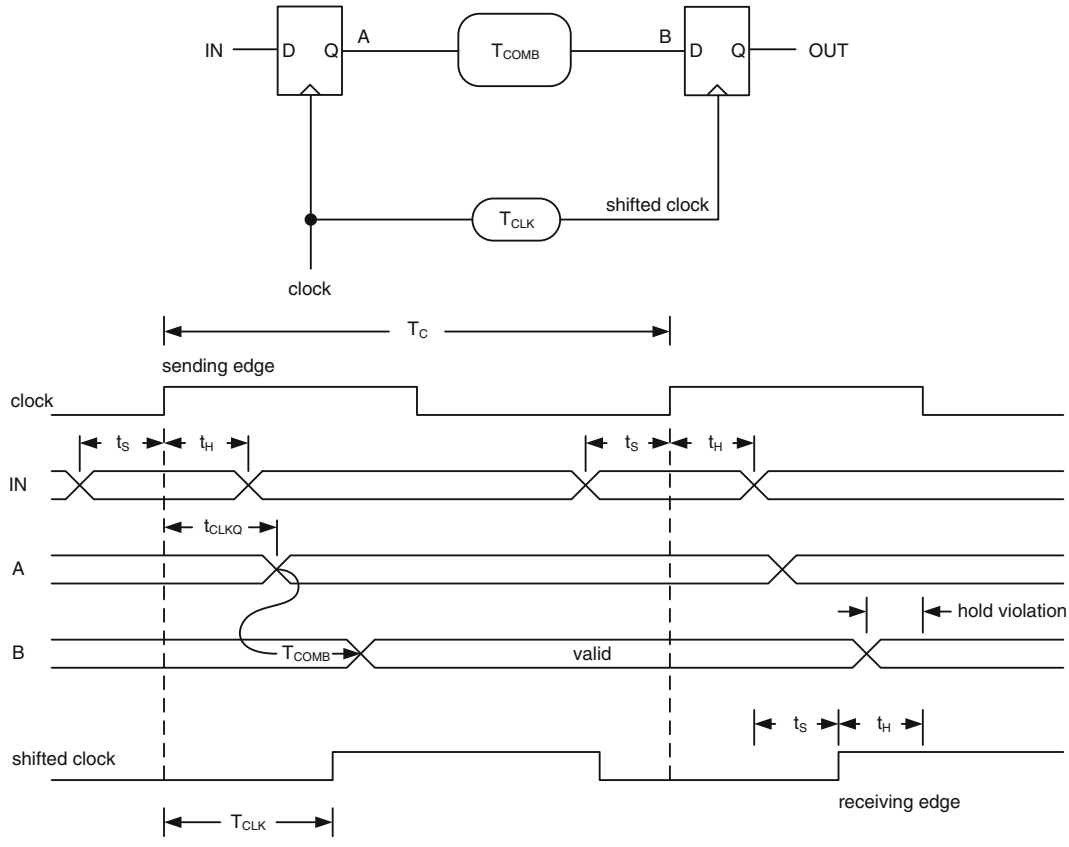


Fig. 9.68 Hold violation

at the node A arrives at the node C much earlier than the data arriving at the node D as shown in the timing diagram. The data at the nodes C and D propagate through the last combinational logic block and arrive at the node E. This scenario creates minimum and maximum paths at the node E. We need to focus on the maximum path, (T2 + T3), when examining the possibility of a set-up violation and the minimum path, (T1 + T3), when examining the possibility of a hold violation at the next clock edge.

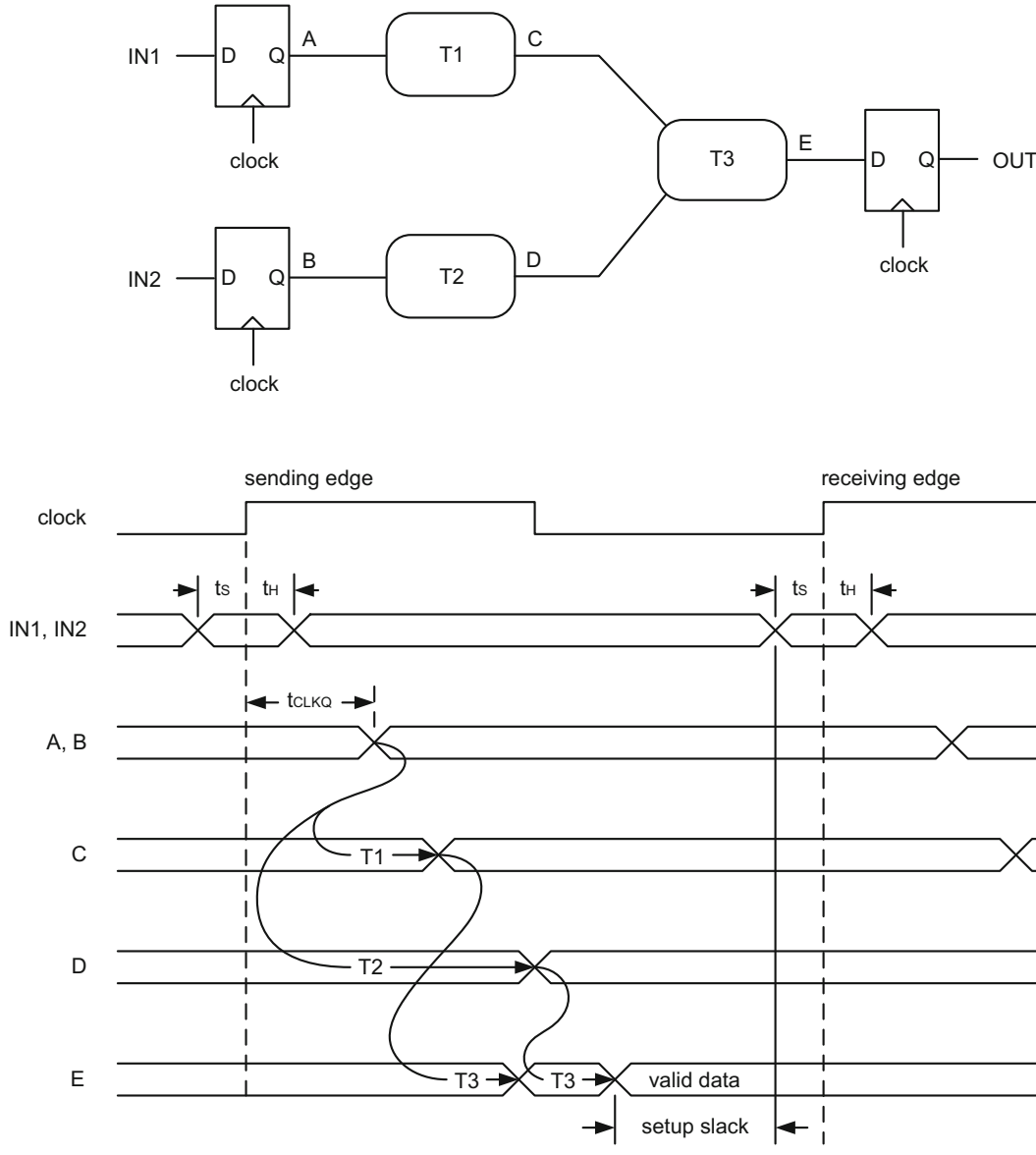


Fig. 9.69 A timing example combining two independent data-paths

To further illustrate the timing issues through multiple combinational logic blocks, an example with logic gates is given in Fig. 9.70. In this example, the inputs of a one-bit adder are connected to the nodes A and B. The adder is bypassed with the inclusion of a 2-1 MUX which selects either the output of the adder or the bypass path by a selector input, SEL.

The propagation delays of the inverter, T_{INV}, and the two-input NAND gate, T_{NAND2}, are given as 100 ps and 200 ps, respectively. The set-up, hold and clock-to-q delays are also given as 100, 0 and 300 ps, respectively.

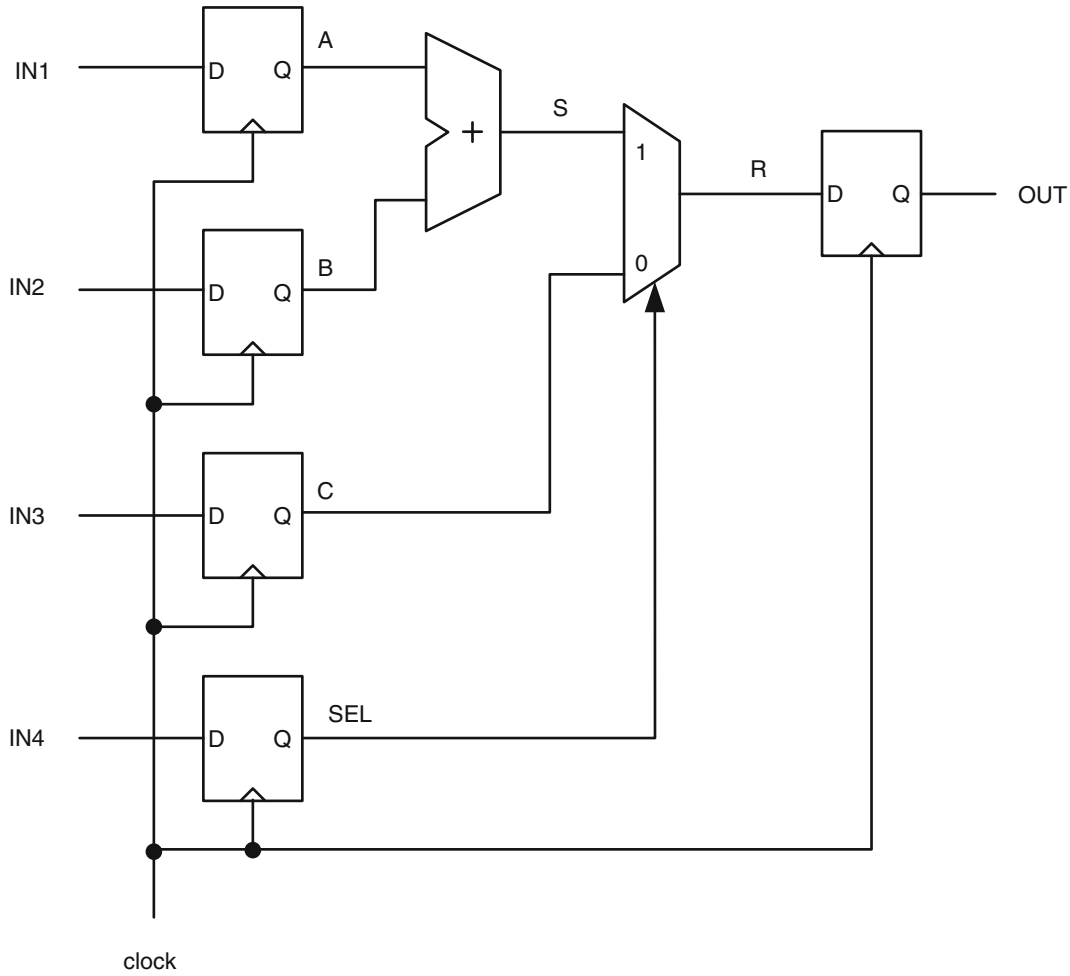


Fig. 9.70 An example with multiple propagation paths

One-bit full adder and the 2-1 MUX are shown in terms of inverters and two-input NAND gates in Fig. 9.71. We obtain a total of seven propagation paths all merging at the node R. However, we only need to search for the maximum and the minimum paths to locate possible set-up and hold violations.

The maximum delay path consists of the inverter 1, and the series combination of four two-input NAND gates numbered as 1, 3, 4 and 6 shown in the schematic. This path results in a total delay of 900 ps. The minimum delay path, on the other hand, contains two two-input NAND gates numbered as 5 and 6, and it produces a delay of 400 ps. Placing these propagation delays in a timing diagram in Fig. 9.72 ultimately yields a set-up slack of 100 ps at the node R when a clock period of 1400 ps is used. There is no need to investigate for hold violations because there is no shift in the clock edge. The data emerges at the nodes A, B, C and SEL after t_{CLKQ} = 300 ps where t_H = 0 ps.

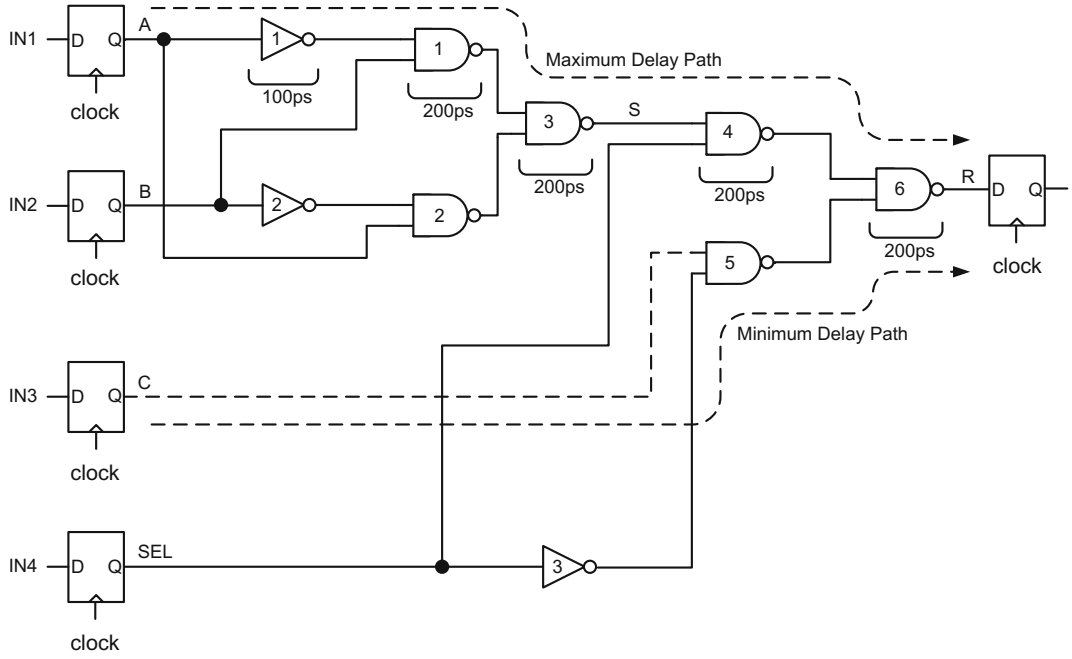


Fig. 9.71 Logic circuit of Fig. 9.70 showing maximum and minimum paths

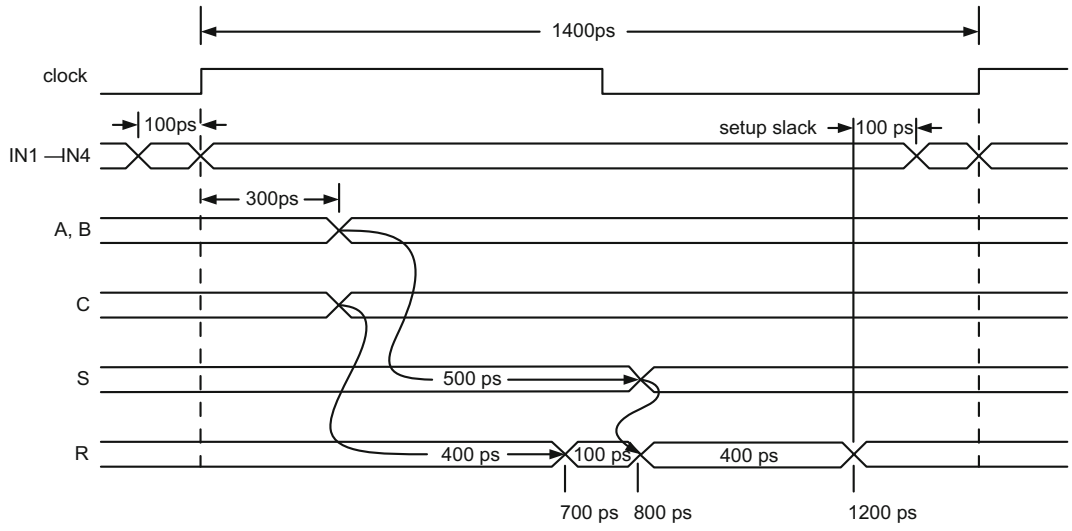


Fig. 9.72 Timing diagram of the circuit in Fig. 9.71

9.15 Register

While a flip-flop can hold data only for one clock cycle until a new data arrives at the next clock edge, a register can hold the same data perpetually until the power is turned off.

Figure 9.73 shows the circuit diagram of a one-bit register composed of a flip-flop and a 2-1 MUX. The Write Enable pin, WE, is a selector input to the 2-1 MUX and transfers new data from the IN terminal to the flip-flop input when WE = 1. If the WE input is at logic 0, any attempt to write a new data to the register is blocked; the old data stored in the flip-flop simply circulates around the feedback loop from one clock cycle to the next.

The timing diagram at the bottom of Fig. 9.73 describes the operation of the one-bit register. The data at the IN terminal is blocked until the WE input becomes logic 1 in the middle of the second clock cycle. At this point, the new data is allowed to pass through the 2-1 MUX and updates the contents of the register at the rising edge of the third clock cycle. Since the WE input is pulled to logic 0 before the end of the third clock cycle, the register output, OUT, stays at logic 1 during the fourth clock cycle.

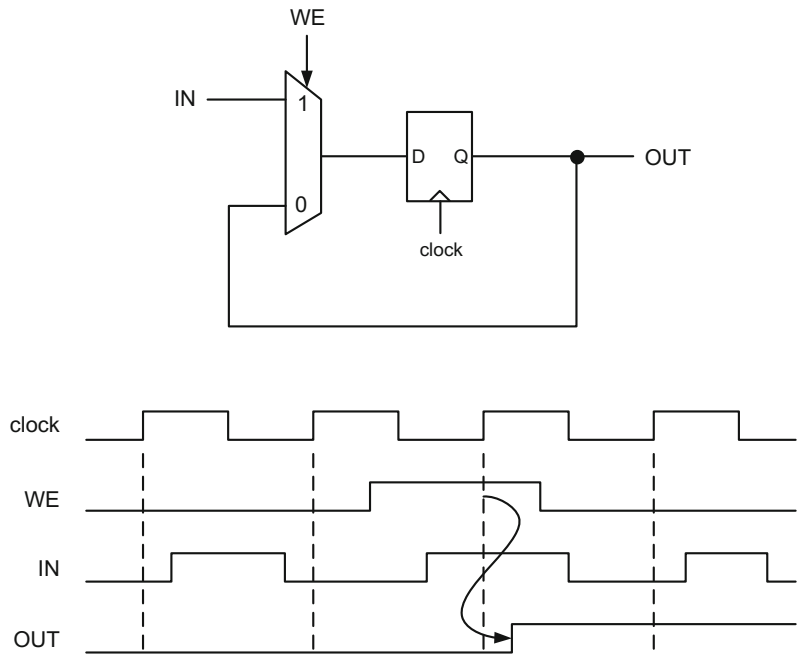


Fig. 9.73 One-bit register and a sample timing diagram

A 32-bit register shown in Fig. 9.74 is composed of 32 one-bit registers. All 32 registers have a common clock and WE input. Therefore, any new 32-bit data introduced at the register input changes the contents of the register at the rising edge of the clock if the WE input is kept at logic 1.

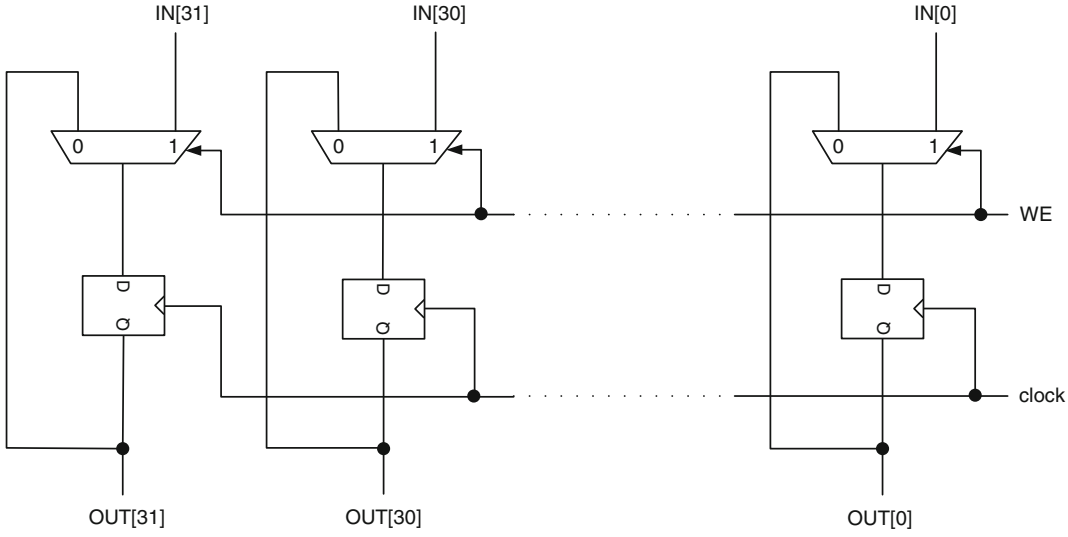


Fig. 9.74 32-bit register

9.16 Shift Register

The shift register is a particular version of an ordinary register and specializes in shifting data to the right or left according to design needs.

Figure 9.75 shows the circuit schematic of a four-bit shift register that shifts serial data at the IN terminal to the left at every positive clock edge.

The operation of this shift register is explained in the timing diagram in Fig. 9.76. In cycle 1, SHIFT = 0. Therefore, the change at the IN terminal during this cycle does not affect the register outputs. However, when the SHIFT input transitions to logic 1 in the middle of cycle 2, it allows IN = 1 to pass to the least significant output bit, OUT[0], at the beginning of the third clock cycle. From the middle of cycle 2 to cycle 13, SHIFT is kept at logic 1. Therefore, any change at the IN node directly transmits to the OUT[0] node at the positive edge of every clock cycle. The other outputs, OUT[1], OUT[2] and OUT[3], produce the delayed version of the OUT[0] one clock cycle apart from each other because the output of a lesser significant bit is connected to the input of a greater significant bit in the shift register.

When the SHIFT input transitions to logic 0 from the middle of cycle 13 to cycle 17, the shift register becomes impervious to any new data entry at the IN terminal, and retains the old values from the beginning of cycle 13 to cycle 18 as shown in Fig. 9.76. From the middle of cycle 17, the SHIFT input becomes logic 1 again, and the shift register distributes new data entries at the IN terminal to its outputs.

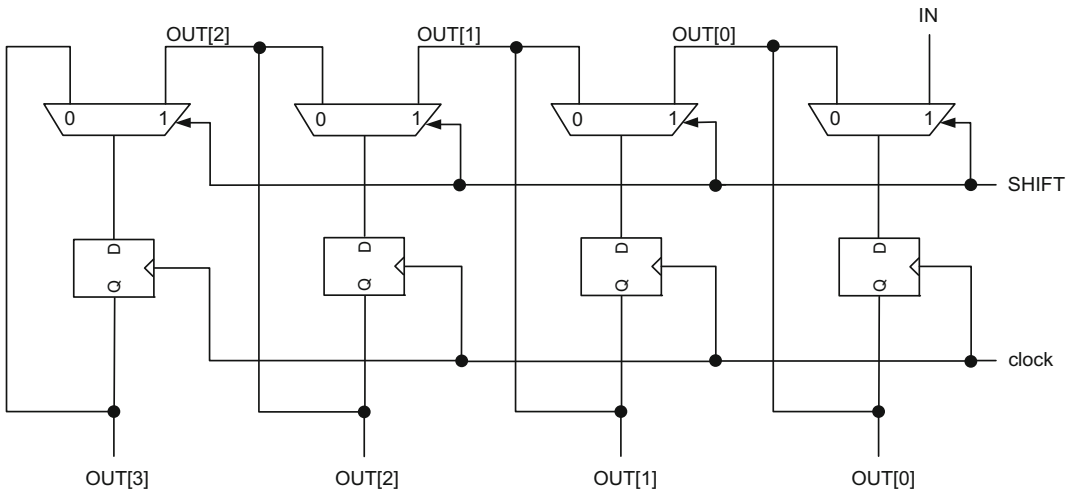


Fig. 9.75 Four-bit shift register

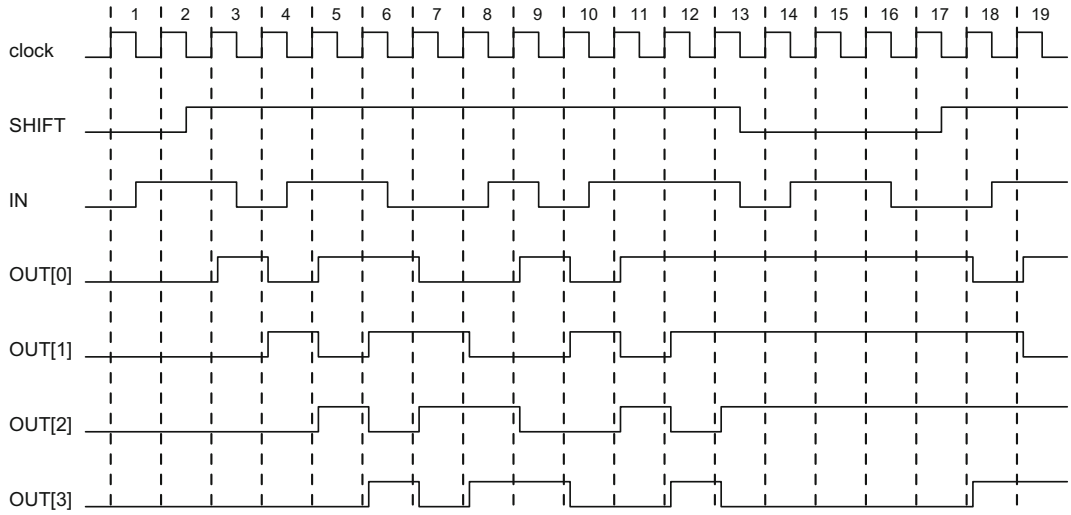


Fig. 9.76 A sample timing diagram of the four-bit shift register in Fig. 9.75

9.17 Counter

The counter is a special form of a register which is designed to count up (or down) at each rising edge of the clock.

The counter in Fig. 9.77 shows a typical 32-bit up-counter with two control inputs, COUNT and LOAD. COUNT = 1 enables the counter to count upwards at the rising edge of each clock cycle. LOAD = 1 loads new data at the IN terminal to the counter. Once loaded, the counter output, OUT[31:0], increments by one at every positive clock edge until all of its outputs become logic 1. The next increment automatically resets all the counter outputs to logic 0. When LOAD = COUNT = 0, the counter neither loads new data nor is able to count upwards; it stalls and repeats its old output value.

The sample timing diagram at the bottom of Fig. 9.77 illustrates its operation. Prior to the first clock edge, the LOAD input is at logic 1 which allows an input value, IN[31:0] = 3, to be stored in the counter. This results in OUT[31:0] = 3 at the positive edge of the first clock cycle. During the first clock cycle, LOAD = 0 and COUNT = 1 start the up-count process. The contents of the output, OUT[31:0] = 3, subsequently increments by one. The result, 3 + 1 = 4, passes through the C-port of the 3-1 MUX and arrives at the flip-flop inputs. At the positive edge of the second clock cycle, this new value overwrites the old registered value, and the OUT[31:0] node becomes equal to 4. In the next cycle, the counter goes through the same process and increments by one. However, in the same cycle, the LOAD input also transitions to logic 0, and turns on the I-port of the 3-1 MUX. This new port prevents any new data from entering the up-counter, but keeps the old data in the following clock cycles. As a result, the counter output stops incrementing and stalls at the value of OUT[31:0] = 5.

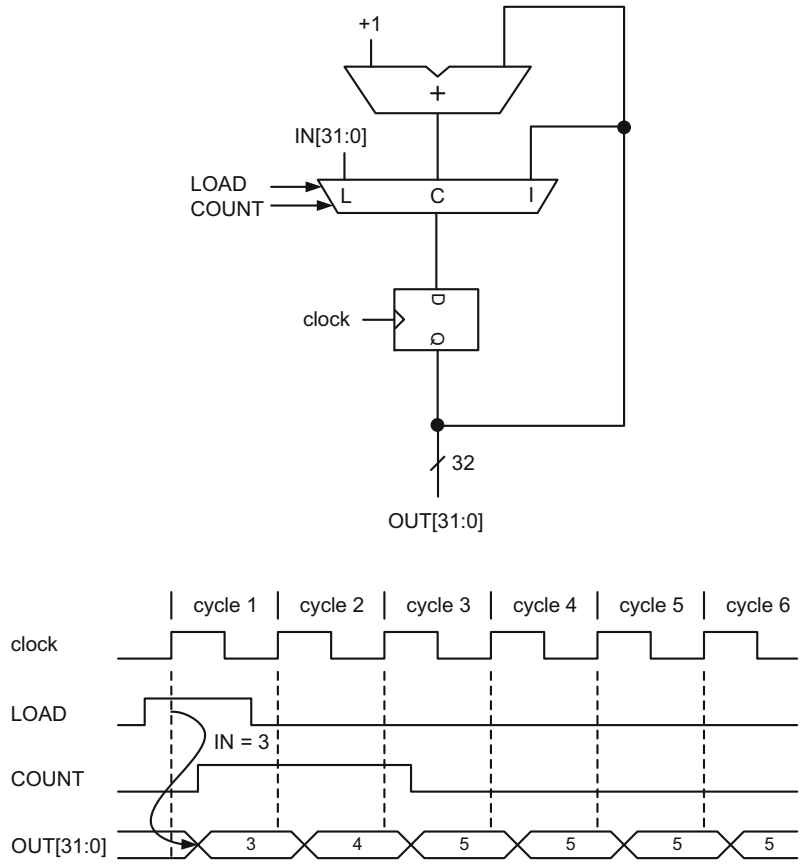


Fig. 9.77 A 32-bit counter and a sample timing diagram

9.18 Moore-Type State Machine

State machines are mainly used in digital designs to control the proper data-flow. Their topology is mainly composed of one or multiple flip-flops and feedback loop(s) that connect flip-flop outputs to flip-flop inputs. There are two types of state machines: Moore-type and Mealy-type.

Figure 9.78 shows the Moore-type state machine topology consisting of a flip-flop and a feedback loop. In this configuration, the feedback loop includes a combinational logic block that accepts both the flip-flop output and external inputs. If there are multiple flip-flops, the combination of all flip-flop outputs constitutes the "present" state of the machine. The combination of all flip-flop inputs is called the "next" state because at the positive edge of the clock these inputs become the flip-flop outputs, and form the "next" present state. Flip-flop outputs may be processed further by an additional combinational logic block to form present state outputs.

The basic state diagram of a Moore machine, therefore, includes the present state (PS) and the next state (NS) as shown on the right hand side of Fig. 9.78. The machine can transition

from the PS to the NS if the required present state inputs are supplied. The outputs of the Moore machine are solely generated by the present state. Therefore, machine outputs emerge only from the present and the next states as shown in the basic state diagram.

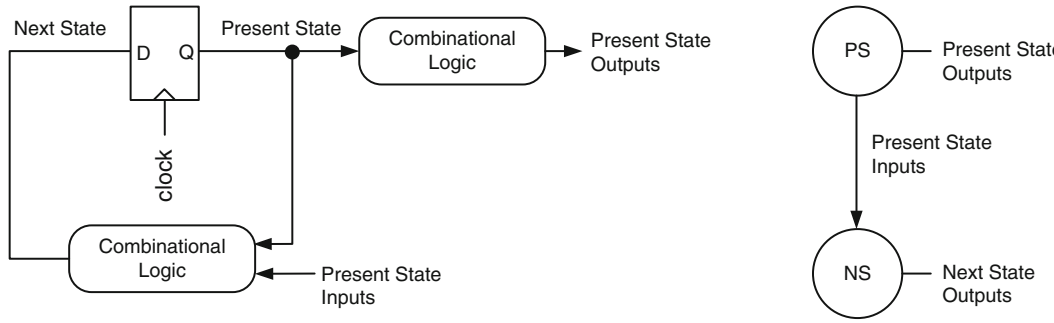


Fig. 9.78 Block diagram and state representation of Moore machine

The state diagram in Fig. 9.79 shows an example of a Moore-type machine with four states. Note that every state-to-state transition in the state diagram requires a valid present state input entry, and every node generates one present state output.

The state 0, S0, produces a present state output, OUT = 1, regardless of the value of the present state input, IN. When IN = 1, the state S0 transitions to the next state S1; otherwise, it maps onto itself. The state S1 produces OUT = 2. Its next state becomes the state S1 if IN = 0; otherwise, it transitions to a new state S2. The state S2 also produces a present state output, OUT = 3, and transitions to the state S3 if IN = 1. The state S2 remains unchanged if IN = 0. In the fourth and final state S3, the present state output, OUT = 4, is produced. The machine stays in this state if IN stays at 0; otherwise, it goes back to the state S1.

The present state inputs and outputs of this Moore machine and its states can be tabulated in a table called the "state table" given in Fig. 9.80. In this table, the first column under PS lists all the possible present states of the state diagram in Fig. 9.79. The middle two columns contain the next state entries for IN = 0 and IN = 1. The last column lists the present state outputs, one for each present state.

The binary state assignment is performed in Fig. 9.81 where only one bit is changed between adjacent states.

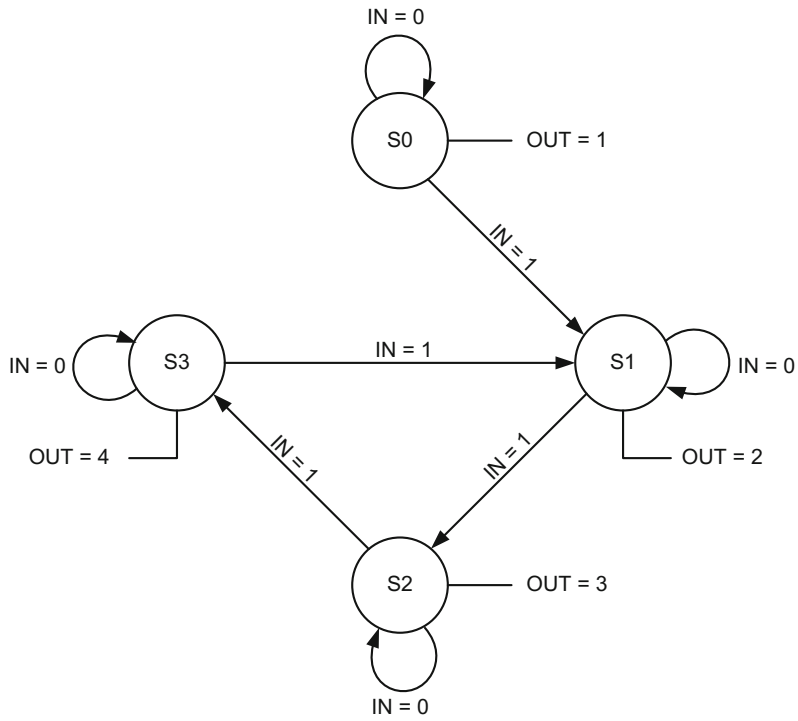


Fig. 9.79 State diagram of a Moore machine with four states

PS	NS		OUT
	IN = 0	IN = 1	
S0	S0	S1	1
S1	S1	S2	2
S2	S2	S3	3
S3	S3	S1	4

Fig. 9.80 State table of the Moore machine in Fig. 9.79

States	NS1	NS0
S0	0	0
S1	0	1
S2	1	1
S3	1	0

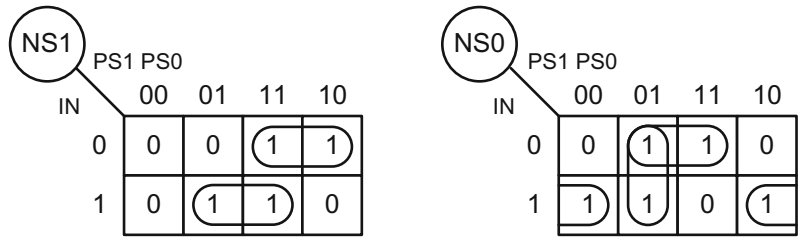
Fig. 9.81 Bit representations of states S0, S1, S2 and S3

The binary form of the state table in Fig. 9.80 is reconstructed in Fig. 9.82 according to the state assignment in Fig. 9.81. This table, called the "transition table", includes the binary representation of the next state and the present state outputs.

		IN = 0		IN = 1				
PS1	PS0	NS1	NS0	NS1	NS0	OUT2	OUT1	OUT0
0	0	0	0	0	1	0	0	1
0	1	0	1	1	1	0	1	0
1	1	1	1	1	0	0	1	1
1	0	1	0	0	1	1	0	0

Fig. 9.82 Transition table of the Moore machine in Fig. 9.79

Forming this machine's K-maps for the NS0, NS1, OUT0, OUT1 and OUT2 requires grouping all the input terms, PS1, PS0 and IN, according to the table in Fig. 9.82. The K-maps and their corresponding SOP representations are shown in Fig. 9.83.



$$NS1 = PS0.IN + PS1.\overline{IN}$$

$$NS0 = PS0.\overline{IN} + \overline{PS1}.PS0 + \overline{PS0}.IN$$

$$= (PS0 \oplus IN) + \overline{PS1}.PS0$$

$$OUT2 = PS1.\overline{PS0}$$

$$OUT1 = \overline{PS1}.PS0 + PS1.PS0 = PS0$$

$$OUT0 = \overline{PS1}.\overline{PS0} + PS1.PS0 = \overline{PS0 \oplus PS0}$$

Fig. 9.83 K-maps and SOP expressions for the Moore machine in Fig. 9.79

The next step is to generate the circuit diagram that produces all five outputs of the Moore machine according to the SOP expressions in Fig. 9.83. This circuit diagram is given in Fig. 9.84.

In order to generate this circuit, the individual combinational logic blocks for the NS0 and NS1 must be constructed first in terms of PS0, PS1 and IN. Then each NS0 and NS1 node is connected to the corresponding flip-flop input to form the feedback loops of the state machine. The logic blocks for the OUT0, OUT1 and OUT2 are generated directly from PS0 and PS1.

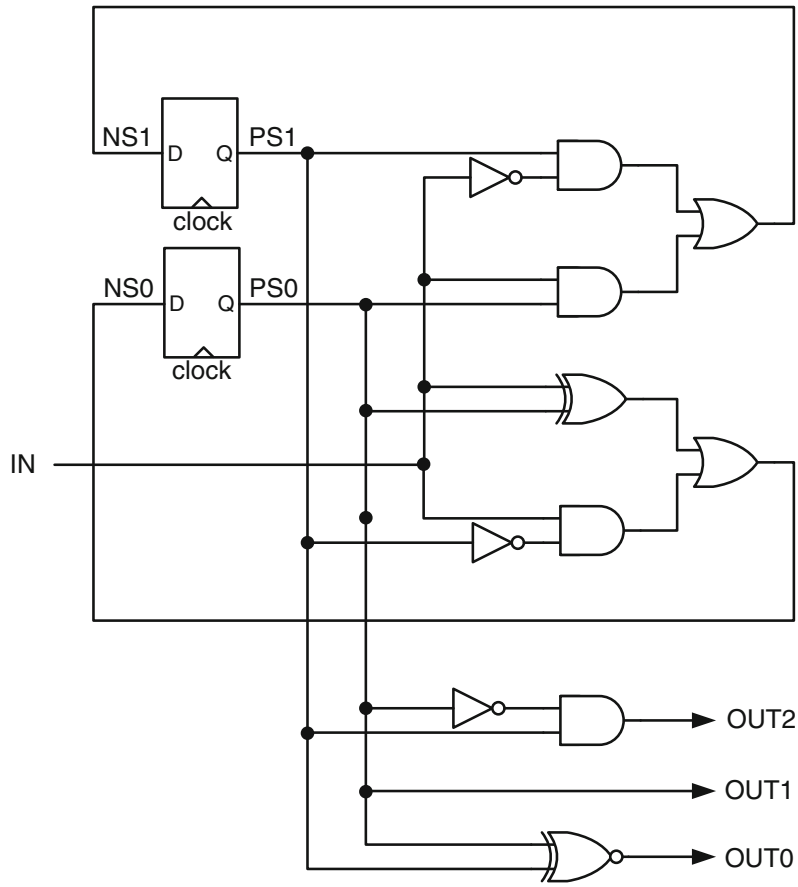


Fig. 9.84 Logic circuit of the Moore machine in Fig. 9.79

9.19 Mealy-Type State Machine

The Mealy-type state machine shares the same circuit topology as the Moore-type machine. The machine contains one or more flip-flops and feedback loop(s) as shown in Fig. 9.85. However, present state outputs are generated from the combinational logic block in the feedback loop rather than from present states in the Moore-type machines.

As a result of this topology, the basic state diagram of a Mealy machine includes the present state, the next state and the input condition that transition the present state to the next state as shown on the right hand side of Fig. 9.85. The present state outputs do not emerge from each present state; instead, they are functions of the present state inputs and the present state.

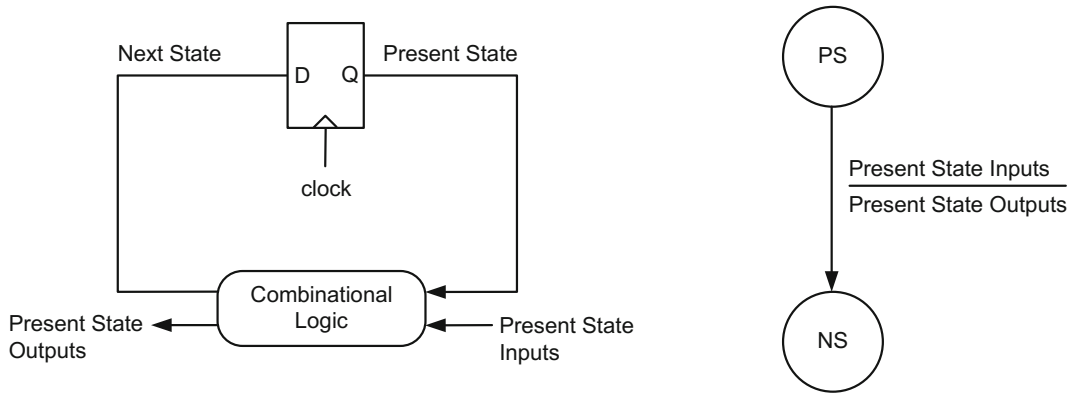


Fig. 9.85 Block diagram and state representation of Mealy-type machine

The Mealy state diagram in Fig. 9.86 exhibits similar characteristics compared to the Moore state diagram in Fig. 9.79, and all the state names and the state-to-state transitions in this diagram are kept the same for comparison purposes. However, each arrow connecting one state to the next now carries the value of the present state output as a function of the present state input as indicated in Fig. 9.85. As a result, the Mealy state table in Fig. 9.87 contains two separate columns that tabulate the values of NS and OUT for IN = 0 and IN = 1. The binary state assignment is the same as in Fig. 9.81, which results in a transition table in Fig. 9.88.

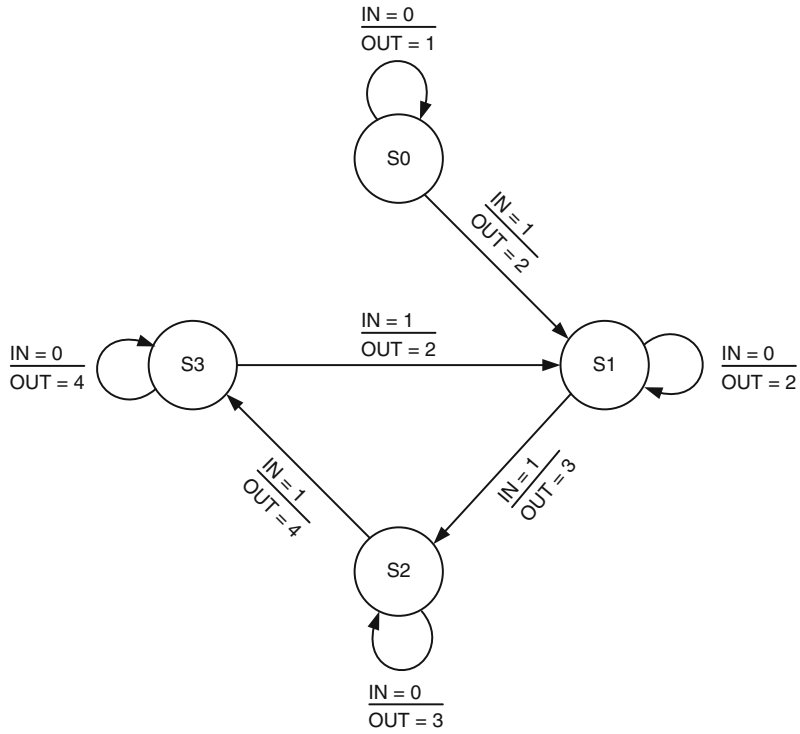


Fig. 9.86 State diagram of a Mealy-type machine with four states

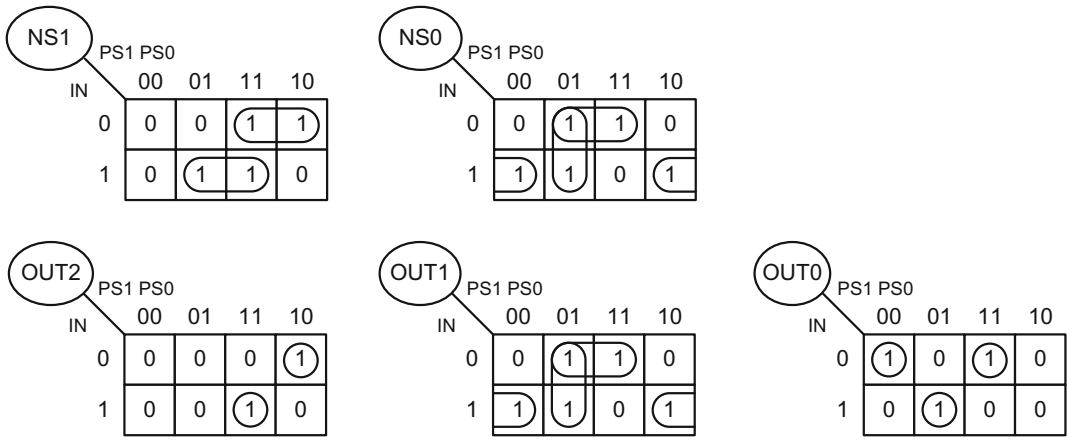
PS	NS		OUT	
	IN = 0	IN = 1	IN = 0	IN = 1
S0	S0	S1	1	2
S1	S1	S2	2	3
S2	S2	S3	3	4
S3	S3	S1	4	2

Fig. 9.87 State table of the Mealy-type machine in Fig. 9.86

PS1	PS0	IN = 0		IN = 1		IN = 0			IN = 1		
		NS1	NS0	NS1	NS0	OUT2	OUT1	OUT0	OUT2	OUT1	OUT0
0	0	0	0	0	1	0	0	1	0	1	0
0	1	0	1	1	1	0	1	0	0	1	1
1	1	1	1	1	0	0	1	1	1	0	0
1	0	1	0	0	1	1	0	0	0	1	0

Fig. 9.88 Transition table of the Mealy-type machine in Fig. 9.86

The K-maps for the NS0, NS1, OUT0, OUT1 and OUT2 are formed according to the table in Fig. 9.88 and are shown in Fig. 9.89 with the corresponding SOP expressions. Figure 9.90 shows the circuit diagram of this machine according to the expressions in Fig. 9.89. The methodology used to construct this circuit diagram is identical to the methodology used in the circuit diagram for the Moore machine in Fig. 9.84.



$$NS1 = PS0.IN + PS1.\overline{IN}$$

$$NS0 = PS0.\overline{IN} + \overline{PS1}.PS0 + \overline{PS0}.IN$$

$$= (PS0 \oplus IN) + \overline{PS1}.PS0$$

$$OUT2 = PS1.\overline{PS0}.\overline{IN} + PS1.PS0.IN = PS1.\overline{(PS0 \oplus IN)}$$

$$OUT1 = (PS0 \oplus IN) + \overline{PS1}.PS0 = NS0$$

$$OUT0 = \overline{PS1}.\overline{PS0}.\overline{IN} + \overline{PS1}.PS0.IN + PS1.PS0.\overline{IN} = \overline{PS1}.\overline{(PS0 \oplus IN)} + PS1.PS0.\overline{IN}$$

Fig. 9.89 K-maps and SOP expressions for the Mealy-type machine in Fig. 9.86

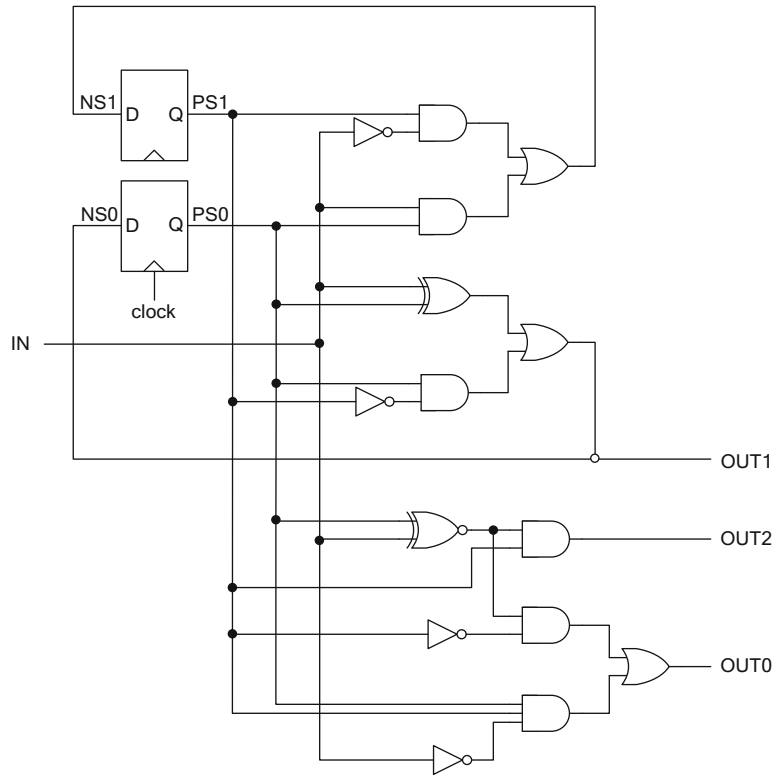


Fig. 9.90 Logic circuit of the Mealy-type machine in Fig. 9.86

9.20 Controller Design: Moore-Type State Machine Versus Counter-Decoder Scheme

Both Mealy and Moore-type state machines have practical implementation limits when it comes to design. A large ring-style state machine composed of N states such as Fig. 9.91 may have multiple outputs attached to each state, making its implementation nearly impossible with conventional state machine implementation techniques. However, these types of designs are excellent candidates for the counter-decoder type of designs where each state in the state diagram is associated with a counter output value. Therefore, as the counter increments, the present state outputs of each state in Fig. 9.91 can simply be generated using a set of decoders connected at the counter output.

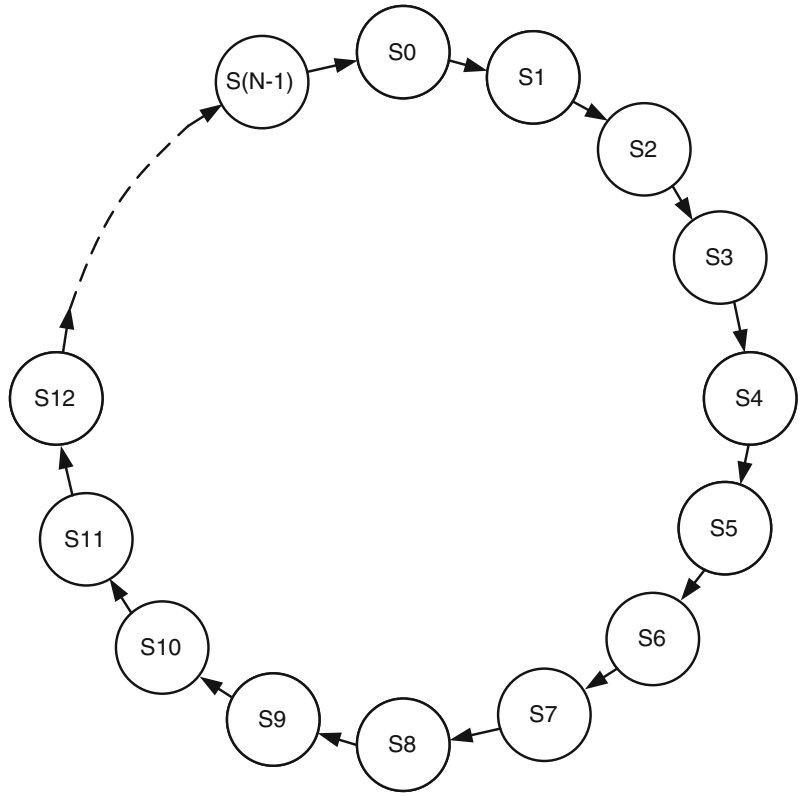


Fig. 9.91 State diagram of a counter with N states

To illustrate this theory, a controller that generates the timing diagram in Fig. 9.92 will be implemented using both the Moore-type state machine and the counter-decoder approach.

As shown in the timing diagram below, this state machine generates a single active-high output, Out = 1, once in every eight cycles as long as Stop = 0. When Stop = 1, the machine stalls and retains its current state.

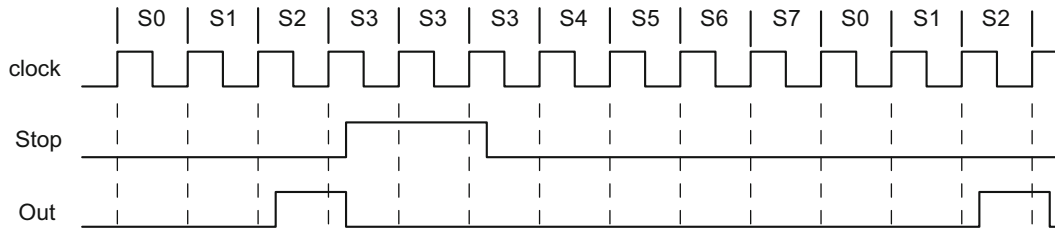


Fig. 9.92 Timing diagram of a state machine with a single input, Stop, and a single output

Once the state assignments are made for each clock cycle of Fig. 9.92, the state diagram in Fig. 9.93 emerges for a Moore-type state machine.

The states S0 and S1 in the timing diagram are assigned to the first and second clock cycles, respectively. The third clock cycle is assigned to state S2 where Out = 1. The fourth

clock cycle corresponds to state S3. The machine stays in state S3 as long as Stop = 1. This ranges from the fourth to the sixth clock cycle in the timing diagram. The state assignments from the seventh to the tenth clock cycles become states S4, S5, S6 and S7. The eleventh clock cycle returns to state S0.

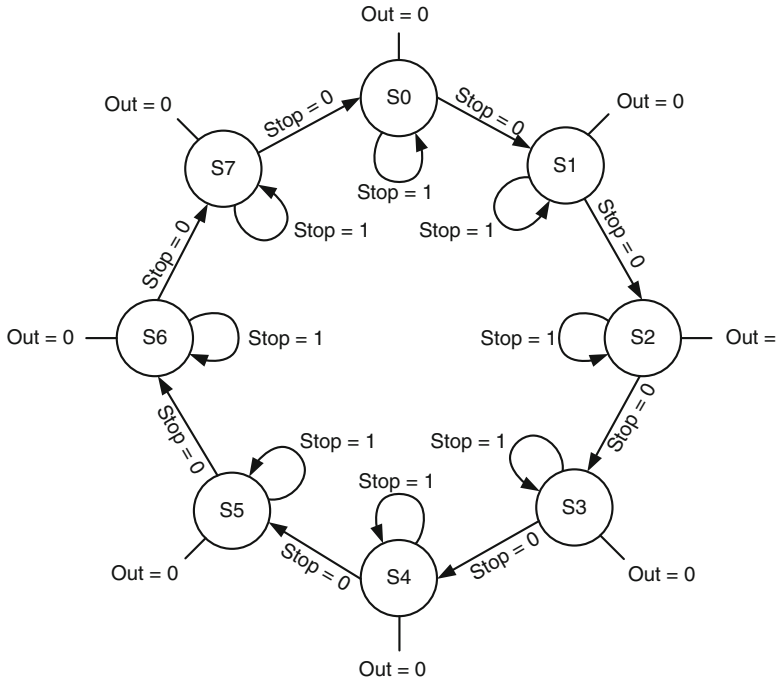


Fig. 9.93 Moore-type state machine representation of the timing diagram in Fig. 9.92

Implementing the state diagram in Fig. 9.93 follows a lengthy process of producing the state tables, transition tables, and K-maps that results in a total of four outputs (three flip-flop outputs due to eight states and one output for Out). However, using a counter-decoder approach minimizes this design task considerably and reveals a rather explicit circuit implementation.

When the timing diagram in Fig. 9.92 is redrawn to implement the counter-decoder design approach, it yields a simple three-bit counter which counts from zero to seven as shown in Fig. 9.94. The counter output, CountOut, is included in this figure to show the relationships between the state assignments, the input node, Stop, and the output node, Out. The figure also shows the clock cycle where the counter resets itself when its output reaches seven.

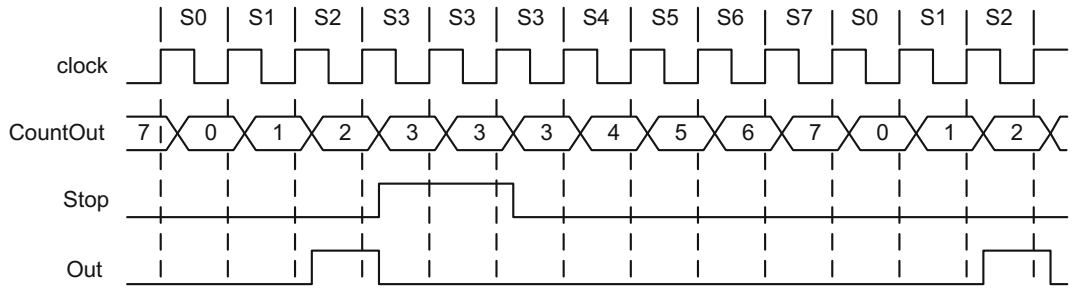


Fig. 9.94 Timing diagram of a three-bit counter with a single input (Stop) and a single output

The first task for the design is to construct a three-bit up-counter as shown in Fig. 9.95. The counter in this figure is derived from a general counter topology, and it consists of a three-bit adder, three 2-1 MUXes and three flip-flops. A three-input AND gate is used as a decoder at the counter output to implement Out = 1 when the CountOut node reaches 2. Therefore, this method follows a simple, step-by-step design approach in producing the final circuit that does not require implicit logic design techniques.

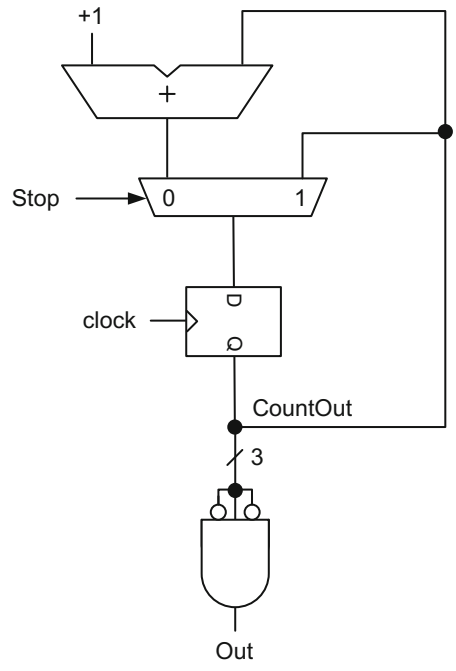


Fig. 9.95 Counter-decoder representation of the timing diagram in Fig. 9.94

9.21 A Simple Memory Block

Small memory blocks can be assembled from one-bit registers shown in Fig. 9.73 to use in a variety of designs. For example, a 32-bit wide, 16-bit deep memory block shown in Fig. 9.96 can be built by stacking 16 rows of 32-bit registers on top of each other. The 32-bit register in each row has tri-state output buffers to be used during read as shown in Fig. 9.97.

All inputs to each column of Fig. 9.96 are connected together to write data. For example, the input terminal, IN[0], in Fig. 9.96 is connected to all the input pins, In[0], of rows 0 to 15 in Fig. 9.97 to write a single bit of data at a selected row. The same is true for the remaining inputs, IN[1] through IN[31].

Similarly, all outputs of each column in Fig. 9.96 are connected together to read data from the memory block. For example, the output pin, OUT[0], is connected to all output pins, Out[0], from each row to read one bit of data from a selected row. The same is true for the remaining output pins, OUT[1] through OUT[31].

Every row of the memory block in Fig. 9.96 is accessed by individual Write Enable (WE) and Read Enable (RE) signals for writing or reading data, respectively.

In order to generate the WE inputs, WE[0] to WE[15], an address decoder is used. This decoder enables only one row while deactivating all the other rows using a four-bit address, Address[3:0], and a single WE input according to the truth table in Fig. 9.98. For example, a 32-bit data is written to row 0 if WE = 1 and Address[3:0] = 0000 at the decoder input. However, WE = 0 blocks writing data to all rows of the memory block regardless of the input address as shown in the truth table in Fig. 9.99.

The RE inputs, RE[0] through RE[15], use address decoders similar to Fig. 9.98 and Fig. 9.99 to read a block of data from a selected row. The read operation is achieved with a valid input address and RE = 1 according to the truth table in Fig. 9.100. An RE = 0 entry disables reading data from any row regardless of the value of the input address as shown in Fig. 9.101.

Therefore, one must provide a valid input address and the control signals, RE and WE, to perform a read or a write operation, respectively. The WE = 0 and RE = 1 combination reads data from the selected row. Similarly, the WE = 1 and RE = 0 combination writes data to a selected row. The WE = 0 and RE = 0 combination disables both reading and writing to the memory block. The control input entry, WE = 1 and RE = 1, is not allowed, and it should be interpreted as memory read.

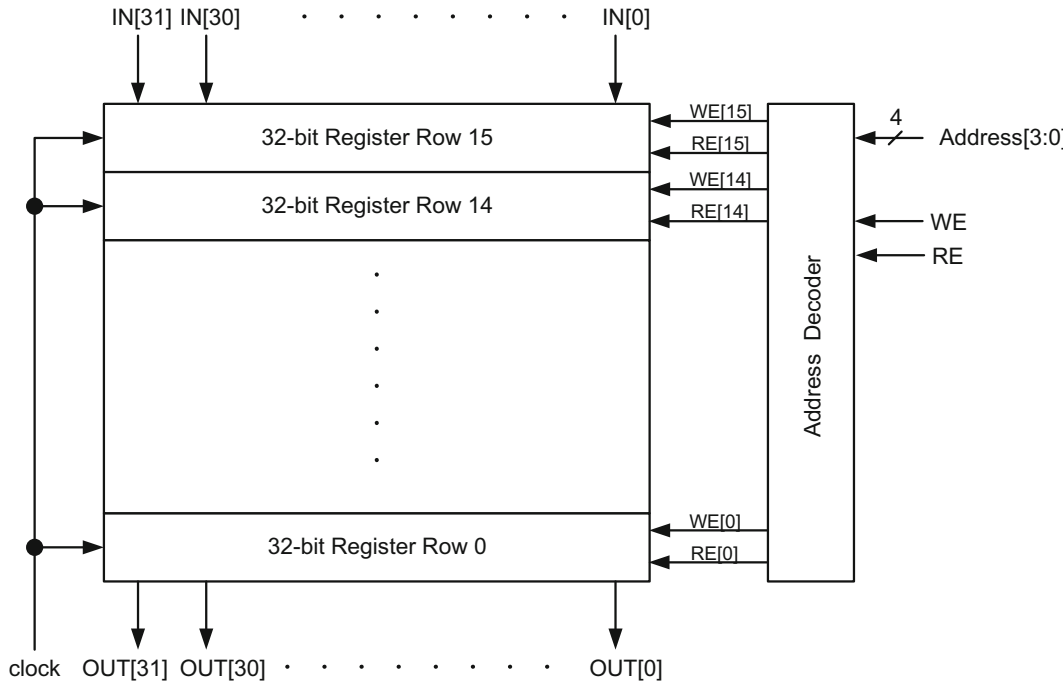


Fig. 9.96 A 32 × 16 memory and the truth table of its address decoder

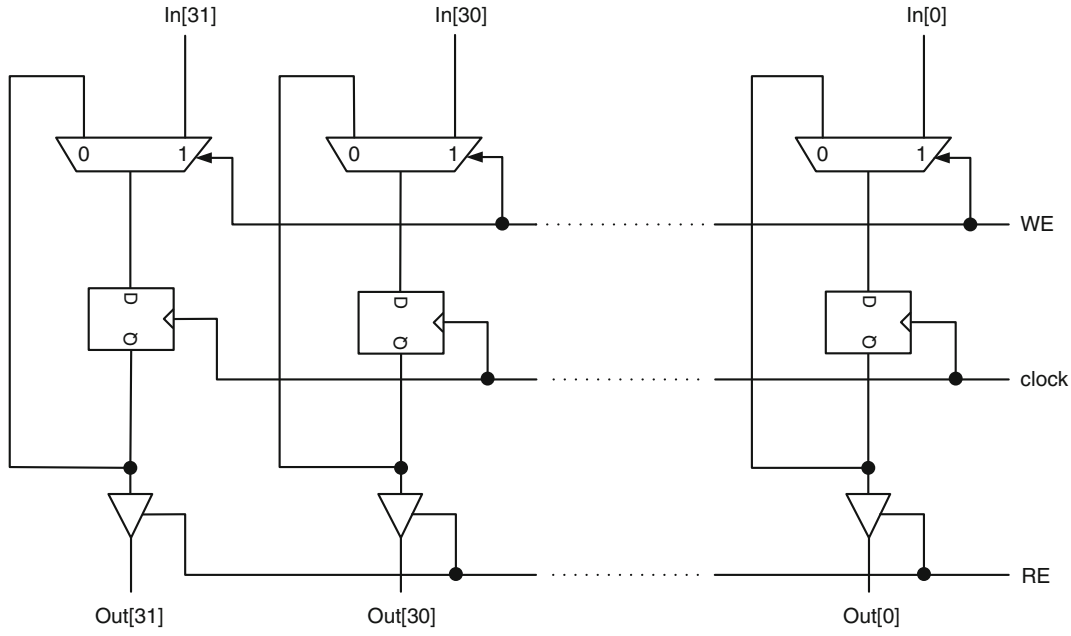


Fig. 9.97 A 32-bit register slice at every row of Fig. 9.96

Address	WE[15]	WE[14]	WE[13]	WE[2]	WE[1]	WE[0]
0 0 0 0	0	0	0 · · · · · · · · · · · ·	0	0	1
0 0 0 1	0	0	0 · · · · · · · · · · · ·	0	1	0
0 0 1 0	0	0	0 · · · · · · · · · · · ·	1	0	0
⋮	⋮	⋮			⋮	
1 1 1 0	0	1	0 · · · · · · · · · · · ·	0	0	0
1 1 1 1	1	0	0 · · · · · · · · · · · ·	0	0	0

Fig. 9.98 The address decoder for the 32 × 16 memory in Fig. 9.96 when WE = 1

Address	WE[15]	WE[14]	WE[13]	WE[2]	WE[1]	WE[0]
0 0 0 0	0	0	0 · · · · · · · · · · · ·	0	0	0
0 0 0 1	0	0	0 · · · · · · · · · · · ·	0	0	0
0 0 1 0	0	0	0 · · · · · · · · · · · ·	0	0	0
⋮	⋮	⋮			⋮	
1 1 1 0	0	0	0 · · · · · · · · · · · ·	0	0	0
1 1 1 1	0	0	0 · · · · · · · · · · · ·	0	0	0

Fig. 9.99 The address decoder for the 32 × 16 memory in Fig. 9.96 when WE = 0

Address	RE[15]	RE[14]	RE[13]	RE[2]	RE[1]	RE[0]
0 0 0 0	0	0	0 · · · · · · · · · · · ·	0	0	1
0 0 0 1	0	0	0 · · · · · · · · · · · ·	0	1	0
0 0 1 0	0	0	0 · · · · · · · · · · · ·	1	0	0
⋮	⋮	⋮			⋮	
1 1 1 0	0	1	0 · · · · · · · · · · · ·	0	0	0
1 1 1 1	1	0	0 · · · · · · · · · · · ·	0	0	0

Fig. 9.100 The address decoder for the 32 × 16 memory in Fig. 9.96 when RE = 1

Address	RE[15]	RE[14]	RE[13]	RE[2]	RE[1]	RE[0]
0 0 0 0	0	0	0 · · · · · · · · · · · ·	0	0	0
0 0 0 1	0	0	0 · · · · · · · · · · · ·	0	0	0
0 0 1 0	0	0	0 · · · · · · · · · · · ·	0	0	0
⋮	⋮	⋮		⋮	⋮	
1 1 1 0	0	0	0 · · · · · · · · · · · ·	0	0	0
1 1 1 1	0	0	0 · · · · · · · · · · · ·	0	0	0

Fig. 9.101 The address decoder for the 32 × 16 memory in Fig. 9.96 when RE = 0

9.22 A Design Example

This design example combines the data-path and controller design concepts described earlier in this chapter. It also introduces the use of important sequential logic blocks, such as flip-flop, register, counter and memory, into the design.

Every design starts with gathering the small or large logic blocks to construct a data-path and setting up a proper data-flow according to the design specifications. Once the data-path is set, then the precise data movements from one logic block to the next is shown using a timing diagram. Any change in the data-path should be included in the timing diagram or vice versa.

When the data-path design and its timing diagram are complete and fully associate with each other, the next step in the design process is to build the controller circuit that governs the data-flow. To define the states of the controller, the clock periods that generate different sets of controller outputs are separated from each other and named as distinct states. Similarly, the clock periods with identical controller outputs are combined under the same state. The controller design can be Moore-type or Mealy-type according to the design needs.

The example design in this section reads two eight-bit data packets from an 8 × 8 source memory (memory A), processes them and stores the result in an 8 × 4 target memory (memory B). The processing part depends on the relative contents of each data packet: if the contents of the first data packet are larger than the second, the contents of the data packets are added. Otherwise, they are subtracted from each other before the result is stored.

The block diagram in Fig. 9.102 demonstrates the data-path required for this memory-to-memory data transfer as described above. The timing diagram in Fig. 9.103

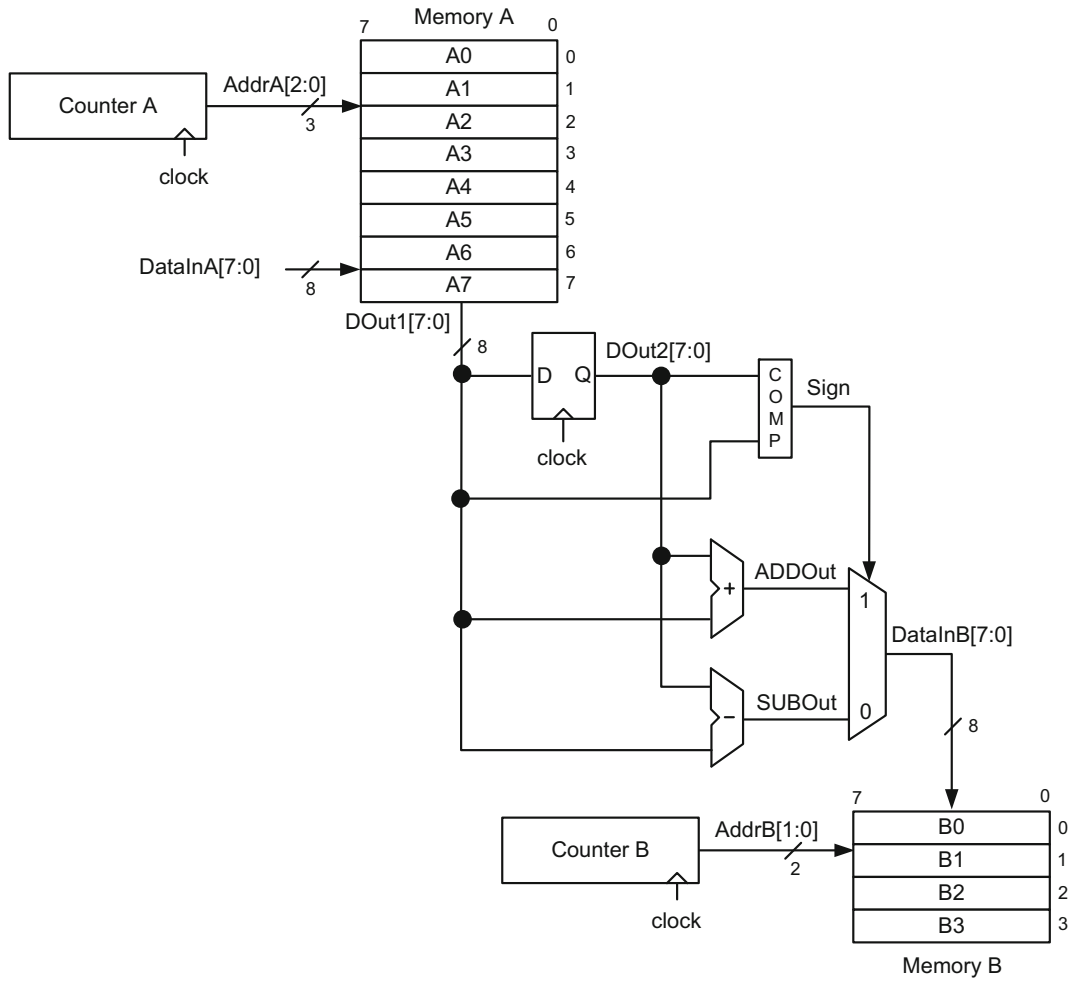


Fig. 9.102 Data-path of a memory-memory data transfer

needs to accompany the data-flow in Fig. 9.102 since it depicts precise data movements at each clock cycle.

Initially, counter A generates the addresses 0 to 7 for the memory A and writes the data packets A0 to A7 through its DataInA[7:0] port. This is shown in the timing diagram in Fig. 9.103 from clock cycles 1 through 8. When this task is complete, counter A resets and reads the first data packet A0 from AddrA[2:0] = 0 in clock cycle 9. In the next clock cycle, A0 becomes available at DOut1[7:0], and the counter A increments by one. In cycle 11, AddrA[2:0] becomes 2, the data packet A1 is read from DOut1[7:0], and the data packet A0 transfers to DOut2[7:0]. In this cycle, the contents of the data packets A0 and A1 are compared with each other by subtracting A1 (at DOut1) from A0 (at DOut2). If the contents of A0 are less than A1, then the sign bit, Sign, of (A0 − A1) becomes negative. Sign = 1 selects (A0 + A1) at ADDOut[7:0] and routes this value to DataInB[7:0]. However, if the

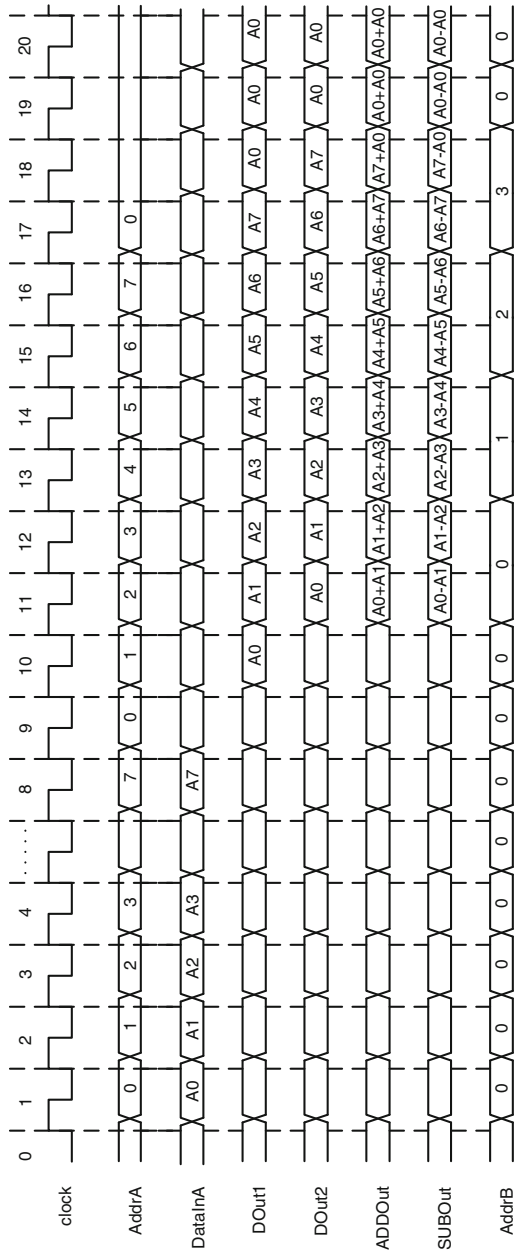


Fig. 9.103 Timing diagram of the memory-memory data transfer in Fig. 9.102

contents of A0 are greater than A1, (A0 − A1) becomes positive. Sign = 0 selects (A0 − A1) and routes this value from SUBOut[7:0] to DataInB[7:0]. The result at DataInB[7:0] is written at AddrB[1:0] = 0 of memory B at the positive edge of clock cycle 12. In the same cycle, A1 is transferred to DOut2[7:0], and A2 becomes available at DOut1[7:0]. A comparison between A1 and A2 takes place, and either (A1 + A2) or (A1 – A2) is written to memory B depending on the value of the Sign node. However, this is an unwarranted step in the data transfer process because the design requirement states that the comparison has to be done only once between data packets from memory A. Since A1 is used in an earlier comparison with A0, A1 cannot be used in a subsequent comparison with A2, and neither (A1 + A2) nor (A1 - A2) should be written to memory B. The remaining clock cycles from 13 through 18 compare the values of A2 with A3, A4 with A5, and A6 with A7, and write the added or subtracted results to memory B. After clock cycle 19, all operations on this data-path suspend, the counters are reset, and all writes to the memory core are disabled.

To govern the data-flow in Fig. 9.103, a Moore-type state machine (or a counter-decoder-type controller) is used. A Mealy-type state machine for a controller design is usually avoided because the present state inputs to this type of state machine may change during the clock period and result in jittery outputs.

The inclusion of the controller identifies the control signals that govern the data-flow in Fig. 9.104. These signals increment the counters A and B, and enable writes to memory A or B when necessary. The timing diagram in Fig. 9.103 is also expanded to include the control signals, IncA, IncB, WEA and WEB, as shown in Fig. 9.105. Therefore, it provides a complete picture of the data transfer process from memory A to memory B in contrast to the earlier timing diagram in Fig. 9.103.

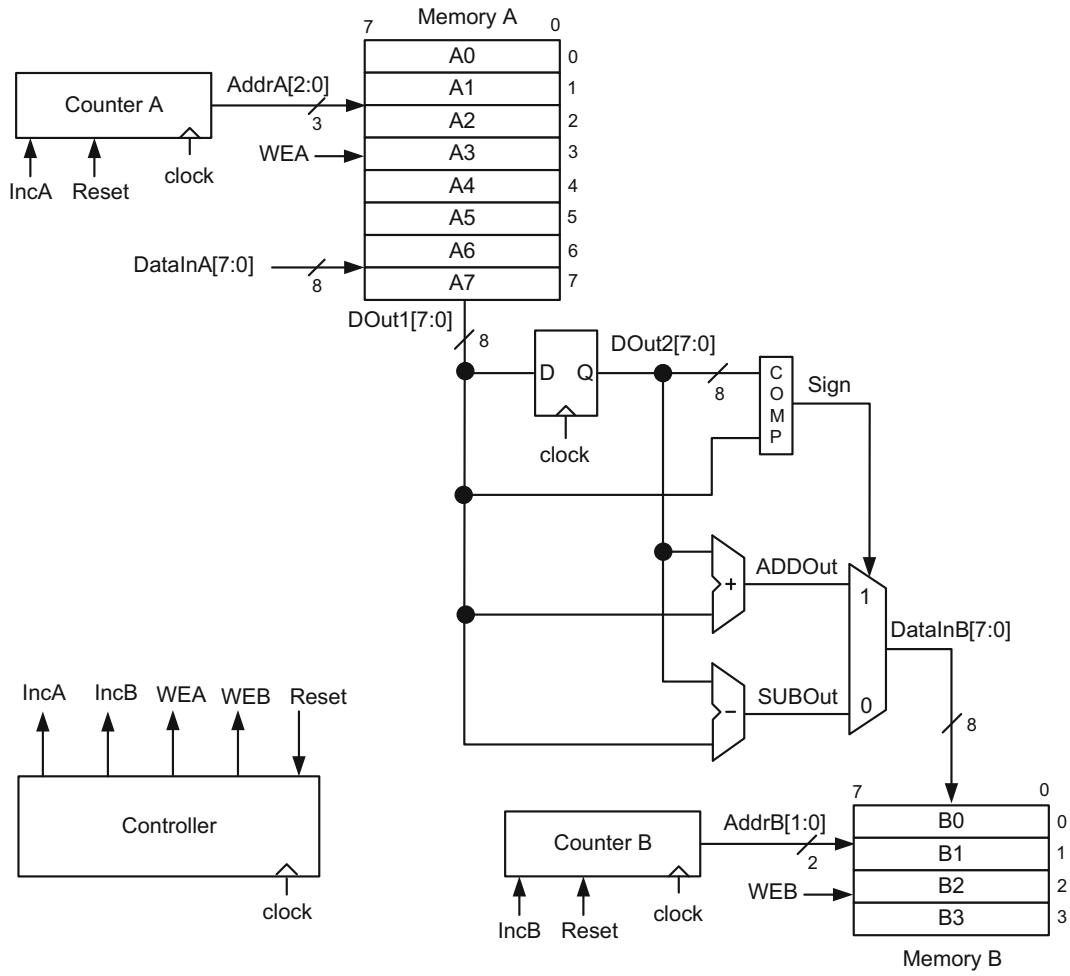


Fig. 9.104 Compete block diagram of the memory-memory data transfer with the controller

The controller in Fig. 9.104 is implemented by a Moore-type state machine in Fig. 9.106 and counter-decoder-type design in Fig. 9.107.

In the Moore type design, the states from S1 through S18 are assigned to each clock cycle of the timing diagram in Fig. 9.105. Subsequently, the values of the present state outputs, WEA, IncA, WEB and IncB, at each clock cycle are read off from the timing diagram and attached to each state in Fig. 9.106. The reset state, S0, is included in the Moore machine in case the data-path receives an external reset signal to interrupt an ongoing data transfer process. Whichever state the machine may be, Reset = 1 always transitions the state of the machine to S0 state. These transitions are not included in Fig. 9.106 for simplicity.

The counter-decoder style design in Fig. 9.107 consists of a five-bit counter and four decoders to generate WEA, IncA, WEB and IncB control signals. To show the operation of

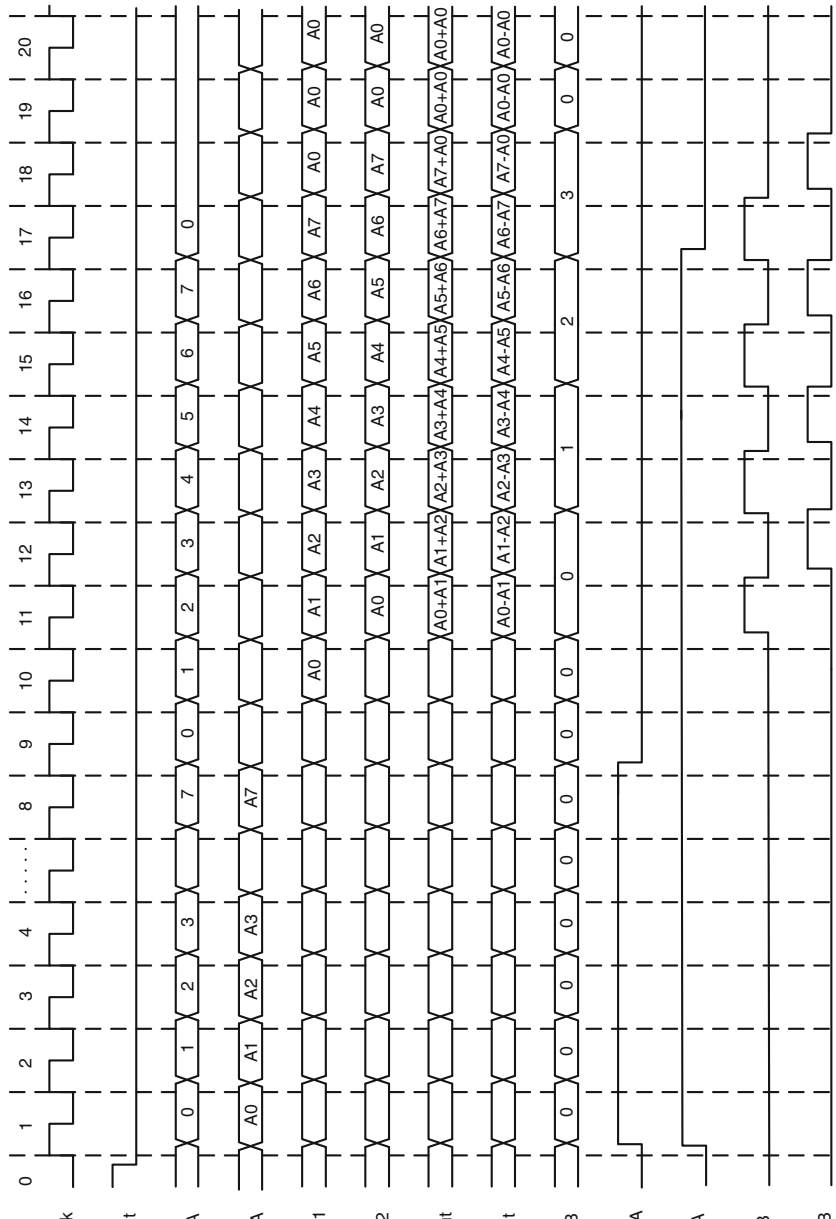


Fig. 9.105 Complete timing diagram of the memory-memory data transfer in Fig. 9.104

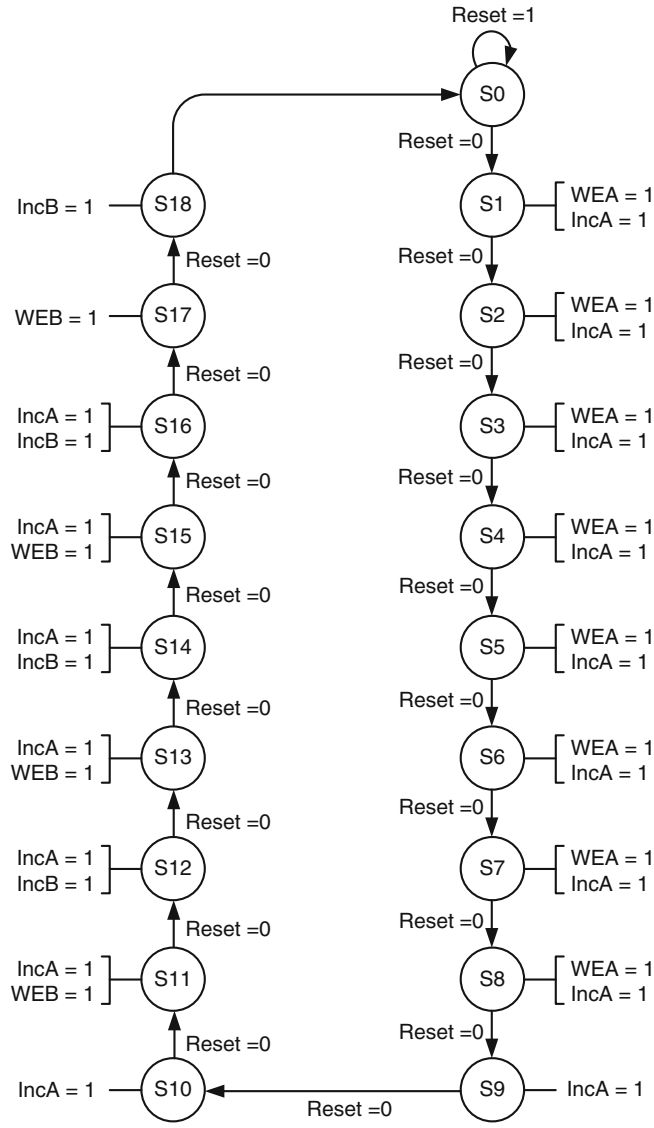


Fig. 9.106 Moore-type state diagram of the controller unit in Fig. 9.104

this design to generate WEA, for example, this particular decoder includes eight five-input AND gates, one for each clock cycle from cycle number one to cycle number eight in order to keep WEA = 1 in Fig. 9.105. The five-bit counter implicitly receives a reset signal from its output when it reaches clock cycle 18 and resets counter A, counter B and the rest of the system in Fig. 9.104.

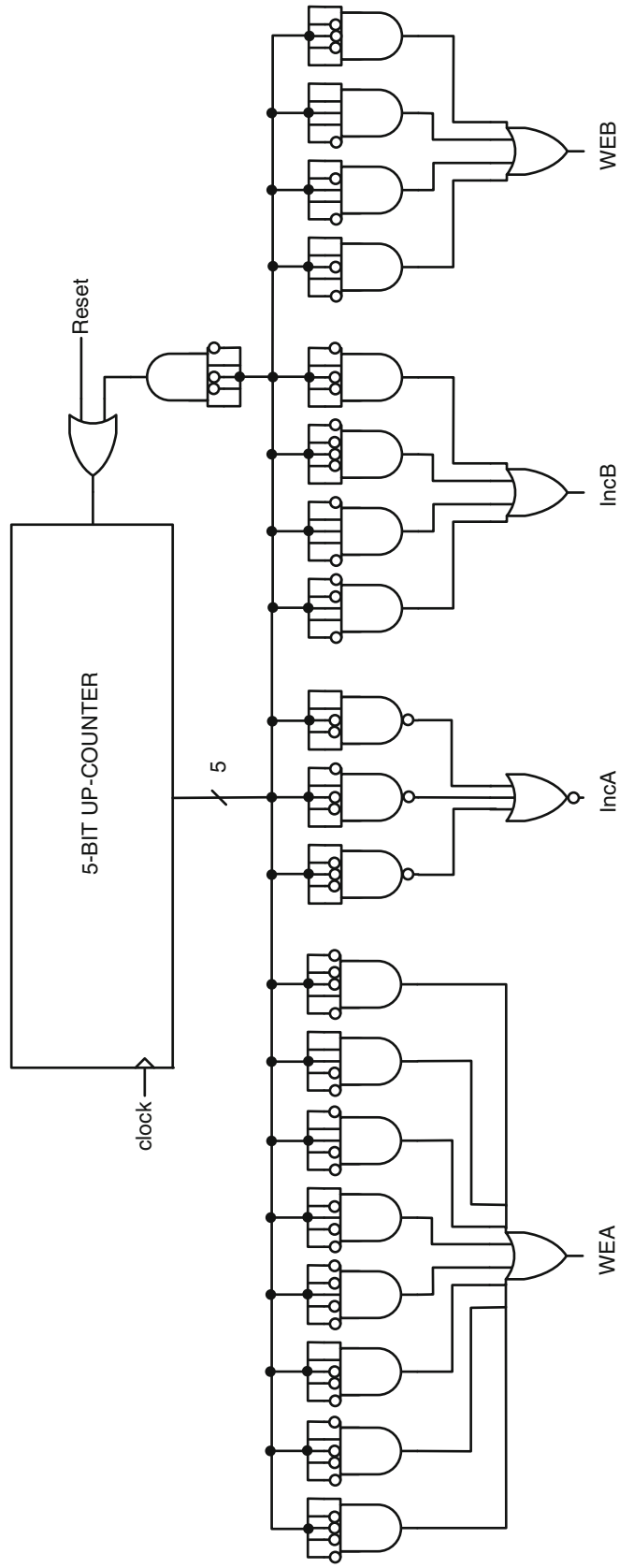


Fig. 9.107 Counter-decoder schematic of the controller unit in Fig. 9.104

Review Questions

1. Implement the following gates:

(a) Implement a two-input XOR gate using two-input NAND gates and inverters.
(b) Implement a two-input AND gate using two-input XNOR gates and inverters.

2. Simplify the equation below:

$$\text{out} = (\overline{A + B}) \,.\, (\overline{\overline{A} + \overline{B}})$$

3. Simplify the equation below:

$$\text{out} = (\overline{A} + C) \,.\, (\overline{A} + \overline{C}) \,.\, (\overline{A + B + \overline{C}})$$

4. Obtain the SOP and POS expressions for the following function:

$$\text{out} = (A \,.\, B + C) \,.\, (B + A \,.\, C)$$

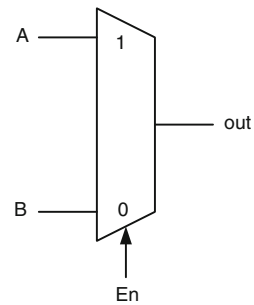
5. Implement the following function using NAND gates and inverters:

$$\text{out} = A \,.\, \overline{C} + B \,.\, C + \overline{A} \,.\, \overline{B} \,.\, D$$

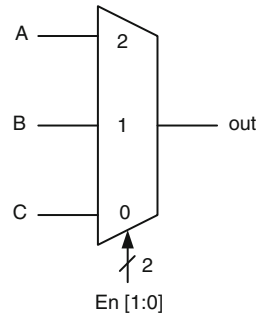
6. Implement the following function using NOR gates and inverters:

$$\text{out} = (A \oplus B) \,.\, (\overline{C \oplus D})$$

7. Implement the following 2-1 multiplexer using AND and OR gates:
Note that the function of this complex gate must produce the following:
If En = 1 then out = A else (when En = 0) out = B.



8. Implement the following 3-1 multiplexer using AND and OR gates:
Note that the function of this complex gate must produce the following:
If En = 2 then out = A; if En = 1 then out = B else (when En = 0 or En = 3) out = C.



9. Implement a two-bit ripple-carry adder with inputs A[1:0] and B[1:0] and an output C[1:0] using one-bit half- and full-adders. Preserve the overflow bit at the output as C[2].

10. Implement a two-bit ripple-carry subtractor with inputs A[1:0] and B[1:0] and an output C[1:0] using one-bit half and full-adders. Preserve the overflow bit at the output as C[2].

11. Implement a two-bit multiplier with inputs A[1:0] and B[1:0] and an output C[3:0] using one-bit half and full-adders.

12. Construct a four-bit comparator with inputs A[3:0] and B[3:0] using a subtractor. The comparator circuit should identify the following cases as active-high outputs:
A[3:0] = B[3:0]
A[3:0] > B[3:0]
A[3:0] < B[3:0]

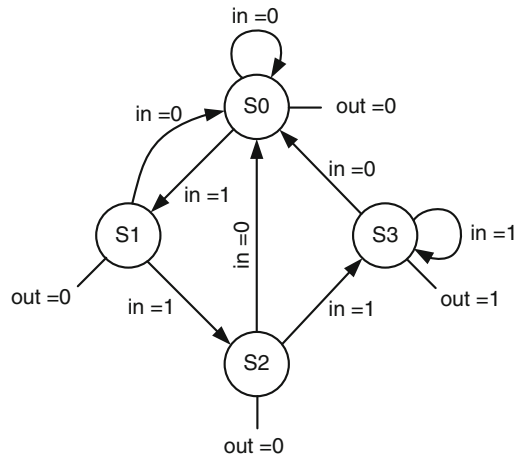
13. Implement a two-bit decoder that produces four outputs.
When enabled the decoder generates the following outputs:

in[1:0] = 0 then out[3:0] = 1
in[1:0] = 1 then out[3:0] = 2
in[1:0] = 2 then out[3:0] = 4
in[1:0] = 3 then out[3:0] = 8

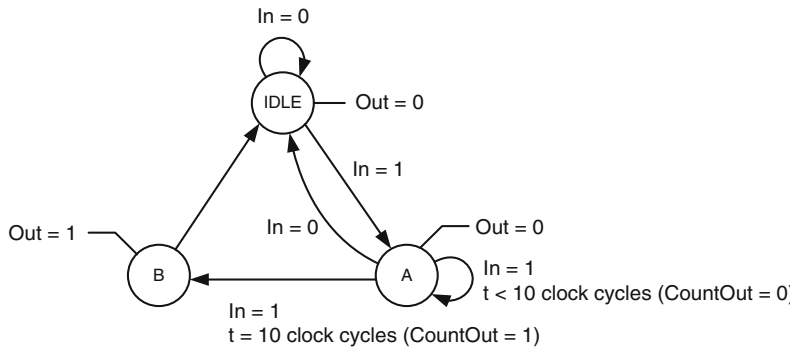
When disabled the out [3:0] always equals to zero regardless of the input value.

14. Design a 64-bit adder in ripple-carry form and compare it against carry-look-ahead, carry-select, and carry-look-ahead/carry-select hybrids in terms of speed and the number of gates which define the circuit area. Divide the 64-bit carry-look-ahead/carry-select hybrid into four-bit, eight-bit, 16-bit and 32-bit carry-look-ahead segments. Indicate which carry-look-ahead/carry-select hybrid produces the optimum design.

15. Implement the following Moore machine:



16. Implement the following Moore machine using a timer. The timer is initiated when In = 1. The state machine goes to the A-state and stays there for 10 cycles. In the tenth cycle, the state machine transitions to the B-state and stays in this state for only one cycle before switching to the IDLE-state. One implementation scheme is to construct a four-bit up-counter to generate the timer. When the counter output reaches 9, the decoder at the output of the counter informs the state machine to switch from the A-state to the B-state.



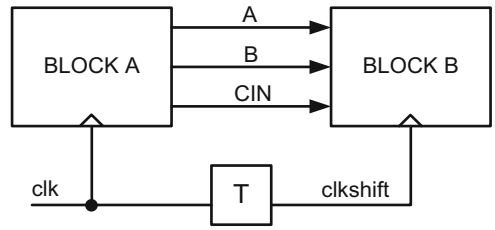
17. The following decoder needs to be implemented using only two-input NAND gates and inverters.

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	3
1	0	1	2
1	1	0	1
1	1	1	0

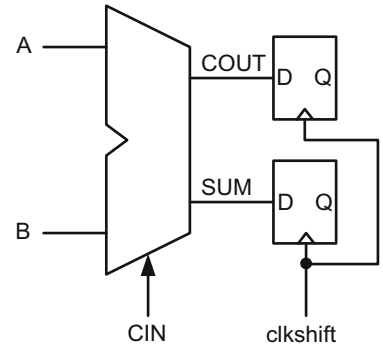
T_{NAND} = 500 ps (two-input NAND propagation delay)
T_{INV} = 100 ps (inverter propagation delay)
t_{CLKQ} = 200 ps (clock-q delay)
t_{SU} = 200 ps (set-up time)
t_H = 300 ps (hold time)

(a) Implement this decoder between two flip-flop boundaries.
(b) Find the maximum clock frequency using a timing diagram.
(c) Now the clock is shifted by 500 ps at the receiving flip-flop boundary. Show whether or not you have any hold violation using a timing diagram.

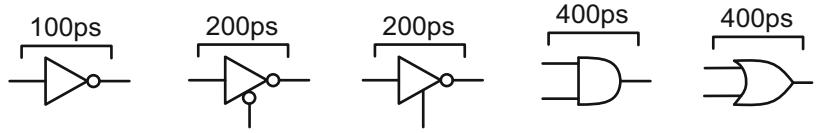
18. The block diagram is given below:



Block B contains one-bit adder with SUM = A ⊕ B ⊕ CIN and COUT = A . B + CIN . (A + B), and two flip-flops as shown below:



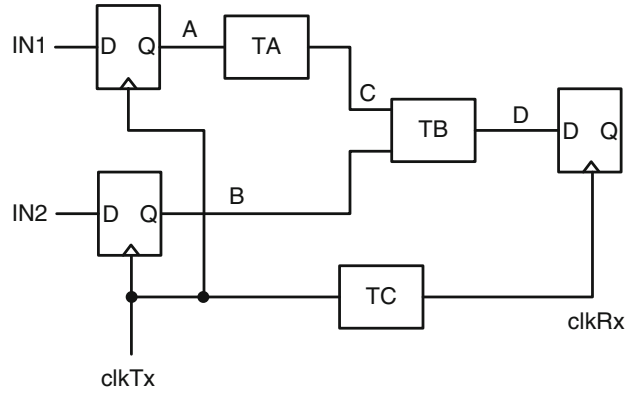
(a) Using the gates with propagation delays below, determine the setup time for the A, B, and CIN inputs with respect to clkshift.



(b) Assuming T = 0 ns and T_{CLK} (clock period) = 5 ns, if data at A, B and CIN become valid and stable 4 ns after the positive edge of clkshift, will there be any timing violations? Assume t_H (hold time) = 3 ns for the flip-flop.

(c) How can you eliminate the timing violations? Show your calculations and draw a timing diagram with no timing violations.

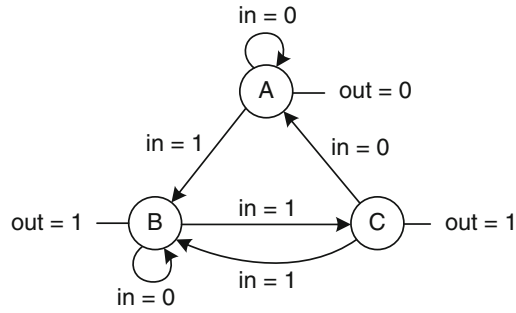
19. A schematic is given below:



(a) Assume t_{SU} (setup time) = 200 ps, t_H (hold time) = 200 ps, $t_{CLK\text{-}Q}$ (clock-to-q delay) = 300 ps for the flip-flop, and T_A = 1000 ps, T_B = 100 ps for the internal logic blocks on the schematic. Show if there is any timing violation or timing slack by drawing a detailed timing diagram when T_C = 0 ps.

(b) What happens if T_C = 400 ps? Show it in a separate timing diagram.

20. The state diagram of a Moore machine is given below:



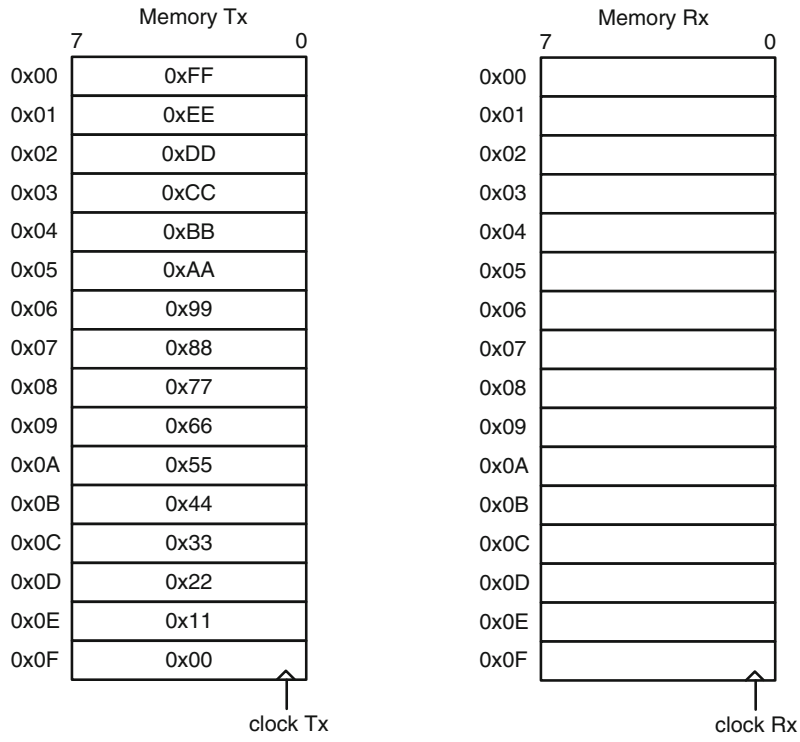
The assignment of the states A, B and C are indicated as follows:

states	PS[1]	PS[0]
A	0	0
B	0	1
C	1	1

(a) Implement this state machine using inverters, two-input and three-input AND gates and two-input OR gates.

(b) Find the maximum operating frequency of the implementation in part (a) if the following timing assignments are applied:
t_{SU} (setup time) = 100 ps, t_H (hold time) = 100 ps, $t_{CLK\text{-}Q}$ (clock-to-q delay) = 200 ps, T_{INV} = 200 ps, T_{AND2} = 300 ps, T_{AND3} = 400 ps, T_{OR2} = 400 ps.
Here, T_{INV}, T_{AND2}, T_{AND3}, T_{OR2} correspond to the inverter, two-input and three-input AND gates and two-input OR gate, respectively.

21. Data is transferred from Memory Tx to Memory Rx starting from the address 0×00 and ending at the address $0 \times 0F$ as shown below. Once a valid address is produced for Memory Tx, the data becomes available at the next positive clock edge. On the other hand, data is written to the Memory Rx at the positive edge of the clock when a valid address is produced. The operating clock frequency of Memory Tx is twice as high as that of Memory Rx.



(a) Assuming address generators for Memory Tx and Memory Rx start generating valid addresses at the same positive clock edge, show which data is actually stored in Memory Rx using a timing diagram. Indicate all the address and data values for Memory Tx and Memory Rx in the timing diagram.

(b) Now, assume that the operating clock frequency of Memory Tx is four times as high as the clock frequency of Memory Rx and the write takes place at the negative edge of the clock for Memory Rx when a valid address is present. Redraw the timing diagram indicating all address and data values transferred from one memory to the next.

22. Serial data is transferred to program four eight-bit registers. The start of the transfer is indicated by a seven-bit sequence = {1010101} immediately followed by the address of the register (two bits) and the data (eight bits). The transfer stops after programming the last register. After this point, all other incoming bits at the serial input is ignored. Design this interface by developing a data-path and a timing diagram simultaneously. Implement the state diagram. Can this controller be implemented by a counter-decoder scheme?