Logic Reference Guide

Advanced Micro Devices

INTRODUCTION

Throughout this data book and design guide we have assumed that you have a good working knowledge of logic. Unfortunately, there always comes a time when you are called on to remember something which can only be found in that logic textbook which you threw away years ago.

This section is intended to provide a quick review and reference of the basic principles of digital logic. We will cover three general areas:

- Basic logic elements
- Basic storage elements
- Binary numbers

Throughout the text, we will use the notation that was used throughout this book. If you are unfamiliar with the syntax, you will probably find it easy to understand as you read; if you wish for a more detailed explanation of the symbols, please refer to the Basic Design with PLDs section where they are defined.

As this is a logic reference only, we cannot take on lengthy discussions, nor can we train you in the basic principles of digital logic if you have not previously been trained. In such a case, we must refer you to your favorite logic textbook.

BASIC LOGIC ELEMENTS

In this section, we will discuss the concepts surrounding combinatorial logic functions.

The Three Basic Gates

There are three basic logic gates from which all other combinatorial logic functions can be generated. These functions are *NOT*, *AND*, and *OR*. A truth table indicating these functions is shown in Table 1. Since they can be used to generate any function, they are said to be *functionally complete.*

Table 1. Truth Table for the NOT, AND, and OR Functions

A	B	/A	A*B	A+B
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

The standard schematic symbols used to represent these gates are shown in Figure 1.

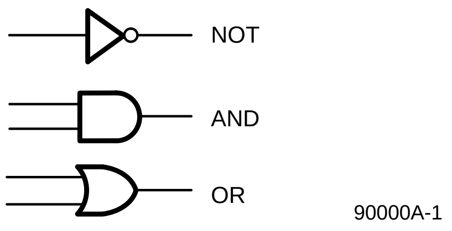


Figure 1. Schematics Symbols for the Three Fundamental Gates

The AND and NOT functions can be combined into the *NAND* function. This is equivalent to an AND gate followed by an inverter, as shown in Figure 2a. Likewise, the OR and NOT gates can be combined into the *NOR* function, as shown in Figure 2b. Each of these gates is functionally complete; any logic function can be expressed solely as a function of NAND or NOR gates.

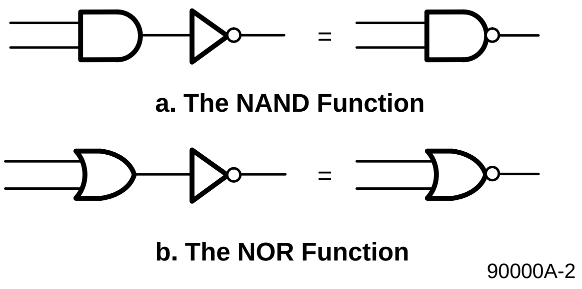


Figure 2. The NAND and NOR Functions

Precedence of Operators

Logic functions may be created with any combination of the three basic functions. How those functions are expressed affects the evaluation of the function. The normal order of evaluation is:

NOT, AND, OR

Evaluation proceeds in order from left to right.

Publication# 90000 Rev. A Amendment /0
Issue Date: June 1993

This order may be altered by inserting parentheses in the function. The contents of the parentheses will always be evaluated before the rest of the expression, from left to right.

Some example functions are evaluated in Table 2.

Table 2. Using Parentheses to Change the Order of Evaluation

A	B	C	D	A*B+/A* C+D	A*B+/A* (C+D)	A*(B+/A)* C+D	A*(B+/A)* (C+D)
0	0	0	0	0	0	0	0
0	1	1	0	1	1	0	0
1	0	0	1	1	0	1	0
1	1	1	1	1	1	1	1

Commutative, Associative, and Distributive Laws

The AND and OR functions are commutative and associative. This means that the operands can appear in any order without affecting the evaluation of the function. This is illustrated in Tables 3 and 4.

Table 3. Commutativity

A	B	A*B	B*A	A+B	B+A
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1

Table 4. Associativity

A	B	C	(A*B)*C	A*(B*C)	(A+B)+C	A+(B+C)
0	0	0	0	0	0	0
0	1	1	0	0	1	1
1	0	1	0	0	1	1
1	1	1	1	1	1	1

There are actually two distributive laws; one of them resembles standard algebra more than the other. These two laws state that:

$$A*(B+C) = (A*B) + (A*C)$$
$$A+(B*C) = (A+B) * (A+C)$$

Duality

The two distributive laws give an example of the concept of *duality*. This principle states that:

Any identity will also be true if the following substitutions are made:

* for +
+ for *
1 for 0
0 for 1

Thus, it is only necessary to prove the first of the distributive laws; the second one will then be true by duality. Note that duality is not required to prove the second law; it can also be proven by truth table or by logic manipulation.

Manipulating Logic

Logic functions may be manipulated by the use of Boolean algebra. The logic functions may be expressed in one of the two canonical forms, or by using a simplified expression.

Canonical Forms

There are two fundamental canonical forms: *sum-of-minterms* and *product-of-maxterms.* The former is by far the most widespread. These are special cases of what are more generally referred to as *sum-of-products* and *product-of-sums* forms. *Minterms* and *maxterms* are products and sums of the variables involved in a function. Each particular combination of noninverted and inverted variables in a product or sum is given a minterm or maxterm number, as shown in Table 5. Within each minterm or maxterm, the individual variables are referred to as *literals.*

For the case of sum-of-minterms form, the expression for a function may be found by ORing the minterms which correspond to the 1's in the function's truth table. Likewise, the product-of-maxterms expression may be found by ANDing the maxterms which correspond to the 0's in the truth table. This is illustrated in Figure 3.

Table 5. Minterms and Maxterms

Table of Minterms for Three Variables

Minterm	Name
/x*/y*/z	m0
/x*/y*z	m1
/x*y*/z	m2
/x*y*z	m3
x*/y*/z	m4
x*/y*z	m5
x*y*/z	m6
x*y*z	m7

Table of Maxterms for Three Variables

Maxterm	Name
x + y + z	M0
x + y +/z	M1
x +/y + z	M2
x +/y + /z	M3
/x + y + z	M4
/x + y + /z	M5
/x + /y + z	M6
/x + /y + /z	M7

Conversion Between Canonical Forms

It is a simple matter to convert between canonical forms. Given a truth table for a function F, there are four different representations that can be used:

- Sum-of-minterms form of F
- Product-of-maxterms form of F
- Sum-of-minterms form of /F
- Product-of-maxterms form of /F

One can convert back and forth between these representations by using the rules shown in Table 6.

A	B	C	D	X	Y	Minterm/ Maxterm Number
0	0	0	0	1	1	0
0	0	0	1	0	1	1
0	0	1	0	1	1	2
0	0	1	1	1	1	3
0	1	0	0	0	1	4
0	1	0	1	1	0	5
0	1	1	0	0	0	6
0	1	1	1	1	1	7
1	0	0	0	1	1	8
1	0	0	1	1	1	9
1	0	1	0	0	0	10
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
1	1	1	1	0	0	15

a. Truth Table

```
X  = m0+m2+m3+m5+m7+m8+m9
   = ∑m (0,2,3,5,7,8,9)

   =  /A   * /B   *  /C  *  /D              ;m0
   +  /A   * /B   *   C  *  /D              ;m2
   +  /A   */B*  C    * D;m3
   +  /A   *  B   *  /C  *   D              ;m5
   +  /A   *  B   *   C  *   D              ;m7
   +   A   * /B   *  /C  *  /D              ;m8
   +   A   */B*  /C   * D;m9

Y  = m0+m1+m2+m3+m4+m7+m8+m9
   = ∑m (0,1,2,3,4,7,8,9)

   =  /A   * /B   *  /C  *  /D              ;m0
   +  /A   */B*  /C   * D;m1
   +  /A   */B*  C    * /D;m2
   +  /A   */B*  C    * D;m3
   +  /A   *  B   *  /C  *  /D              ;m4
   +  /A   *  B   *   C  *   D              ;m7
   +   A   */B*  /C   * /D;m8
   +   A   */B*  /C   * D;m9
```

b. The Sum-of-Minterms Expression

```
X  = M1*M4*M6*M10*M11*M12*M13*M14*M15
     ΠM (1,4,6,10,11,12,13,14,15)

   =(A+B+C+/D)                      ;M1
   *(A+/B+C+D)                      ;M4
   *(A+/B+/C+D)                     ;M6
   *(/A+B+/C+D)                     ;M10
   *(/A+B+/C+/D)       ;M11
   *(/A+/B+C+D)                     ;M12
   *(/A+/B+C+/D)       ;M13
   *(/A+/B+/C+D)       ;M14
   *(/A+/B+/C+/D)   ;M15

Y  = M5*M6*M10*M11*M12*M13*M14*M15
   = ΠM (5,6,10,11,12,13,14,15)

   =(A+/B+C+/D)                     ;M5
   *(A+/B+/C+D)                     ;M6
   *(/A+B+/C+D)                     ;M10
   *(/A+B+/C+/D)       ;M11
   *(/A+/B+C+D)                     ;M12
   *(/A+/B+C+/D)       ;M13
   *(/A+/B+/C+D)       ;M14
   *(/A+/B+/C+/D)   ;M15
```

c. The Product-of-Maxterms Expression

Figure 3. Finding the Canonical Form from the Truth Table

Table 6. Conversion of Forms Table

Given Form	Desired Form			
	Minterm Expansion of F	**Maxterm Expansion of F**	**Inverted Minterm Expansion of F**	**Inverted Maxterm Expansion of F**
Minterm expansion of F	–	Maxterm numbers are those numbers not in the Minterm list of F	List Minterms not present in F	Maxterm numbers are the same as Minterm numbers of F
Maxterm expansion of F	Minterm numbers are those numbers not on the Maxterm list of F	–	Minterm numbers are the same as Maxterm numbers of F	List Maxterms not present in F

Simplifying Logic

Canonical forms are convenient in that it is easy to derive and convert them. However, the representation is bulky, since all variables must appear in each sum or product. These expressions can be simplified by applying the basic laws and theorems of Boolean algebra.

There are four basic postulates, two of which are the commutative and distributive laws which were discussed above. From these postulates, it is possible to derive nine basic theorems. The postulates and theorems are listed in Table 7.

Table 7. Postulates and Theorems of Boolean Algebra

Postulate 1	`(A)   X  +  FALSE  = X`
	`(B)   X*TRUE = X`

Postulate 2	`(A)   X  +  /X  =  TRUE`
	`(B)   X * /X  = FALSE`

Postulate 3	`(A)   X + Y  =  Y + X`
	`(B)   X*Y = Y*X`

Postulate 4	`(A)   X * (Y + Z) = (X*Y) + (X*Z)`
	`(B)   X + (Y*Z) = (X + Y) * (X + X)`

Theorem 1	`(A)   X + X = X`
	`(B)   X * X = X`

Theorem 2	`(A)   X + TRUE = FALSE`
	`(B)   X*FALSE = FALSE`

Theorem 3	`      / (/X)  = X`

Theorem 4	`(A)   X + (Y + Z) = (X + Y) + Z`
	`(B)   X * (Y*Z) =  (X*Y) * Z`

Theorem 5	`(A)   / (X + Y) = /X * /Y`
	`(B)   / (X * Y) = /X + /Y`

Theorem 6	`(A)   X + (X * Y) = X`
	`(B)   X * (X + Y) = X`

Theorem 7	`(A)   (X*Y) + (X*/Y) = X`
	`(B)   (X + Y) * (X + /Y) = X`

Theorem 8	`(A)   X + (/X*Y) = X + Y`
	`(B)   X * (/X + Y) X*Y`

Theorem 9	`(A)   (X*Y) + (/X*Z) + (Y*Z) = (X*Y) + (/X*Z)`
	`(B)   (X + Y) * (/X + Z) * (Y + Z) = (X + Y)*(/X + Z)`

Notice that each theorem and postulate (with the exception of theorem 3) has two forms. This is a result of the duality principle; once one form of a theorem is established, the dual representation follows immediately. Theorem 3 has no dual because it does not involve any of the elements that have duals (+, *, 1, or 0).

As the logic expression is simplified, it no longer contains minterms (or maxterms), since some of the minterms and literals are being eliminated. What was a sum-of-minterms (product of maxterms) representation is now simplified to a sum-of-products (product of sums).

DeMorgan’s Theorem

Once an expression has been simplified, it is no longer possible to invert the function by using Table 6. Inverting simplified logic requires DeMorgan’s theorem:

/(X*Y) = /X + /Y
/(X + Y) = /X*/Y

This is theorem 5 in Table 7.

There is one shortcut which can be used. The effect of inversion can be accomplished by inverting all literals and then using the dual representation. For example, given the expression

/(A*/B + A*C + /A*B*D)

we can invert to obtain:

/A*B + /A*/C + A*/B*/D	;step one, invert literals
(/A + B)*(/A + /C)* (A + /B + /D)	;step two, take dual

This expression must still be simplified to obtain a sum-of-products representation, but this shortcut eliminates some of the early steps.

Karnaugh Maps: Minimizing Logic

Simplifying by hand by using algebraic manipulation can be a tedious and error-prone procedure. When only a few variables are used (generally less than 5 or 6), Karnaugh maps (also called K-maps) provide a simpler graphical means of simplifying logic. K-maps not only allow for logic simplification, but for logic minimization, where an expression has a minimal number of product terms (or sum terms) and literals.

A Karnaugh map consists of a box which has one cell for each minterm. These cells are arranged so that only one literal is inverted when moving from one cell to an adjacent cell. The headings placed by each row and column indicate the polarities of the literals for that row or column. The literals themselves are indicated in the top left corner of the map. An example of a Karnaugh map for three variables is shown in Figure 4.

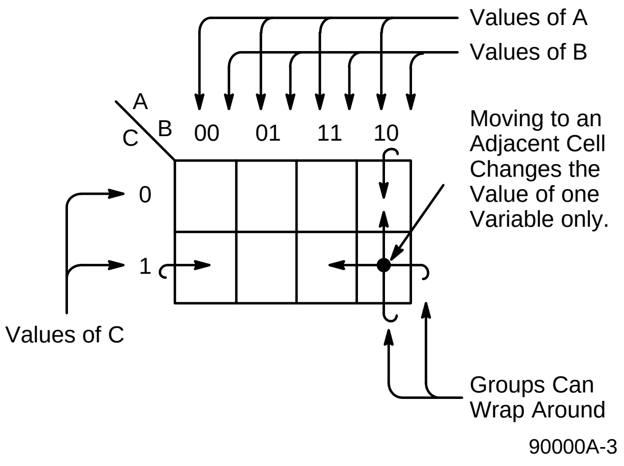


Figure 4. A Karnaugh Map for Three Variables

The truth table for a function is then transferred to the K-map by placing the 1’s and 0’s in the appropriate cells.

Since each cell differs from its neighbor only in the polarity of one of the literals, 1’s in adjacent cells can be combined by theorem 7a, which says that

x*y + x*/y = x

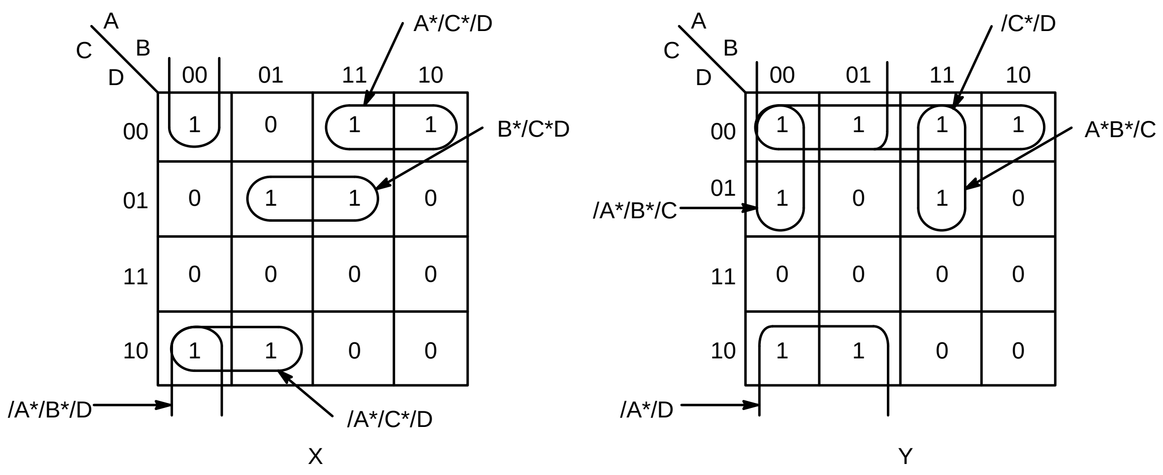
In this manner, two product terms are combined into one. This procedure can conceptually be repeated to allow groupings of two, four, eight, or any group of adjacent cells whose size is a power of two. A cell may appear in more than one group. Just enough groups are found to include all of the 1’s. The groups should be as large as possible.

This process provides a minimal sum of products. The product-of-sums form can be obtained by grouping 0’s instead of 1’s and inverting the header for each cell.

The two functions from Figure 3 have been placed into K-maps in Figure 5. The groups are then used as individual product terms. When reading the product terms from the map, the only literals which will appear in the product term are the ones whose values are constant for each cell in the group. If that value is 1, then the non-inverted form of the literal is used. If the value is 0, then the inverted form of the literal is used.

For active-LOW functions, the same procedure is used, except that the 0’s are grouped instead of the 1’s. The active-LOW version of the functions from Figure 3 are derived in Figure 6.

Hand simplification and minimization is not needed as frequently today as in the past, since software is now available for handling these logic manipulations. Most software can perform logic simplification and minimization automatically.

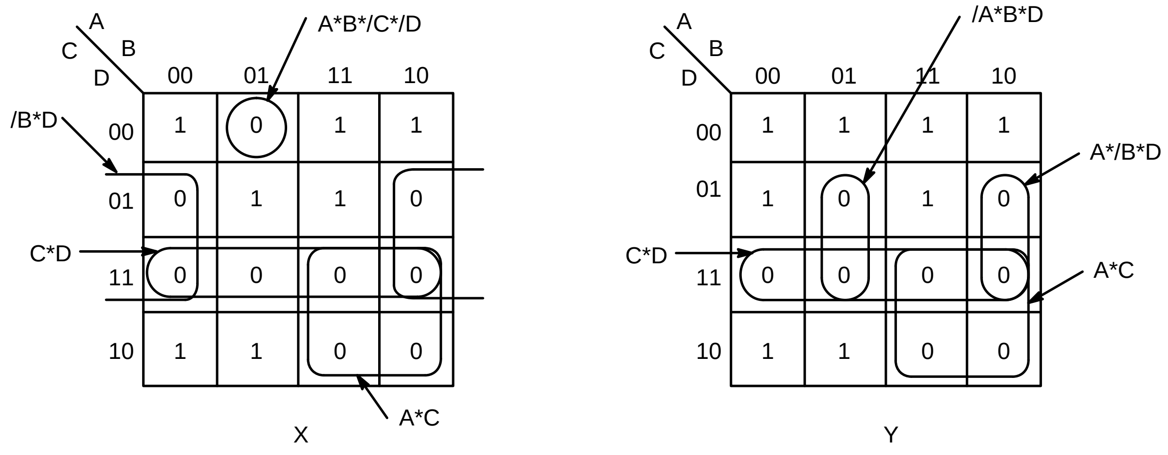


X = /A*/B*/D
+ A*/C*/D
+ B*/C*D
+ /A*C*/D

Y = /A*/D
+ /C*/D
+ /A*/B*/C
+ A*B*/C

90000A-4

Figure 5. Using a K-map to Minimize the Functions in Figure 3



/X = C*D
+ /B*D
+ A*C
+ /A*B*/C*/D

/Y = C*D
+ A*D
+ /A*B*D
+ A*/B*D

90000A-5

Figure 6. Finding Inverse Functions

Comparison and Equivalence: the XOR and XNOR Gates

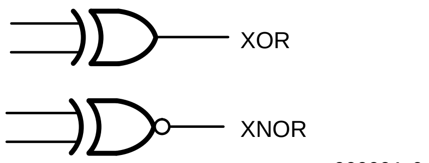
The Exclusive-OR (XOR) and Exclusive-NOR (XNOR) gates are two special gates which are relatively common. These gates have schematic symbols as shown in Figure 7a. They are actually compound gates, and can be generated by AND, OR, and NOT gates using the functions:

```
x :+: y = x*/y + /x*y  ;XOR gate
x :*: y = x*y + /x*/y  ;XNOR gate
```

The XOR and XNOR functions are actually inverses of each other; that is,

```
x :+: y = /(x :*: y)
```

The truth tables for these gates are shown in Figure 7b. Note that the XOR function is true if and only if the operands are different. For this reason, it is useful as a comparator. The XNOR function is true if and only if its operands are the same; therefore it is used as an equivalence indicator.



a. Schematic Symbols

A	B	A:+:B	A:*:B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

b. XOR and XNOR Truth Table

Figure 7. The Exclusive-OR and Exclusive-NOR Functions

Some basic properties of the XOR and XNOR functions are listed in Table 8.

Table 8. Properties of the XOR and XNOR Functions

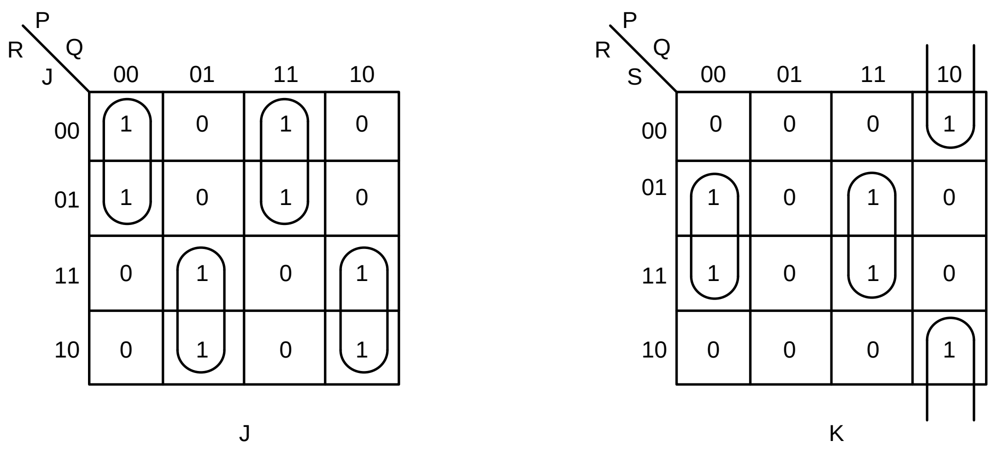
XOR	XNOR
x :+: 0 = x x :+ 1 = /x	x :*: 0 = /x x :*: 1 = x
x :+: x = 0 x :+: /x = 1	x :*: x = 0 x :*: /x = 1
x :+: y = y :+: x x :+: y = :+: z = (x :+: y) :+: z = x :+: (y :+: z)	x :*: y = y :*: x x :*: y :*: z = (x :*: y) :*: z) = x :*: (y :*: z)
x :+: y = /x :+: /y	x :*: y = /x :*: /y
/ (x :+: y) = /x :+: y = x :+: /y = x :+: y	/ (x :*: y) = /x :*: y = x :*: /y = x :+: y
x :+: y = x* /y + /x*y	x :*: y = x* y + /x*/y
x :+: x* y = x*/y x :+: /x*y = x + y	x :*: x* y = /x + y x :*: /x*y = /x * /y
x* (y :+: z) = (x*y) :+: (x*z) /x*(y :+: z) = (x + y) :+: (x + z)	x + (y :*: z) = (x + y) :*: (x + z) /x + (y :*: z) = (x*y) :*: (x*z)

When deriving equations from a Karnaugh map, XOR and XNOR functions can usually be identified by their characteristic pattern. Exactly what the operands are may or may not be obvious for more complicated functions. Some examples are shown in Figure 8.

The XOR gate can be used as an “UNLESS” operator. In other words, the function, A = X :+: Y can be interpreted as:

“A will have the same value as X UNLESS Y is true.”

This can be helpful when trying to derive a logic equation for a function which can be described in words.



J = /P*/Q*/R
+ P*Q*/R
+ /P*Q*R
+ P*/Q*R
= ((/P*/Q) + (P*Q))*/R
+ ((/P*Q)+(P*/Q))*R
= (P:*:Q)*/R
+ (P:+:Q)*R
= /(P:+:Q)*/R
+ (P:+:Q)*R

$\boxed{= (P{:}{+}{:}Q){:}{*}{:}R}$

K = /P*/Q*S
+ P*Q*S
+ P*/Q*/S
= ((/P*/Q) + (P*Q))*S
+ P*/Q*/S

$\boxed{\begin{array}{l} = (P{:}{*}{:}Q){*}S \\ + P{*}/Q/{*}/S \end{array}}$

90000A-7

Figure 8. Finding XOR and XNOR Functions in Karnaugh Maps

Basic Storage Elements

Storage elements provide circuits with the capability of remembering past conditions or events. The prototypical storage element is just a pair of cross-coupled NAND gates, as shown in Figure 9. These elements are normally called *flip-flops.*

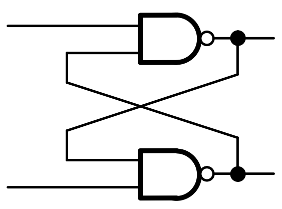


Figure 9. Basic Storage Element

In general, there are two primary classes of flip-flops:

- *Unclocked* flip-flops, or *latches*
- *Clocked* flip-flops

Clocked flip-flops are sometimes referred to as *registers,* although technically speaking, a register is a bank of several flip-flops with a common clock signal.

Flip-flops can also be characterized by their control scheme. There are four types of flip-flops, each of which can be unclocked or clocked:

- S-R
- J-K
- D
- T

The discussion below will be divided between unclocked and clocked flip-flops. Each of the four flip-flop types will be treated for each section.

Unclocked Flip-Flops—Latches

S-R Latches

An S-R latch can be built out of NOR gates as shown in Figure 10, and behaves according to the truth table in Table 9. ‘S’ stands for ‘set’ and ‘R’ stands for ‘reset,’ as suggested by the truth table.

Note that the latch actually has two outputs, which are complementary. These are referred to as Q and $\overline{Q}$. If both S and R are raised at the same time, then both Q and $\overline{Q}$ will be HIGH; although this is physically possible, it does not make sense if Q and $\overline{Q}$ are to be complementary signals. Thus, this condition is not allowed.

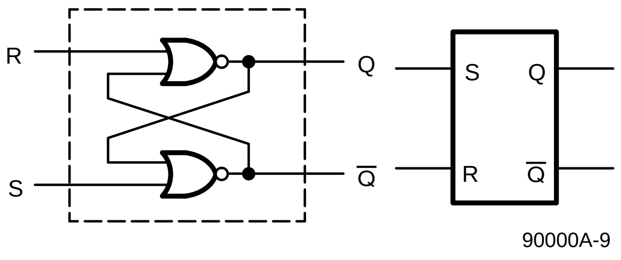


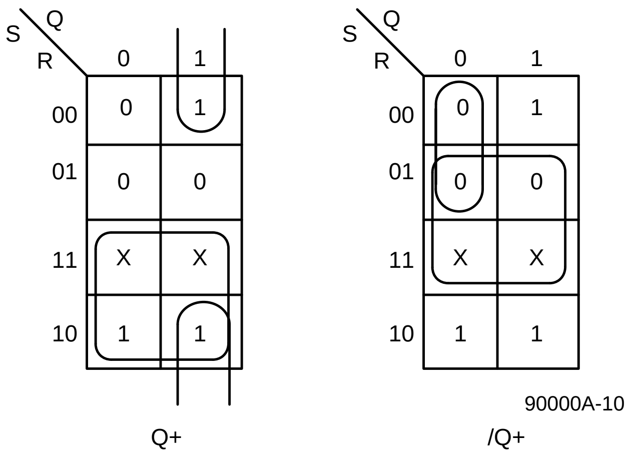
Figure 10. An S-R Latch

Table 9. S-R Latch Truth Table

S	R	Q+
0	0	Q
0	1	0
1	0	1
1	0	Not allowed

The transfer function for this latch can be derived with a Karnaugh map, as shown in Figure 11. By choosing either 1's or 0's, we can obtain two representations:

a. Q+ = S+/R*Q
b. /Q+ = R+/S*/Q



a. Q+ = S + /R*Q

b. /Q+ = R + /S*/Q

Figure 11. Karnaugh Map for an S-R Latch

Waveforms illustrating the operation of the S-R latch are shown in Figure 12.

There are some applications where it is desirable for the input data to be effective only when another signal—usually called a control signal—is active. The circuit of Figure 10 can be modified to give an S-R latch with a control input, as shown in Figure 13. The operation of this circuit is summarized in Table 10 and Figure 14.

The S-R latch is somewhat restrictive, since both inputs cannot be HIGH at the same time. The other latch types are based on the S-R latch, but have additional logic which removes the input restrictions.

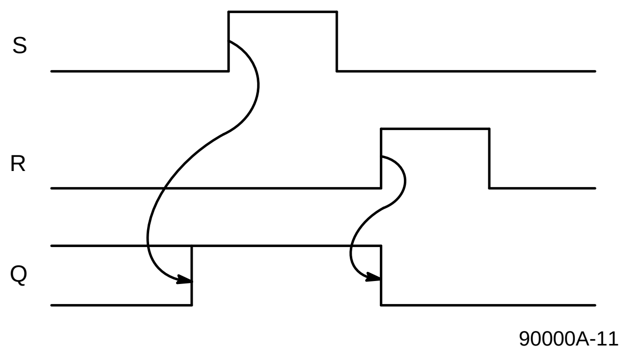


Figure 12. S-R Latch Behavior

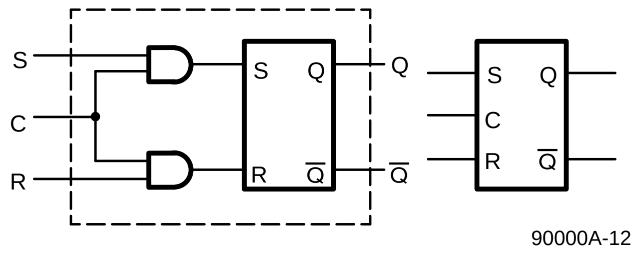


Figure 13. Adding a Control Input to an S-R Latch

Table 10. Truth Table for an S-R Latch with a Control Input

S	R	C	Q+
X	X	0	Q
0	0	1	Q
0	1	1	0
1	0	1	1
1	1	1	Not allowed

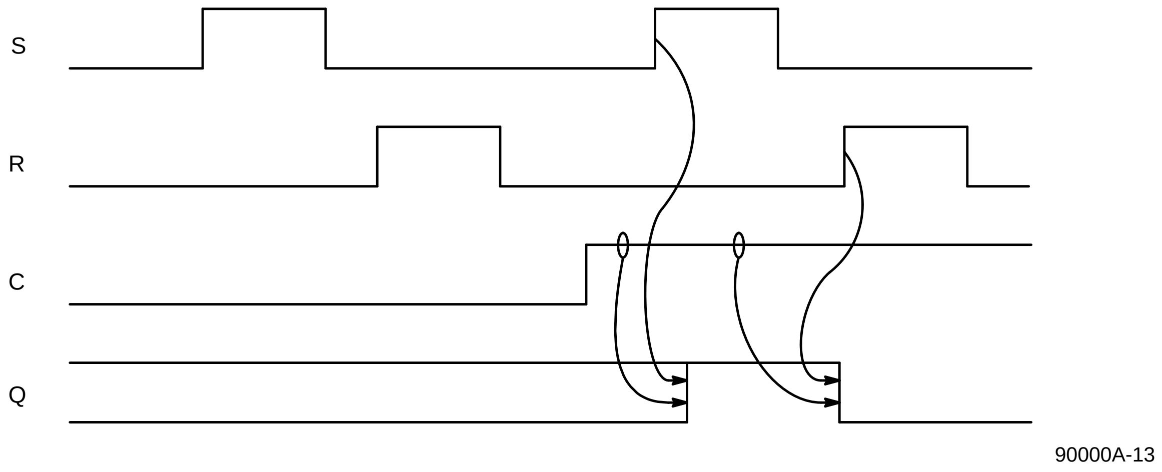


Figure 14. Behavior of an S-R Latch with a Control Input

D-Type Latches (Transparent Latches)

A single-input latch can be formed by adding some logic to the controlled S-R latch in Figure 13; this gives rise to the D-type latch in Figure 15. This latch is often called a *transparent* latch, since data on the input passes right through to the output as long as the control input is HIGH. If the control input is set LOW, then the latch holds whatever data was present when the control went LOW. With this type of latch, the control is usually called a *gate.*

The behavior of the D-type latch is shown in Table 11 and Figure 16.

The basic transfer function for a D-type latch can be derived from the Karnaugh map in Figure 17.

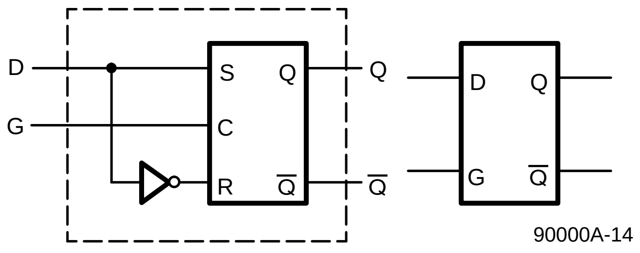


Figure 15. A D-Type (Transparent) Latch

Table 11. Truth Table for a D-Type Latch

D	G	Q+
X	0	Q
0	1	0
1	1	1

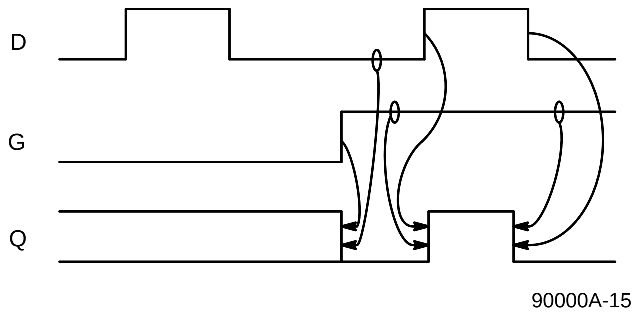
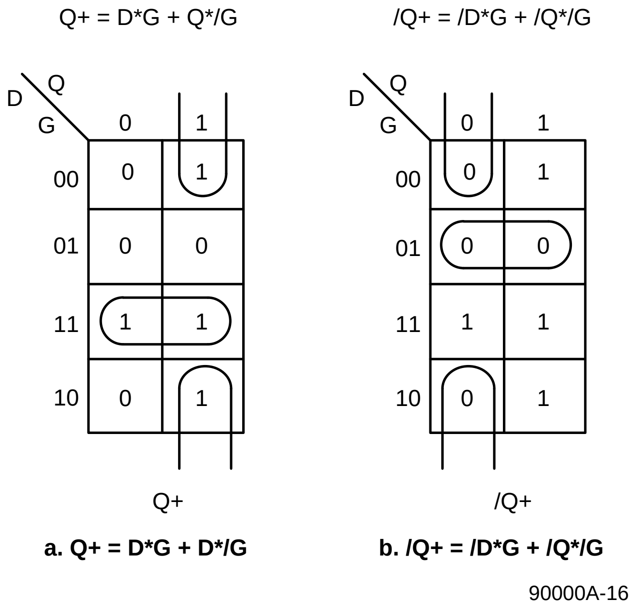


Figure 16. D-Type (Transparent) Latch Behavior



a. Q+ = D*G + D*/G

b. /Q+ = /D*G + /Q*/G

Figure 17. Karnaugh Maps for a D-Type Latch

If realized exactly as the transfer function indicates, the result is actually a glitchy circuit.

J-K Latches

Another two-input latch can be derived from the S-R latch as shown in Figure 18. This is called a J-K latch, and operates in the same manner as an S-R latch, except that the condition where both inputs are HIGH is now allowed. The truth table is shown in Table 12; the waveforms are shown in Figure 19.

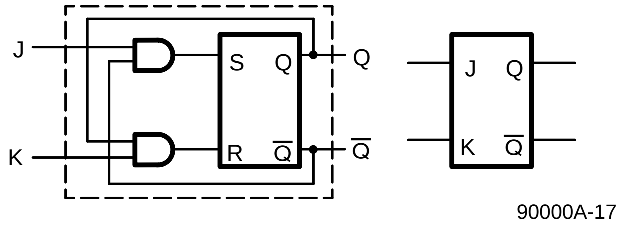


Figure 18. A J-K Latch

Table 12. Truth Table for a J-K Latch

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

There are still some potential problems here for the case where J and K are both HIGH. If J and K are left HIGH for too long, the output may change more than one time; if left HIGH forever, the output will oscillate. Thus, J and K should not be asserted for a time longer than the propagation delay of the latch. There are also potential race conditions if J and K are not asserted and removed at exactly the same time. If one of the inputs is raised slightly ahead of the other, it may give the output time to react, giving the wrong output once the second input is raised. The same problem can occur if one input is lowered slightly before the other. This is illustrated in Figure 20.

There are several ways to derive transfer functions for J-K latches. Two can be derived directly from Karnaugh maps, as shown in Figure 21; the others are not as obvious, and make use of the XOR gate described before. The basic transfer functions are listed in Table 13.

Table 13. Transfer Functions for a J-K Latch

Q + = J * /Q + /K*Q	/Q+ = /J* /Q + K*Q
Q+ = Q :+: (J* /Q + K*Q)	/Q+ = /Q :+: (J*/Q + K*Q)
Q+ = /Q :+: (/J*/Q + /K*Q)	/Q+ = Q :+: (/J*/Q + /K*Q)

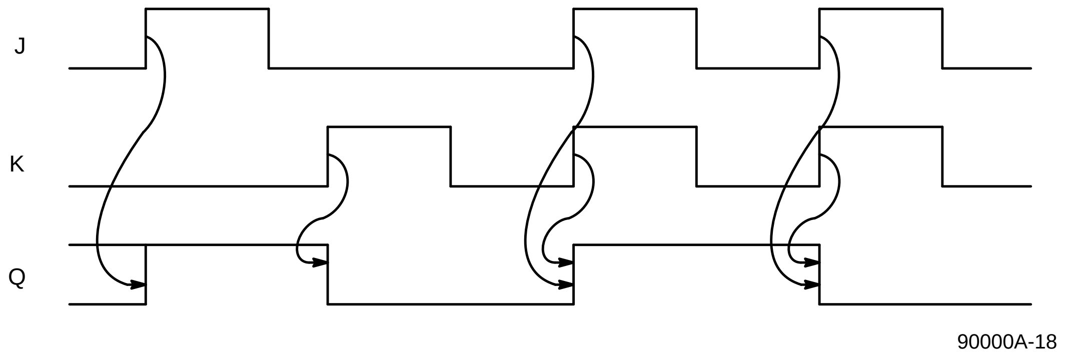
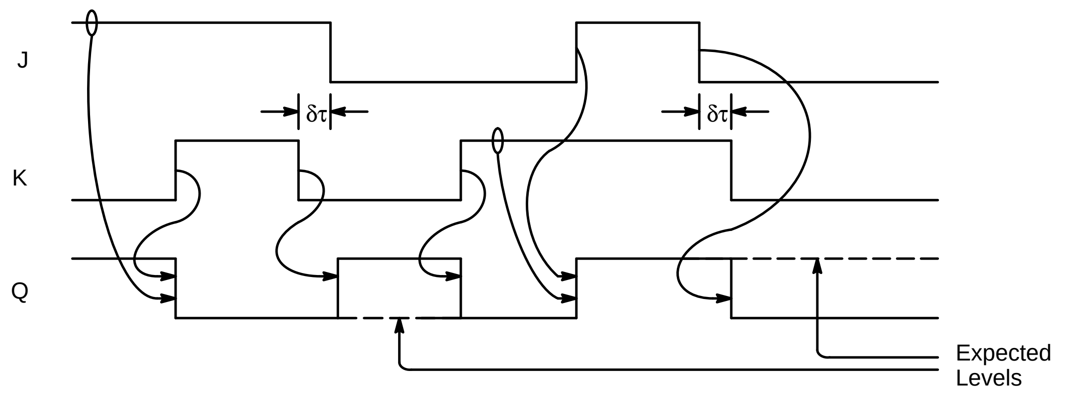
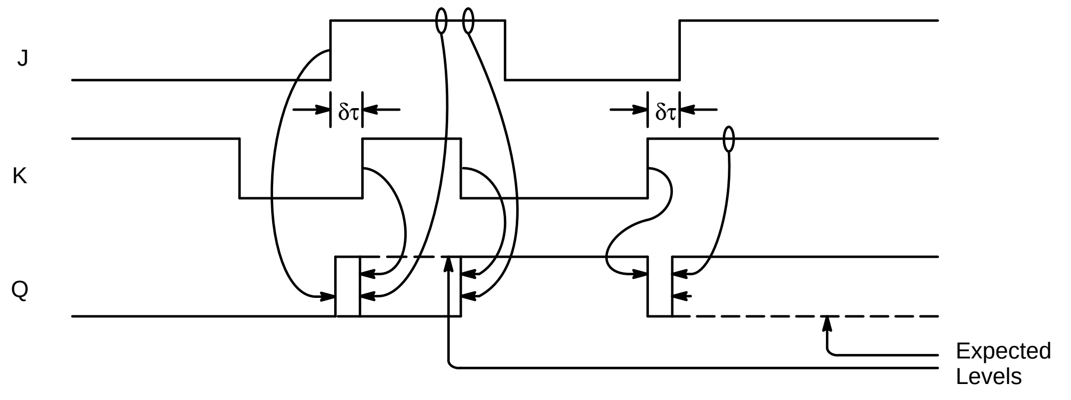


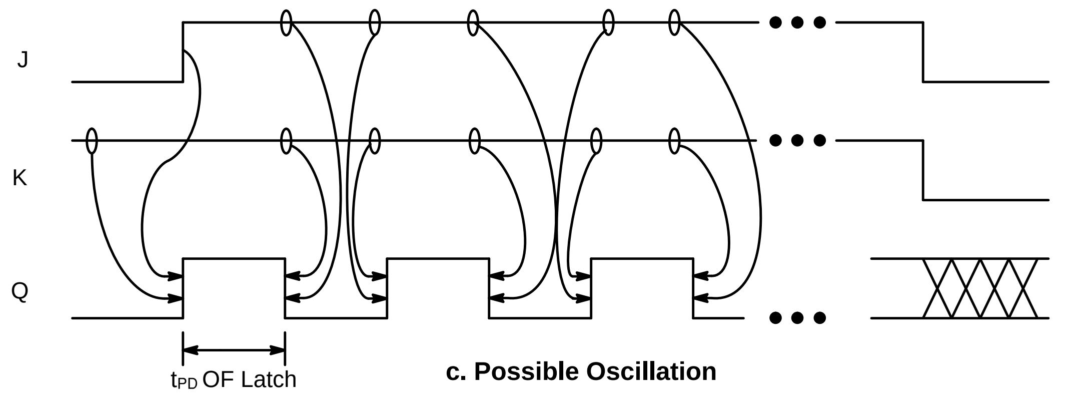
Figure 19. Behavior of a J-K Latch



a. Falling Edge Race Conditions



b. Rising Edge Race Conditions



c. Possible Oscillation

90000A-19

Figure 20. Hazards Inherent in a J-K Latch

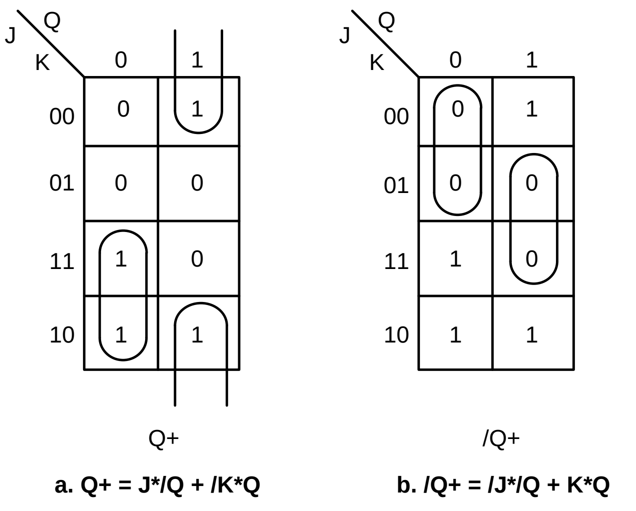


Figure 21. Karnaugh Maps for a J-K Latch

T-Type Latches

T-type latches are formed by connecting the J and K inputs of a J-K latch together to form a single input, as shown in Figure 22. This latch has two possible functions: hold the present state or invert the output, as summarized in Table 14. ‘T’ stands for ‘trigger’ or ‘toggle’ depending on who you talk to. That is, when T is HIGH, a change at the output is triggered; or, put another way, raising T causes the output to toggle.

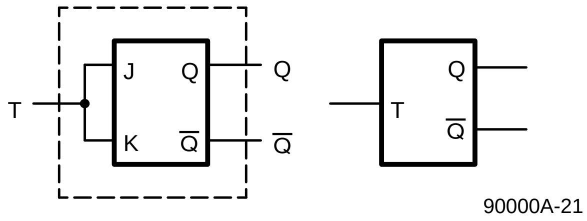


Figure 22. A T-Type Latch

Table 14. The Truth Table for a T-Type Latch

T	Q+
0	Q
1	/Q

This Latch also has the problem that if T is left HIGH for too long, the output will oscillate. However, since there is only one input, the race condition problems of the J-K latch have been eliminated. Unfortunately, this comes at the cost of initialization. There is now no way to get the output into a fixed state without knowing what the previous state was. Thus, this device is not very useful without some kind of initialization circuit.

The general waveforms for a T-type latch are shown in Figure 23.

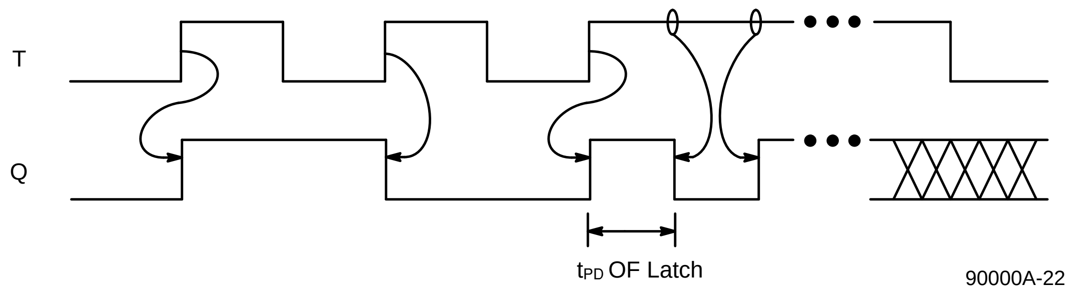
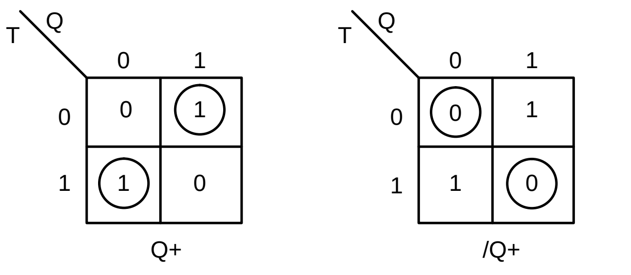


Figure 23. Behavior of a T-Type Latch

From the Karnaugh map in Figure 24, we can generate the following transfer functions:

```
Q+  =  T*/Q              /Q+   =  T*Q
     + /T*Q                     + /T*/Q

Q+  =  Q:+:T             /Q+ = /Q :+: T
Q+  =  /Q:+: /T    /Q+ = Q :+: /T
```



a. Q+ = T*/Q + /T*Q   **b. /Q+ = T*Q + /T*/Q**

90000A-23

Figure 24. Karnaugh Maps for a T-Type Latch

Clocked Flip-Flops

Latches can be modified by adding a *clock* input. The purpose of the clock is to delay any output changes until the clock signal changes. Whereas latch control inputs (such as the gate) are *level-sensitive,* clock inputs are generally *edge-sensitive (*or *edge-triggered),* meaning that output transitions can occur only when a clock transition is detected. A device is classified as positive edge-triggered or negative edge-triggered, depending on whether it responds to the rising or falling edge of the clock signal, respectively. The behavior to a clocked S-R flip-flop is illustrated in Figure 25.

The clock provides two basic advantages. It removes the hazards inherent in the J-K and T flip-flops, since all inputs will have settled by the time the clock edge arrives, and only one transition is possible for each clock edge. The clock also allows the design of synchronous systems, where all signals are coordinated with other signals. The entire system is then regulated by the clock.

The basic behavior of the four flip-flops types does not change with the addition of a clock; the output changes are merely made to wait for the clock edge. Thus, the basic transfer equations for most of the flip-flops are the same. We can indicate the clocked nature of the flip-flops by using the “registered” assignment ‘:=’ instead of ‘=.’

D-Type Flip-Flops

This is the only flip-flop type whose basic transfer characteristic changes, because the clock input replaces the gate input. Thus the transfer equations become:

Q+:= D/Q+   := /D

That is, whatever data appears on the input will be transferred to the output after the next clock edge. The input is not changed in any way.

The simplicity of this flip-flop makes it the most widely used flip-flop. However, functions are sometimes more conveniently expressed using J-K flip-flops, or using T-type flip-flops. If we replace the D signal with the transfer function for one of the other flip-flop types, we can then emulate that flip-flop type in the D-type flip-flop. This is equivalent to taking a latch and placing a clocked D-type flip-flop after the latch output for synchronization. Figure 26 illustrates how each flip-flop can be emulated in a D-type flip-flop. The standard schematic symbols for the flip-flop types are also shown.

Table 15 summarizes the transfer functions for all of the flip-flop types. These functions can directly be used to emulate a particular flip-flop type in a D-type flip-flop. This can be particularly useful since D-type flip-flops are available in most registered PLDs.

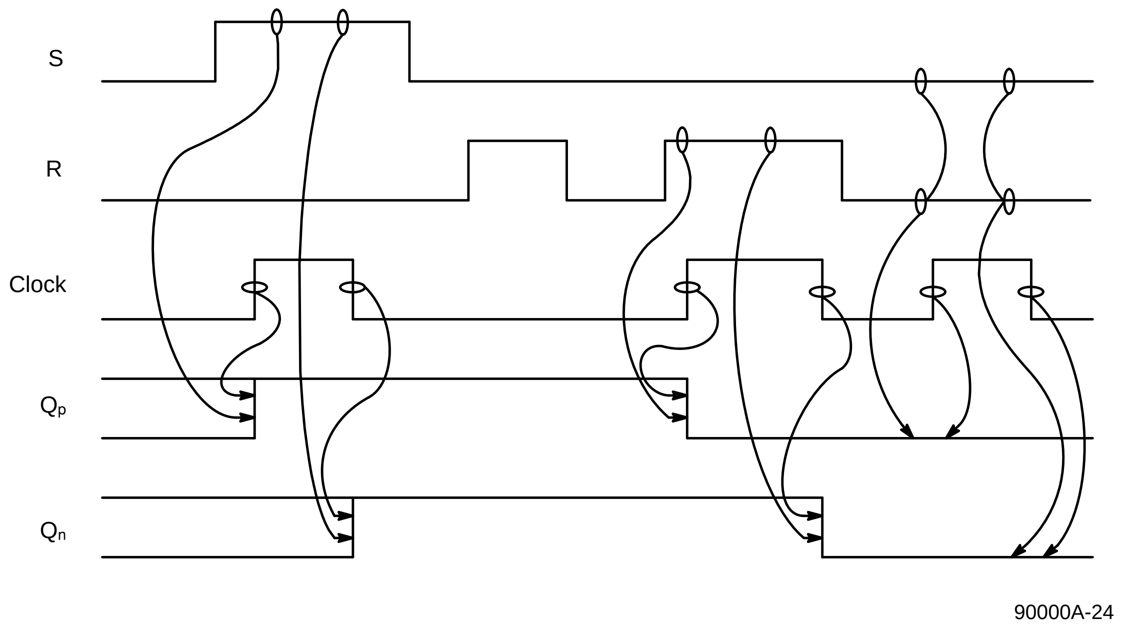
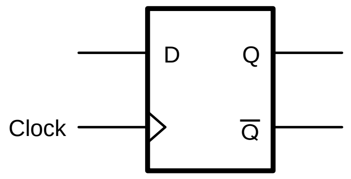
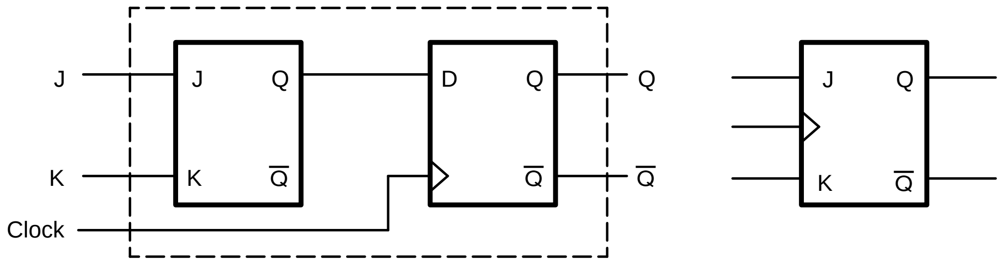


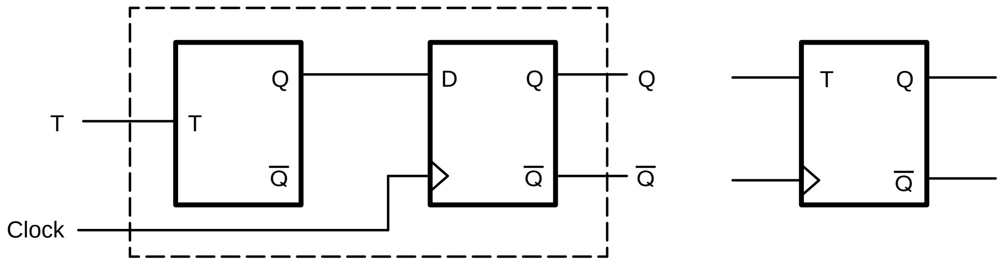
Figure 25. Behavior of a Clocked S-R Flip-Flop for Positive (Qp) and Negative (Qn) Edge-Triggered S-R Flip-Flops



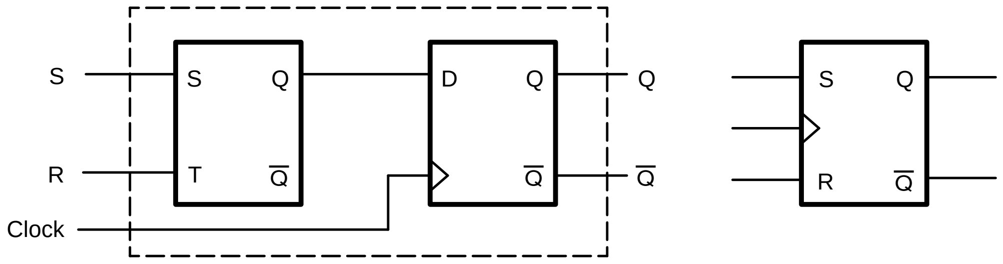
a. Clocked D-Type Flip-Flop



b. Clocked J-K Flip-Flop



c. Clocked T-Type Flip-Flop



d. Clocked S-R Flip-Flop

90000A-25

Figure 26. Clocked Flip-Flops. All can be Emulated with a D-Type Flip-Flop

Table 15. Clocked Flip-Flop Transfer Functions

D-Type	Q+ := D	/Q+ := /D
J-K-Type	Q+ := J*/Q + /K*Q Q+ := Q :+: (J*/Q + K*Q) Q+ := /Q :+: (/J*/Q + /K*Q)	/Q+ := /J* /Q + K*Q /Q+ := /Q :+: (J*/Q + K*Q) /Q+ := Q :+: (/J*/Q + /K*Q)
T-Type	Q+ := T*/Q + /T*Q Q+ := Q:+:T Q+ := /Q :+: /T	/Q+: = T*/Q + /T*/Q /Q+ := /Q :+: T /Q+ := Q :+: /T
S-R-Type	Q+ := S + /R*Q	/Q+ := R + /S*/Q

Binary Numbers

The concept of *a number is* taken for granted by most people. And most people equate numbers in general with the *decimal* system, with which we are most familiar. However, there is nothing particularly special about the decimal system; the choice of system is actually rather arbitrary. History has chosen the decimal system for most humans.

For electronic systems, the *binary* system is more appropriate. It makes possible arithmetic and logical calculations that would be much more difficult—likely impractical—if implemented directly in a decimal system. Closely related to the binary system are the *octal* and *hexadecimal* systems, which will also be discussed here. Arithmetic is normally performed using binary numbers in a computer. Octal and hexadecimal representations are generally used as a way to “abbreviate” what might otherwise be lengthy binary numbers. This will be seen when conversion is discussed below.

There are several terms which must be defined before proceeding further. A *number* is an abstract entity which is used to describe quantity. There are many ways of representing a number. Normally, the representation is designed around a *base.* The number is expressed as a sum of multiples of the powers of the base. The decimal system is a base-10 system, meaning that 10 is used as the base. The binary system is base-2; the octal system is base-8; and the hexadecimal system is base-16. The binary, octal, and hexadecimal systems are closely related because 8 and 16 are both powers of 2. When different bases are being used, a number will often be followed by its base in subscript, to indicate exactly what the base is. For example, the decimal number 25 would be written 25_{10} if its base were in doubt.

A number can thus be expressed in terms of some base x as follows:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0 + a_{-1} x^{-1} + \ldots + a_{-m} X^{-m} \quad (1)$$

The numbers $a_n \ldots a_{-m}$ are called *digits.* The value of each digit can range from 0 to x–1. Each digit is represented by a symbol, called a *numeral.* X numerals are required to represent a number in base x. The most familiar numerals are the symbols ‘0,’ ‘1,’...‘9.’ There are ten of them, since they are used for the decimal system. For binary numbers, only ‘0’ and ‘1’ are used; for octal numbers, the numerals ‘0’ through ‘7’ are used. Hexadecimal numbers are more difficult, since sixteen numerals are required. Therefore, the numerals ‘0’ through ‘9’ are used to represent the quantities 0_{10} through 9_{10}; the letters A through F are used to represent the quantities 10_{10} through 15_{10}.

The number expressed by equation 1 is normally represented as a string of digits:

$$a_n a_{n-1} \ldots a_1 a_0 . a_{-1} \ldots a_{-m}$$

The digits representing negative powers of the base are separated from those representing non-negative powers by *a point.* In the decimal system, this is referred to as a *decimal point;* in the binary system, it is referred to as a *binary point.*

There are two basic classes of manipulation which will be discussed: conversions between bases and arithmetic within a base.

Converting Between Bases

Base-2 <–> Base-10

Converting a binary number to a decimal number is accomplished by using equation 1 directly.

Example:

Converting 110100.011_2 to decimal:

$$
\begin{aligned}
Y &= 110100.011 \\
&= 1\bullet2^5 + 1\bullet2^4 + 0\bullet2^3 + 1\bullet2^2 + 0\bullet2^1 + 0\bullet2^0 + 0\bullet2^{-1} + 1\bullet2^{-2} + 1\bullet2^{-3} \\
&= 32 + 16 + 4 + .25 + .125 \\
&= 52.375
\end{aligned}
$$

When converting whole numbers from decimal to binary, the decimal number is repeatedly divided by 2. Integer division is used, so the quotients are "rounded down" to the next integer. The remainders form the digits of the number. The least significant digit is the first one calculated.

Example:

Converting 61_{10} to binary:

61/2 = 30	remainder = 1	LSB
30/2 = 15	remainder = 0	
15/2 = 7	remainder = 1	
7/2 = 3	remainder = 1	
3/2 = 1	remainder = 1	
1/2 = 0	remainder = 1	MSB

$61_{10} = 111101_2$

When converting a decimal fraction into a binary fraction, the decimal number is multiplied by 2. This results in a whole number and a fraction. The whole number is a digit; the procedure is repeated on the new fraction. This procedure is repeated until the fractional portion is zero. If the procedure does not terminate, then the result is a repeating fraction. The first digit calculated is the most significant digit.

Example:

Converting $.1625_{10}$ to binary:

0.1625•2 = 0.3250	whole portion = 0	MSB
0.3250•2 = 0.65	whole portion = 0	
0.65•2 = 1.3	whole portion = 1	
0.3•2 = 0.6	whole portion = 0	
0.6•2 = 1.2	whole portion = 1	
0.2•2 = 0.4	whole portion = 0	
0.4•2 = 0.8	whole portion = 0	
0.8•2 = 1.6	whole portion = 1	
0.6•2 = 1.2	whole portion = 1	

Here we see that the fraction will repeat, since we have already multiplied 0.6 earlier. Thus

$0.1625_{10} = 0.00101001100110011..._2$

For mixed numbers, it is necessary to calculate the whole and fractional portions separately. Thus, for example, we know that

$61.1625_{10} = 111101.0010100110011..._2$

These are actually general procedures which can be used to convert a decimal number into any base, and vice versa.

Examples:

1. Converting 321.54_8 to decimal:

$$
\begin{aligned}
Y &= 3\bullet8^2 + 2\bullet8^1 + 1\bullet8^0 + 5\bullet8^{-1} + 4\bullet8^{-2} \\
&= 192 + 16 + 1 + .625 + .0625 \\
&= 209.6875
\end{aligned}
$$

$321.54_8 = 209.6875_{10}$

2. Converting 106.10375_{10} to octal:

106/8 = 13	remainder = 2	LSB
13/8 = 1	remainder = 5	
1/8 = 0	remainder = 1	MSB

Thus, the whole portion is 151_8.

0.10375•8 = 0.83	whole portion = 0	MSB
0.83•8 = 6.64	whole portion = 6	
0.64•8 = 5.12	whole portion = 5	
0.12•8 = 0.96	whole portion = 0	
0.96•8 = 7.68	whole portion = 7	
0.68•8 = 5.44	whole portion = 5	

At this point we have enough significant digits. We could continue either until the procedure terminated, or until the pattern started repeating. However, those last digits are not likely to be significant. Thus, we can approximate by saying that...

$106.10375_{10} = 152.065075_8$

3. Converting $31F.A2_{16}$ to decimal:

$$
\begin{aligned}
Y &= 31F.A2_{16} \\
&= 3\bullet16^2 + 1\bullet16^1 + 15\bullet16^0 + 10\bullet16^{-1} + 2\bullet16^{-2} \\
&= 768 + 16 + 15 + 0.625 + 0.0078125 \\
&= 799.6328125
\end{aligned}
$$

$31F.A2_{16} = 799.6328125_{10}$

4. Converting 7689.100854_{10} to hexadecimal:

7689/16 = 480	remainder = 9	LSB
480/16 = 30	remainder = 0	
30/16 = 1	remainder = E	
1/16 = 0	remainder = 1	MSB

Thus, the whole portion is $1EO9_{16}$.

0.100854•16 = 1.613664	whole portion = 1	MSB
0.613664•16 = 9.818624	whole portion = 9	
0.818624•16 = 13.097984	whole portion = D	
0.097984•16 = 1.567744	whole portion = 1	
0.567744•16 = 9.083904	whole portion = 9	
0.083904•16 = 1.342464	whole portion = 1	

Again, we likely have enough digits at this point. The exact fraction could be either very long or a long repeating pattern. For our purposes, we can approximate the overall result as:

$7689.100854_{10} = 1E09.19D191_{16}$

Binary <–> Octal, Hexadecimal

Converting between the binary-related systems is very easy. The procedure consists of dividing the binary digits into groups, and replacing each group with an appropriate digit. For this reason, octal and hexadecimal numbers are often used to shorten long binary numbers.

To convert from binary to octal, group the digits by three, starting on each side of the binary point, and then convert each group of three digits into its corresponding octal digit. Leading and trailing zeroes may have to be added to the left of the whole portion and the right of the fractional portion, respectively, to make complete groups of three binary digits.

Example:

Converting $11011010110101.001001101_2$ to octal:

Divide into groups of three digits:

```
011  011   010   110  101   .  001    001    101
 3    3     2     6    5    .   1      1      5
```

Thus $11011010110101.001001101_2 = 33265.115_8$

To convert from binary to hexadecimal, the digits are divided into groups of four digits, and then given their corresponding hexadecimal digits. Again, leading and/or trailing zeroes may be needed.

Example:

Converting $100101011101100.110110001_2$ to hexadecimal:

Divide into groups of four digits:

```
0100  1010  1110 1100   .    11011000 1000
 4     A     E    C     .     D    8    8
```

Thus $100101011101100.110110001_2 = 4AEC.D88_{16}$

To convert from octal or hexadecimal to binary, merely expand each digit into its corresponding binary representation.

Examples:

1. Convert 7324.34_8 to binary:

```
 7    3    2    4    .   3    4
111  011  010  100   .  011  100
```

Thus $7324.34_8 = 111011010100.0111_2$

2. Convert $1A2.3F5_{16}$ to binary:

```
 1     A     2     .    3     F     5
0001  1010  0010   .  0011  1111  0101
```

Thus $1A2.3F5_{16} = 110100010.001111110101_2$

Binary Arithmetic

Positive binary arithmetic is very simple, and completely analogous to decimal arithmetic. However, if we are restricted to positive numbers, then we are also restricted to addition. We need a means of representing negative numbers. Using a dash '–' is unacceptable for representation in a computer. There are two general schemes which can be used. In binary systems, they are referred to as *1s complement* and *2s complement* representation, although they can be generalized for any base system as *diminished-radix complement* and *radix complement* representation.

One's Complement Representation

The one's complement of a binary number can be calculated by inverting all of the bits of the number. Fractions are handled exactly the same way, although this is convenient only for fixed-point arithmetic. Floating-point arithmetic requires other methods, which will not be discussed here.

Example:

Finding the one's complement of 110111.0101:

110111.0101
001000.1010 (Inverting each bit)

Thus, the one's complement of 110111.0101 is 001000.1010.

The sign of a number is determined by the most significant bit. If the MSB is 0 the number is positive; if the MSB is 1, then the number is negative. Zero is represented by all bits being zero. However, one normally thinks of zero as being its own complement. But if we take the one's complement of zero,

0000
1111

we see that 1111 is another representation of zero. Thus, in an eight-bit representation, positive numbers range from 00000001 to 01111111; negative numbers range from 10000000 to 11111110. Note that there are just as many negative numbers as positive numbers.

This eight-bit code allows us to represent the numbers from –127 to +127.

When performing addition with one's complement numbers, it is important to watch for overflow results. Whenever an overflow occurs, a correction must be made by adding 1 to the result.

In some cases, the results of an operation will not be meaningful, since the intended result cannot be represented. For instance, in the eight-bit system above, adding 127 to 127 will give a meaningless result, since 254 cannot be represented in this system. Thus, the operation must be evaluated to ensure that the result is meaningful.

Examples:

All examples will use 4-bit systems. Thus, the range of representable numbers is from –7 to +7.

Add 3 + 2:

```
       0011        3
  +    0010    +   2
  ---------    -----
       0101        5   result meaningful
```

Add 7 + 7 (14 cannot be represented):

```
       0111        7
  +    0111    +   7
  ---------    -----
       1110       –1   result
                       meaningless
```

Subtract 3 from 7:

```
       0111        7
  +    1100    +  –3
  ---------    -----
      10011            overflow – add 1,
         +1            discard overflow
  ---------
       0100        4   bit
```

Subtract 5 from 2:

```
       0010        2
  +    1010    +  –5
  ---------    -----
       1100       –3   result meaningful
```

Subtract 6 from –5 (–11 cannot be represented):

```
       1010       –5
  +    1001    +  –6
  ---------    -----
      10011            overflow – add 1,
         +1            discard overflow bit
  ---------
       0100        4   result meaningless
```

Subtract 5.25 from 3.5 (fixed point; requires 6 bits):

```
    0011.10      3.5
  + 1010.10   + –5.25
  ---------   -------
    1110.00    –1.75   result meaningful
```

Subtract 7 from 7:

```
       0111        7
  +    1000    +  –7
  ---------    -----
       1111        0   one of the
                       representations of 0
```

The advantage of one’s complement code is the fact that it is easy to compute the complement. However, the fact that there are two representations for zero is a problem. In addition, the results of subtraction frequently have to be adjusted for overflow by adding 1.

Two’s Complement Representation

The two’s complement of a binary number is more difficult to calculate. It is generated by taking the one’s complement, and then adding 1. Any overflow is discarded. Fractions are again handled in the same way, although 1 is added to the least significant bit.

Example:

Finding the two’s complement of 110111.0101:

```
 110111.0101
 001000.1010      (take one’s
          +1      complement)
 -----------
 001000.1011
```

Thus, the two’s complement of 110111.0101 is 001000.1011.

The sign of a number is again determined by the most significant bit. If the MSB is 0 the number is positive; if the MSB is 1, then the number is negative. Zero is represented by all bits being zero. In this case, if we take the two’s complement of zero, we get:

```
 0000
 1111
   +1
 ----
 0000    (overflow is discarded)
```

giving only one representation for zero.

Thus, in an eight-bit representation, positive numbers range from 00000001 to 01111111; negative numbers range from 10000000 to 11111111. This means that there is one more negative number than there are positive numbers. So this eight-bit code allows us to represent the numbers from –128 to +127.

Addition is handled in the same fashion as with one’s complement code, except that when an overflow occurs, the overflow bit is disregarded. No correction must be made to the results.

After any operation, one must still make sure that the results are meaningful.

Examples:

Add 3 + 2:

```
       0011        3
 +     0010   +    2
 ----------   ------
       0101        5     result meaningful
```

Add 7 + 7 (14 cannot be represented):

```
       0111        7
 +     0111   +    7
 ----------   ------
       1110       –2     result meaningless
```

Subtract 3 from 7:

```
       0111        7
 +     1101   +   –3
 ----------   ------
      10100        4     overflow – discard
                         overflow bit
```

Subtract 5 from 2:

```
       0010        2
 +     1011   +   –5
 ----------   ------
       1101       –3     result meaningful
```

Subtract 6 from –5 (–11 cannot be represented):

```
       1011       –5
 +     1010   +   –6
 ----------   ------
      10101        5     overflow – discard
                         overflow bit result
                         meaningless
```

Subtract 5.25 from 3.5 (fixed point; requires 6 bits):

```
    0011.10        3.5
 +  1010.11   +  –5.25
 ----------   --------
    1110.01      –1.75   result meaningful
```

Subtract 7 from 7:

```
       0111        7
 +     1001   +   –7
 ----------   ------
      10000        0     overflow – disregard
                         overflow bit
```

The benefits of two’s complement lie in the fact that there is only one representation for zero, and the fact that the results of operations never need adjusting due to overflow. The disadvantage is the fact that it is harder to generate the two’s complement of a number.