Chapter Seven

Introduction to Digital Electronics

1.1. Decimal, binary, and hexadecimal numbers

We all know the decimal number system. For example, 2019 means 2*1000 + 0*100 + 1*10 + 9*1. The numbers 2, 0, 1, 9 are called the digits of the number 2019.

If we want to describe this mathematically, we will call the rightmost digit d0, the next digit d1, etc. If there are *n* digits: d0, d1, d2, ... dn-1, then the value v can be calculated by the formula

$$\mathrm{v} = \sum_{i=0}^{n-1} di10i$$

In the example of 2019, we have d0 = 9, d1 = 1, d2 = 0, d3 = 2, n = 4. This gives v = 2019.

10 is the base or radix of the number system. The reason why we are using ten as the radix is that this makes it easy to use our ten fingers for counting. The name *decimal* comes from Latin *decem* which means ten. The word *digit* means finger.

If we use a number system with radix *r* then the above formula becomes

$$\mathrm{v} = \sum_{i=0}^{n-1} dir\ i$$

For example, if the radix *r* is sixteen then 2019 does not mean two thousand and nineteen. Instead the value becomes

2019(base 16) = 9*160 + 1*161 + 0*162 + 2*163 = 8217(base 10).

1.1.1. Binary numbers

Computers do not have ten fingers so the decimal number system is not the most efficient system to use in computers. Instead, the binary system with radix 2 is used. For example,
1101(base 2) = 1*20 + 0*21 + 1*22 + 1*23 = 13(base 10). The rightmost digit is the least significant digit with
the worth 20 = 1. We can call the least significant the 1's, the next digit from the right is the 2's. Next comes the 4's, the 8's, etc. The calculation is illustrated in this table:

8's	4's	2's	1's
1	1	0	1
1*8=8	1*4=4	0*2=0	1*1=1
8 + 4 + 0 + 1 = 13			

Each digit in a binary number can have only two different values: 0 and 1. These are conveniently represented in an electrical wire as two different voltages. The lower of the two voltages represents 0 and the higher voltage represents 1. For example, we may choose 0V and 5V for the numbers 0 and 1, respectively. A binary number with four digits, as in the example above, can be represented by four wires, where each wire has either 0V or 5V.

A binary digit is also called a bit. A bit can be 0 or 1. This is the smallest piece of information that you can store in any system.

1.1.2. Hexadecimal numbers

The hexadecimal number system has radix 16. The sixteen possible digits are written as this:

Digit	Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

For example 4AD1(base 16) = 1*160 + 13*161 + 10*162 + 4*163 = 19153(base 10).

The hexadecimal system is often used as a short way to represent binary numbers with many digits. Each hexadecimal digit corresponds to four binary digits, because 16 = 24. For example, the binary number 0100101011010001(base 2) is easily converted to the hexadecimal number 4AD1(base 16) by dividing the bits into groups of four:

$$\underbrace{0100}_{4}\ \underbrace{1010}_{A}\ \underbrace{1101}_{D}\ \underbrace{0001}_{1}$$

1.2. Conversion from another number system to decimal

There are two convenient ways to convert from any other number system to decimal. The first method starts with the rightmost digit (least significant digit) and multiplies by powers of the radix. We used this method above to convert the number 4AD1 from hexadecimal to decimal:

4AD1(base 16) = 1*160 + 13*161 + 10*162 + 4*163 = 19153(base 10).

The second method starts with the leftmost digit (most significant digit). Multiply the leftmost digit by the radix. Add the next digit. Multiply the result by the radix. Add the next digit, and so on. Stop after adding the last digit. Do not multiply by the radix after adding the last digit.

If we apply this method to the same example, we get:

4AD1(base 16) = ((4 * 16 + 10) * 16 + 13) * 16 + 1 = 19153(base 10).

The second method is convenient to use with a pocket calculator.

1.3. Conversion from decimal to another number system

There are also two ways to convert from decimal to another number system. The first method gives the rightmost digit (least significant digit) first: Divide the number repeatedly by the radix and get the integer part of each result. Save the remainders from each division. The remainders represent the converted number with the least significant digit first. This method is illustrated with the same example as above:

Convert 19153 from decimal to hexadecimal:

Number	Division	Fraction	Remainder
19153	19153 / 16 = 1197, remainder 1	1197	1
1197	1197 / 16 = 74, remainder 13	74	13
74	74 / 16 = 4, remainder 10	4	10
4	4 / 16 = 0, remainder 4	0	4

You get the result from the remainders in reverse order: 4 : 10 : 13 : 1 gives 4AD1(base 16).

The second method gives the leftmost digit (most significant digit) first: Find the highest power of the radix that is not bigger than the number you want to convert. Divide by the highest power of the radix first. The result of the first division is the most significant digit of the result. Divide the remainder by the next lower power of the radix, and so no. Our example gives these results:

Convert 19153 from decimal to hexadecimal:

Number	Division	Fraction	Remainder
19153	19153 / 16^3= 4, remainder 2769	4	2769
2769	2769 / 16^2= 10, remainder 209	10	209
209	209 / 16^1= 13, remainder 1	13	1
1	1 / 16^0= 1, remainder 0	1	0

You get the result from the fractions: 4 : 10 : 13 : 1 gives 4AD1(base 16).

You can choose the method you think is most logical or easiest to remember.

1.3.1. Conversion from hexadecimal to binary

It is easy to convert a number from hexadecimal to binary. Just convert each hexadecimal digit to four bits. If any digit gives less than four bits then you must remember to put zeroes in front of the number to get four bits.

Converting 4AD1 from hexadecimal to binary:

$$\begin{array}{cccc} 4 & A & D & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 0100 & 1010 & 1101 & 0001 \end{array}$$

0100 : 1010 : 1101 : 0001 gives 0100101011010001 = 100101011010001.

1.3.2. Conversion from binary to hexadecimal

To convert from binary to hexadecimal, you divide the bits into groups of four. If the number of bits is not divisible by four then add extra zeroes in front of the number. Remember that you can add zeroes to the left of the number without changing the value. You cannot add a zero to the right of the number because this would multiply the number by two.

To convert 100101011010001 from binary to hexadecimal, we divide it into groups of four. The leftmost group has only three bits so we add a zero in the front:

$$\underbrace{0100}_{4}\ \underbrace{1010}_{A}\ \underbrace{1101}_{D}\ \underbrace{0001}_{1}$$

It is easier to write 4AD1 than 100101011010001 and it is easy to convert between these two number systems. This is the reason why hexadecimal numbers are often used in digital systems.

It is convenient to use the hexadecimal representation as an intermediate if you want to convert from binary to decimal or from decimal to binary. The conversion: binary → hexadecimal → decimal is easier than binary → decimal. Likewise, the conversion: decimal → hexadecimal → binary is faster than decimal → binary because you need fewer divisions.

1.4. Addition of binary numbers

Addition of binary numbers goes the same way as for decimal numbers, as we learned at school. This example calculates 7 + 21, using binary numbers:

$$\begin{array}{rr} 111 & \\ 00111 & 7 \\ +10101 & +21 \\ \hline 11100 & 28 \end{array}$$

In the example above, we start with the 1's place which is the rightmost column. 1 + 1 = 2. The number 2 in binary is 10, so we get zero and a carry which goes to the 2's place. The carries are indicated in red here. The second column from the right is the 2's place. Here we have 1+1+0 = 2. This gives a zero in the 2's place and a carry to the 4's place. The third column from the right is the 4's place. Here we have 1+1+1 = 3. The binary code for 3 is 11, so we get 1 in the 4's place and a carry to the 8's place. There are no more carries, so the result is 00111 + 10101 = 11100, or in decimal: 7 + 21 = 28.

1.5. Signed binary numbers

There are several different ways of representing negative numbers. Today, almost all computers use a system called *two's complement* for representing numbers that can be both positive and negative. We will explain this system shortly.

A digital system typically has a fixed number of bits to represent a binary number. For example, if we have four bits, we can have the numbers from 0 to 15:

binary	decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

We can add 1 by going one step down in this table. The maximum number we can have with four bits is 15. If we try to add 1 to 15 we get the binary number 10000. Since we have only four bits in this example, we will not see the extra bit, but only 0000. In other words, if we add 1 to 15 in a four-bit system we get an overflow and the result will be 0. Likewise, if we try to subtract 1 from 0 we get an underflow and we will see the result 15.

This behavior is illustrated in the wheel on figure 1.1. You add one by going one step clockwise. When you pass 1111 (15) you will see the number overflow back to zero. The binary numbers are shown in black. The unsigned decimal numbers are shown in blue.

The idea of the two's complement system is that we use the same number wheel with a fixed number of bits and ignore overflow and underflow. We can find the representation of -1 by subtracting 1 from 0. This will underflow and give the bits 1111. Now, we define that 1111 means -1 instead of 15. We can find -2 by subtracting 1 from -1. This gives 1110 as the representation for -2. We will use the left half of the circle for negative numbers and the right half for positive numbers. We want to make it easy to distinguish between positive and negative numbers, so we decide that the leftmost bit is 1 for negative numbers and 0 for positive numbers. This bit is called the *sign bit*. Now, the positive numbers go from 1 to 7, and the negative numbers go from -1 to -8. There are eight negative numbers but only seven positive numbers because the value zero also has the sign bit set to 0. The signed numbers are shown in red on the figure.

The same bit pattern can be interpreted in two different ways now. The bit pattern 1111 means 15 if we interpret it as an unsigned number (blue numbers on the figure), but it means -1 if we interpret this bit pattern as a signed number (red numbers on the figure). The values from 0000 to 0111 are the same whether you use signed or unsigned numbers, while the values from 1000 to 1111 are interpreted differently for signed and unsigned numbers. Some computer programming languages, such as C and C++, allow you to specify whether a binary variable is interpreted as a signed or an unsigned number.

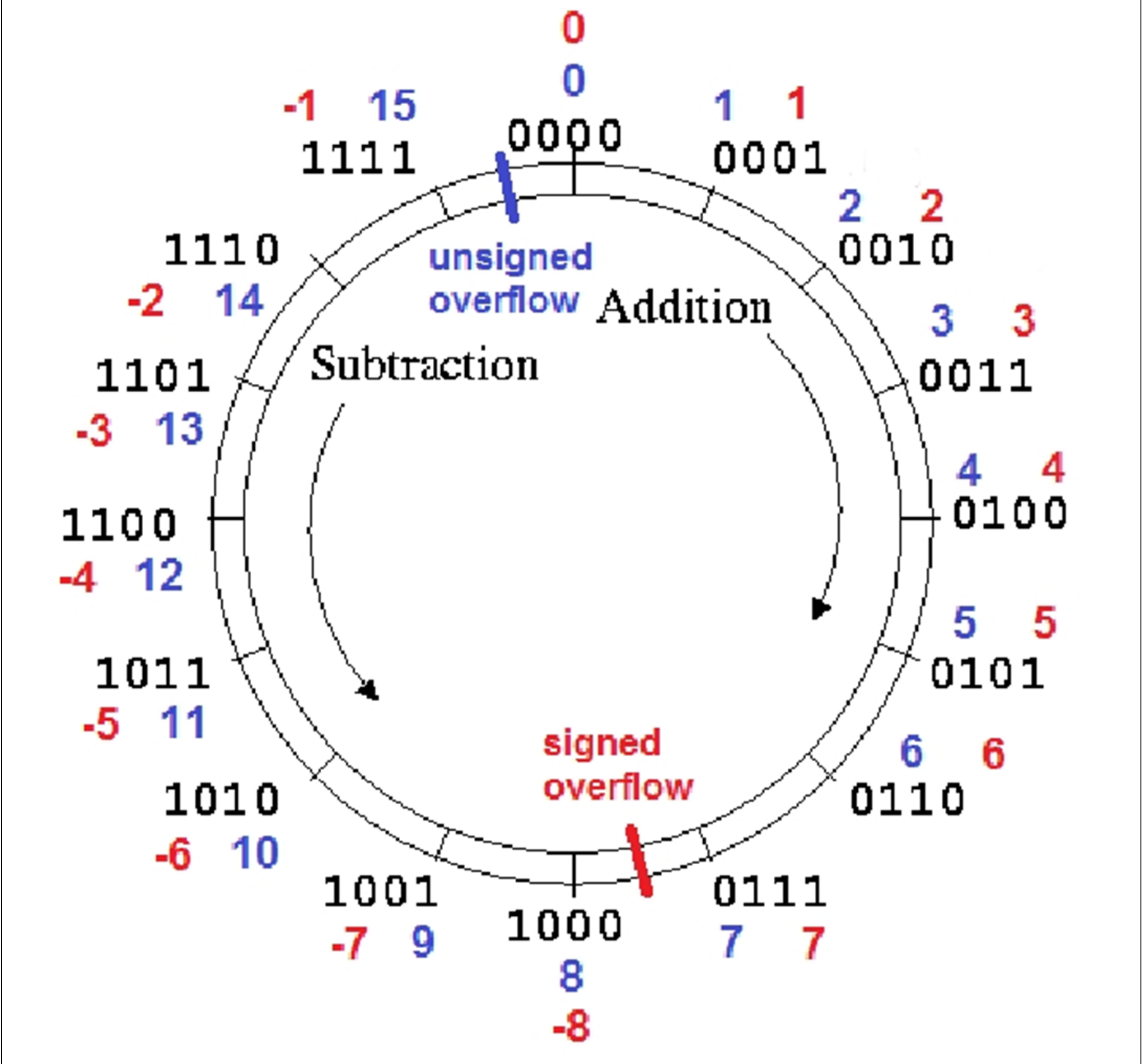


Fig. 1.1. Two's complement number system

1.5.1. How to change the sign of a number
You cannot change the sign of a number just by inverting the sign bit. The rule is that you change the sign of a number by inverting all bits and then add 1. For example, if you want to find the two's complement representation of -5, you first find the binary representation of 5. Then invert all bits and add 1:

0101	Binary representation of 5
1010	Invert all bits
1011	Add 1. This is the representation of -5

We can also use this rule to find the value of a bit pattern, for example 1001. The sign bit is 1 so it must be a negative number. We want to change the sign in order to find the corresponding positive number:

1001	We want to find out what this signed bit pattern means
0110	Invert all bits
0111	Add 1. This means 7. Therefore, 1001 means -7

1.5.2. The ranges of signed and unsigned n-bit numbers
In the above example we used only four bits for the sake of simplicity. This gave us a quite limited range of possible values. Four bits gives us 24 = 16 different bit combinations, from 0000 to 1111. We can use these sixteen different bit combinations to represent either the unsigned numbers from 0 to 15, or the signed numbers from -8 to +7. If we want higher numbers, then we need more bits.

If we have eight bits then we have 28 = 256 different bit combinations, from 00000000 to 11111111. We can use these 256 different bit combinations to represent either the unsigned numbers from 0 to 255, or the signed numbers from -128 to +127.

In general, if we have *n* bits then we have 2n different bit combinations. We can use these 2n different bit combinations to represent either the unsigned numbers from 0 to 2n-1, or the signed numbers from -2n-1 to +2n-1-1.

Modern computers typically use 8, 16, 32, or 64 bits for representing integer numbers. We can calculate the ranges using these formulas.

number of bits	**range for unsigned numbers**	**range for signed numbers**
8	0 ... 255	-128 ... +127
16	0 ... 65535	-32768 ... +32767
32	0 ... 4294967295	-2147483648 ... +2147483647
64	0 ... $1.8 \cdot 10^{19}$	$-9.2 \cdot 10^{18}$ ... $+9.2 \cdot 10^{18}$
n	0 ... $2^{n}-1$	-2^{n-1} ... $+2^{n-1}-1$

1.6. Binary coded decimal numbers
We prefer to use binary numbers in digital applications, but sometimes we have to use decimal numbers in order to make the numbers easier to read for humans. For example, we may want a decimal number on a display, or a human operator may enter a decimal number on a keyboard. We can represent a decimal number in a digital system by using four bits for each decimal digit. Four bits

gives us sixteen different combinations which is more than enough to represent the possible digits from 0 to 9. We are using only ten of the sixteen possible bit combinations. This method is called binary coded decimal numbers (BCD).

If you need, for example, a display that can show the numbers from 000 to 999 then you need three groups of four wires each for the three digits. For example, the number 256 in binary code is 100000000, while 256 in binary coded decimal is 0010 : 0101 : 0110.

1.7. Exercises

Exercise 1.1.

The powers of 2 are used everywhere in digital systems. Write a table of the powers of 2 from 20 to 210 in decimal, hexadecimal, and binary representation.

Exercise 1.2.

Convert these numbers from binary to decimal:
1111
1100100

Exercise 1.3.

Convert these numbers from decimal to binary:
71
1023

Exercise 1.4.

Convert these numbers from binary to hexadecimal:
100100011
1111000000001101

Exercise 1.5.

Convert the numbers in exercise 1.4 to decimal. Tip: It is easier to convert from hexadecimal to decimal than from binary to decimal.

Exercise 1.6.

Convert these numbers from hexadecimal to binary:
2468
ABCD

Exercise 1.7.

Find the sum of the numbers in exercise 1.4 by binary or hexadecimal addition.

Exercise 1.8.

How many different binary numbers can you write with:

4 bits
5 bits
n bits

Exercise 1.9.

Write the number -4 in two's complement representation for a binary system with 16 bits.

Exercise 1.10.

We want to build a digital thermometer that can show the temperature up to 200 °C without decimals.

How many bits do we need to represent the temperature as a binary number if only positive temperatures can be shown?

How many bits do we need to represent the temperature if the thermometer can also show negative temperatures, and the two's complement representation is used?

How many bits do we need to represent the temperature in binary coded decimal (BCD) representation?

2. Boolean algebra

Boolean algebra is a branch of mathematics where variables can have only two possible values: false and true, or 0 and 1. The basic operations in Boolean algebra are AND, OR, and NOT.

These operators are defined in the following tables.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

The AND operator gives 1 if both inputs are 1. The OR operator gives 1 if at least one input is 1. The NOT operator gives the opposite of the input.

Different people use different symbols for these operators. Mathematicians use the symbols ∧ ∨ ¬ for AND, OR, NOT. Software programmers use && || ! in programming languages like C, Java, etc. Engineers often write AND as multiplication, OR as addition, and NOT as an overbar ($\overline{A}$).

Operator	Mathematics	Software	Engineering
AND	∧	&&	A*B
OR	∨	||	A+B
NOT	¬	!	$\overline{A}$

It is clear that Boolean AND is the same as multiplication if you look at the table for the AND operation. It is less obvious why engineers write A + B when they mean A OR B. The tables for OR and PLUS are slightly different:

A	B	A ∨ B	A+B
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	2

The reason why it is convenient to use the multiplication and addition signs for AND and OR is that the rules for multiplication and addition that we are used to from elementary algebra also apply to AND and OR in Boolean algebra, as we will see. We just have to remember that 1 + 1 = 1 when we are dealing with Boolean algebra.

The normal rules for the precedence of operators apply. Multiplication comes before addition if there is no parenthesis:

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

This rule also applies if you use the other symbols ∧ ∨ or && ||.

2.1. Laws and rules

I am sure you have learned the basic rules of elementary algebra in school, even if you do not know the names of these rules:

Name of law	Boolean AND	Boolean OR
Commutative law	A*B=B*A	A+B=B+A
Associative law	A * (B * C) = (A * B) * C	A + (B + C) = (A + B) + C
Distributive law	A * (B + C) = (A * B) + (A * C)	A + (B * C) = (A + B) * (A + C)

These laws are the same for elementary algebra and Boolean algebra, except the last one:

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

The latter law is valid for Boolean algebra, but not for elementary algebra.

There are many other useful rules in Boolean algebra:

A*0=0	A+1=1
A*1=A	A+0=A
A*A=A	A+A=A
$A * \overline{A} = 0$	$A + \overline{A} = 1$
$A + \overline{A}*B = A + B$	$A * (\overline{A} + B) = A * B$
$\overline{\overline{A}} = A$	

These rules are easy to prove by inserting all possible values on the left hand side of the equation sign and see if you get the same values on the right hand side.

One rule is particularly good to remember. It is called *De Morgan's rule*:

$\overline{A} * \overline{B} =$ [illegible]	$\overline{A} + \overline{B} =$ [illegible]

De Morgan's rule can be expressed more generally for any Boolean expression: You can invert the output of a Boolean expression by inverting all the inputs, replace all AND operations by OR, and replace all OR operations by AND. This rule is often used for simplifying digital circuits.

All the tables above have two columns. The rules in the left column can be derived from the rules in the right column, or vice versa, by applying de Morgan's rule. Let's try this for the commutative law:

A*B=B*A

Define X = $\overline{A}$ and Y = $\overline{B}$, and insert:

$\overline{X} * \overline{Y} = \overline{Y} * \overline{X}$

Use de Morgen's rule on both sides:

[illegible]

Invert on both sides

X+Y=Y+X

This is the result we want, just with different letters.

2.2. Truth tables

A truth table is a table of the value of a Boolean expression for all possible values of the inputs. For example, this is a table for the expression F = A * (B+C).

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

Sometimes, you want to build a digital circuit with a given functionality that is defined only by a truth table. You can use the so-called *sum-of-products* method to find a Boolean expression that corresponds to a given truth table. This method works as follows:

Find all the lines in the truth table for which the output is 1. Make one expression for each of these lines by AND'ing all the inputs and inverting those inputs that are 0 in that line. Each of these expressions is 1 in the corresponding line and 0 in the rest. The final result is the OR-combination of all these expressions. Let's try this method with the example above.

A	B	C	F = A * (B+C)	sum of products expression
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	$A*\overline{B}*C$
1	1	0	1	$A*B*\overline{C}$
1	1	1	1	A*B*C

Now we know that F = $A*\overline{B}*C + A*B*\overline{C} + A*B*C$.

This is called a *sum of products*, even though the * and + actually mean AND and OR.

Now we have a valid expression for F, but not the simplest possible one. We may simplify this expression by using the laws and rules that we have learnt:

$A*\overline{B}*C + A*B*\overline{C} + A*B*C =$

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

$A*\overline{B}*C + A*B*1 =$

$A*\overline{B}*C + A*B =$

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

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

$A * (B + C)$

If there are many 1's and few 0's in the output column, then it is easier to invert the output and apply the sum-of-products method to the inverted output. This gives a simpler expression, but we must remember to invert the result of the expression. We can use de Morgan's rule for converting the inverted sum of products to a "product of sums".

2.3. Reducing a Boolean expression

In the above example, we could reduce the expression by looking for common factors that we could put outside of a parenthesis. This may be hard, but at least you can do it in simple cases with a little practice. Unfortunately, some cases are so tricky that you basically have to know the result in advance in order to find a way to the result. I will give you one example here:

$G = A*B + \overline{A}*C + B*C$

It is not obvious that this expression can be reduced, but look what happens when we use the rule $A + \overline{A} = 1$:

G=
$A*B + \overline{A}*C + B*C =$

$A*B + \overline{A}*C + 1*B*C =$

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

$A*B + \overline{A}*C + A*B*C + \overline{A}*B*C =$

$A*B + A*B*C + \overline{A}*C + \overline{A}*C*B =$

$A*B*(1+C) + \overline{A}*C*(1+B) =$

$A*B*1 + \overline{A}*C*1 =$

$A*B + \overline{A}*C$

Pure magic! The term B*C just disappeared. This is not the way forward if we want to reduce Boolean expressions that may be more complicated than this example. There is a graphical method that makes this kind of reductions more intuitive. It is called a *Karnaugh map* (pronounced in French: Kar-nó map). The Karnaugh map shows geometrically that all input combinations covered by the term B*C in the above example are already covered by the two terms A*B and $\overline{A}$*C.

I will not teach you how to make a Karnaugh map because it takes some time and practice to learn, and it becomes quite difficult if there are more than four inputs. You may look up "Karnaugh map" on the web if you are interested. However, we have software that does the complicated job of finding the simplest possible expression for a given Boolean function. There is an online program at www.32x8.com that does the job. There is also an open source program called *Karnaugh Map Minimizer* that you can download from https://sourceforge.net/projects/k-map/.

Today, complicated digital circuits are designed with the use of a *hardware description language*, such as VHDL or Verilog. There is not much need for learning how to make Karnaugh maps nowadays because the reduction of Boolean expressions comes automatically when you use such development tools.

2.4. Exercises

Exercise 2.1.

The distributive law for Boolean algebra looks like this:

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

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

Prove this by using truth tables.

Exercise 2.2.

Use the rules of Boolean algebra to reduce these expressions to the simplest possible:

(1) $A*B*C + \overline{A}*B + A*B*\overline{C}$

(2) $\overline{X}*Y*Z + X*Z$

(3) $(X + Y) * (\overline{X} + \overline{Y})$

(4) $X*Y + X*(W*Z + W*\overline{Z})$

(5) $(B*\overline{C} + \overline{A}*D) * (A*\overline{B} + C*\overline{D})$

(6) $(X + \overline{Y} + \overline{Z}) * (\overline{X} + \overline{Z})$

Exercise 2.3.

We want to implement a Boolean function F with the following truth table

Inputs				output
A	**B**	**C**	**D**	**F**
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
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	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

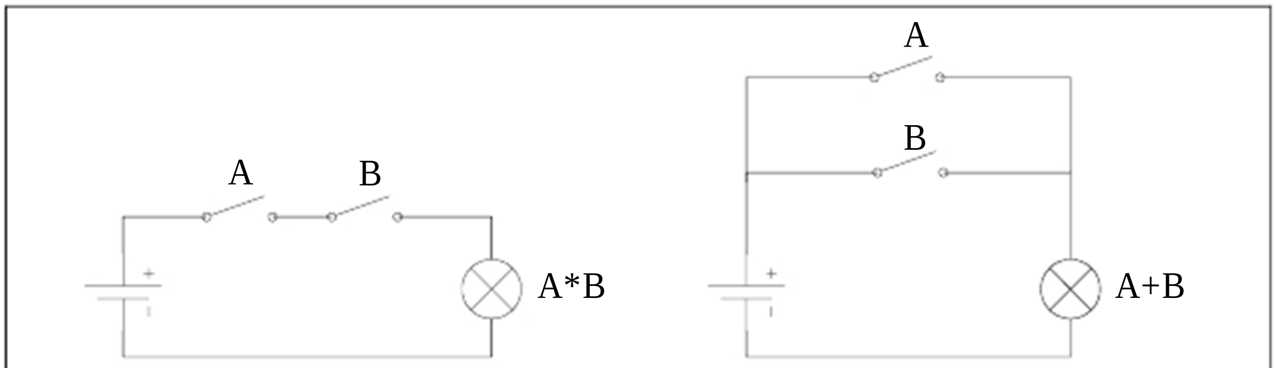
Write the expression for F as a sum of products.

Reduce the expression for F to the simplest possible by using the online program at www.32x8.com or the program "Karnaugh Map Minimizer" from https://sourceforge.net/projects/k-map.

3. Digital circuits

3.1. How Boolean gates are made

The Boolean operations AND and OR can be made with simple switches, as shown here:



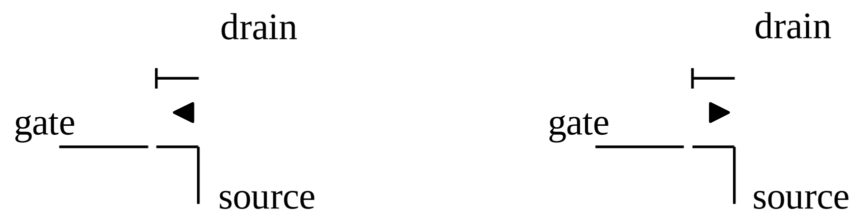
ANDfunction:
Thelampwill light whenAandBare bothon.

ORfunction:
Thelampwill light whenat least one of AandBis on.

Fig. 3.1. Boolean functions implemented with switches

Two switches connected in series will produce the AND function. Two switches connected in parallel will produce the OR function.

This principle is used in digital circuits, where the switches are replaced by transistors. MOSFET transistors are often used for this. The symbols for two kinds of MOSFET transistors are shown here:



N-channel MOSFET

P-channel MOSFET

Fig. 3.2. N-channel and P-channel MOSFETs

A MOSFET has three connections named *drain*, *source*, and *gate*. There is no current going through the gate, but the voltage on the gate controls the current that can go between drain and source. We can use this as a switch that can be turned on and off by changing the voltage on the gate. The N-channel MOSFET is turned on by a positive voltage on the gate relative to the source. The P-channel MOSFET is turned on by a negative voltage on the gate relative to the source.

The simplest digital circuit we can make of MOSFET transistors is an inverter. Our inverter is made of
a p-channel and an n-channel MOSFET:

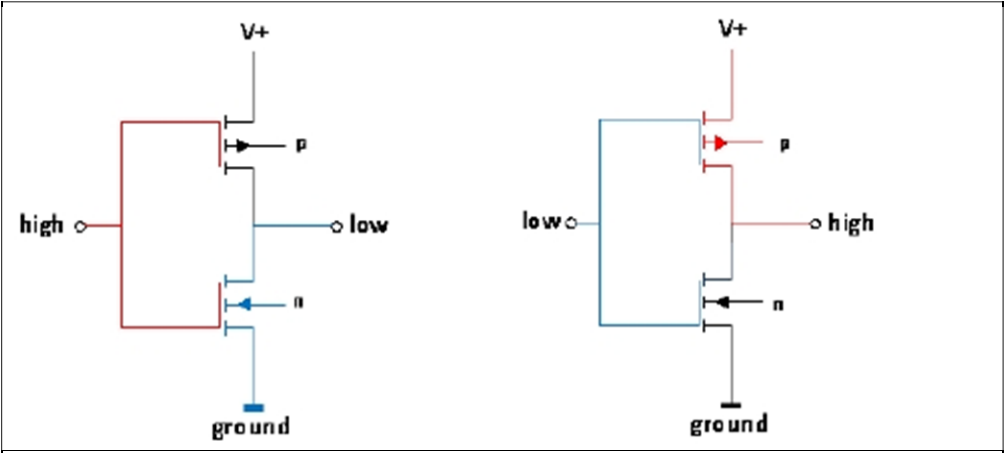


Fig. 3.3. CMOS inverter

If the input is high (for example 5V) then the p-channel MOSFET is off and the n-channel MOSFET is
on. The n-channel MOSFET connects the output to ground to make it low.

If the input is low (0V) then the p-channel MOSFET if on and the n-channel MOSFET is off. The p-channel MOSFET connects the output to the positive supply (5V) to make it high.

The circuit in figure 3.3 can be used as an inverter because a high input gives a low output, and a low input gives a high output.

Now, we can make logical circuits by connecting the MOSFET switches in series or parallel. Such a circuit is called a *logical gate*. Figure 3.4 shows an OR gate with inverted output, also called a NOR gate.

If the inputs A and B are both low, then the n-channel MOSFETs will be off and the p-channel MOSFETs will be on. The output is connected to the V+ through the p-channel MOSFETs so that the output is high.

If both inputs are high, then the n-channel MOSFETs will be on and the p-channel MOSFETs will be off. The output is connected to ground through the n-channel MOSFETs so that the output is low.

If one of the inputs is high and the other is low, then the output will be low because one of the n-channel MOSFETs is on, and these are connected in parallel to ground. The output has no connection to V+ in this case because one of the p-channel MOSFETs is off, and these are connected in series to V+.

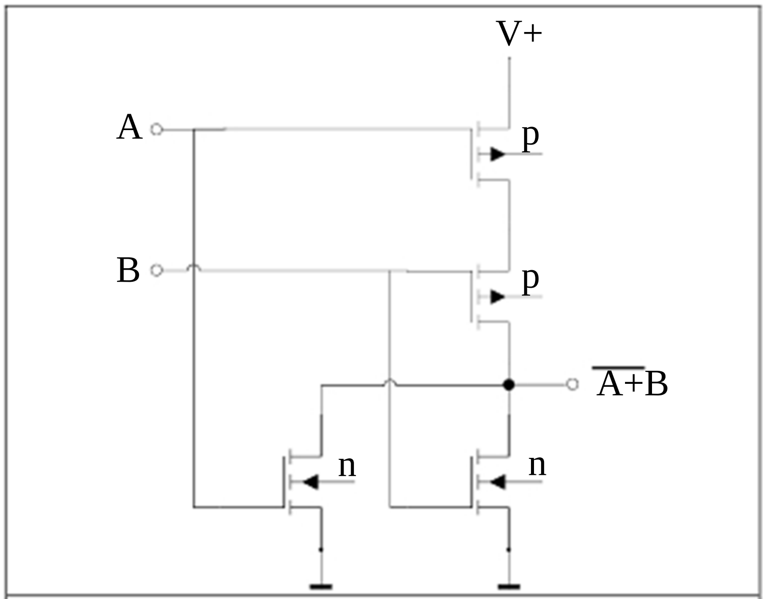


Fig. 3.4. CMOS NOR gate

The truth table for the NOR gate is:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

We can make an OR gate from this NOR gate by putting an inverter after the output.

Figure 3.5 shows an AND gate with inverted output, also called a NAND gate:

If the inputs A and B are both low, then the n-channel MOSFETs will be off and the p-channel MOSFETs will be on. The output is connected to the V+ through the p-channel MOSFETs so that the output is high.

If both inputs are high, then the n-channel MOSFETs will be on and the p-channel MOSFETs will be off. The output is connected to ground through the n-channel MOSFETs so that the output is low.

If one of the inputs is high and the other is low, then the output will be high because one of the p-channel MOSFETs is on, and these are connected in parallel to V+. The output has no connection to ground in this case because one of the n-channel MOSFETs is off, and these are connected in series to ground.

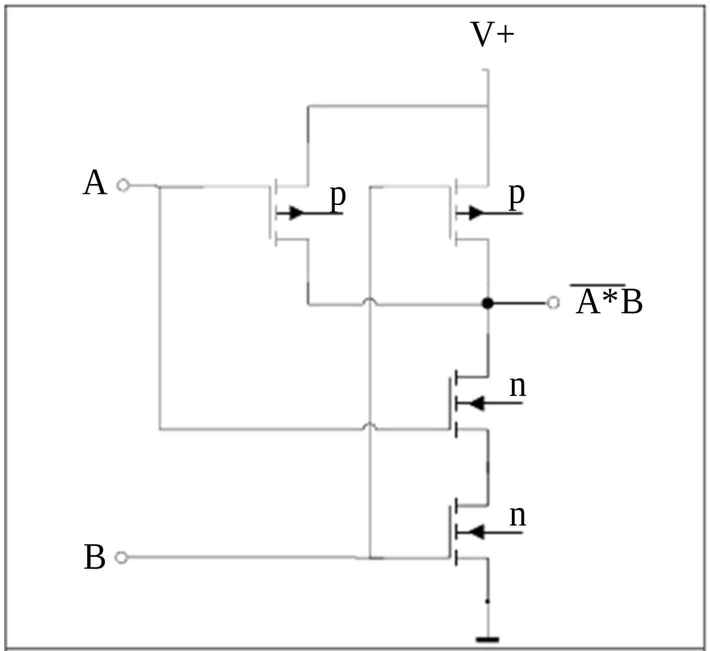


Fig. 3.5. CMOS NAND gate

The truth table for a NAND gate is:

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

We can make an AND gate from this NAND gate by putting an inverter after the output.

More complicated Boolean functions can be made by combining these circuits.

3.2. Gate symbols

The symbols for the different gates are shown in figure 3.6. An inverted connection is indicated by a bubble. The AND gate with inverted output is called a NAND gate. The OR gate with inverted output is called a NOR gate.

An exclusive-or gate, also called an XOR gate, has a high output if one – and only one – of the inputs is high. The XOR operator is sometimes written as a circled plus: ⊕

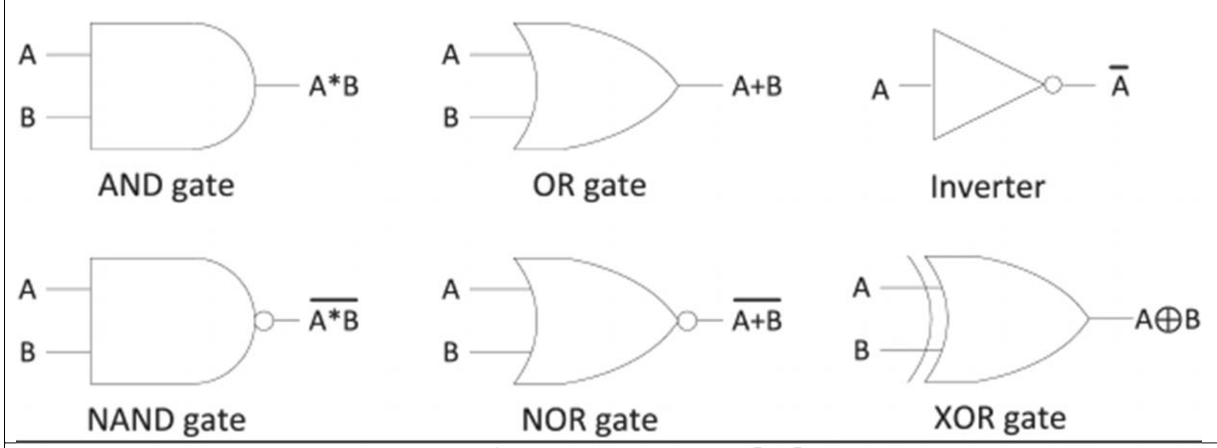


Fig. 3.6. Gate symbols

The truth tables for the different gates are as follows:

A	B	A*B	A+B	$\overline{A*B}$	$\overline{A+B}$	A⊕B
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

You can combine these gates to make any Boolean function. Figure 3.7 shows an example. Usually, we prefer to draw the inputs to the left and the outputs to the right. It is not necessary to draw the power supply to all the gates because this will only make the diagram more complicated without adding any important information.

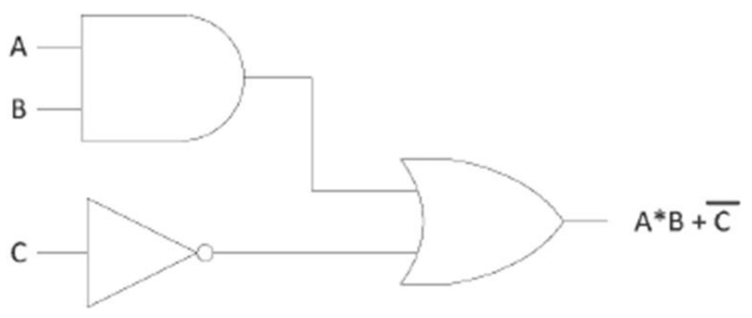


Fig. 3.7. Boolean function implemented with gates

All the different gates are available as integrated circuits. For example, the integrated circuit 74HC00 contains four NAND gates in an integrated circuit with 14 pins as shown in figure 3.8. Vcc is the positive supply voltage (+5V), and GND is ground (0V).

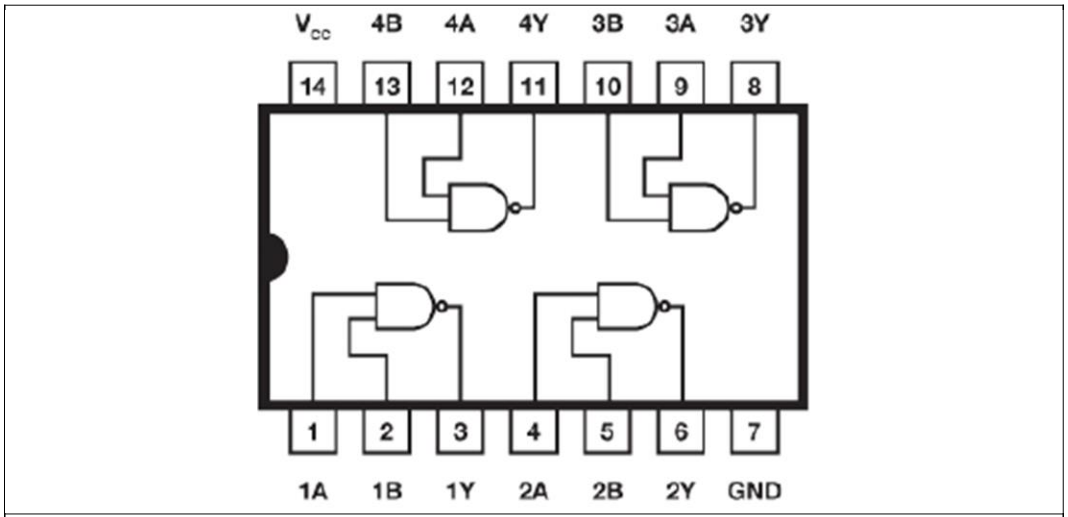


Fig. 3.8. Pin configuration of 74HC00 NAND gate

Many different integrated circuits are available with different digital functions. You can connect these together to make more complex circuits. An output can be connected to more than one input, but an input cannot be connected to more than one output, because you will have a short circuit if one output is high and the other output is low. All inputs must be connected to either an output, ground, or to the positive supply.

The inputs of MOSFET circuits are controlled by the voltage on the input, not the current. The current that goes through an input is negligible (< 1 nA). Therefore, it is necessary that all inputs are connected to something with a known voltage. An input that is not connected to anything will be *floating*. This
means that it has a random and unpredictable voltage. Even the smallest electromagnetic noise can make a floating input change erratically.

MOSFET circuits should be handled with care because they are easily destroyed by electrostatic charges, for example if you touch them after walking on a synthetic carpet.

In the next chapter, we will look at some circuits we can build with the different gates.

3.1. Exercises

Exercise 3.1.

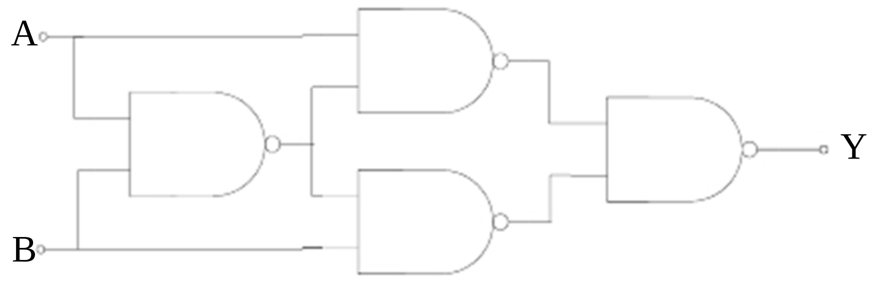
Draw a diagram with gates to implement the function $F = A * B + \overline{A} * C$.

Exercise 3.2.

Change the diagram from exercise 3.1 so that it uses only NAND gates. Tip: Use De Morgan's rule to convert the OR to an AND.

The integrated circuit 74HC00 contains four NAND gates. How many 74HC00 chips do you need to implement this function?

Exercise 3.3.

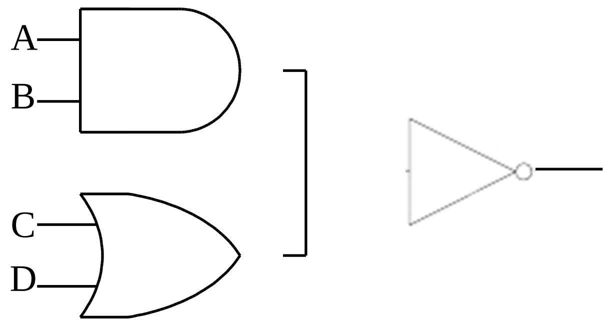


Write the truth table for this function.

Is there a simpler way to make the same function if all types of gates are available?

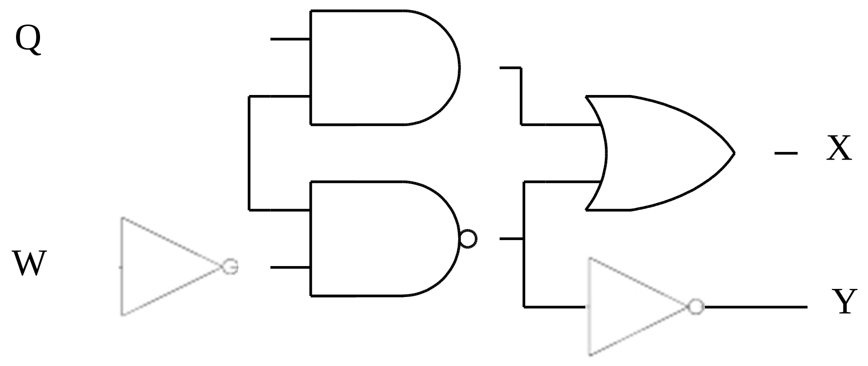
Exercise 3.4.

What is wrong with this circuit? What will happen?



Exercise 3.5.

What is wrong with this circuit? What will happen?



4. Commonly used Boolean circuits

4.1. Decoders

A decoder is a circuit that converts a binary number to some other code that requires more wires. A very common kind of decoder has one output for each possible combination of the inputs. For example, if you have three inputs then you have eight possible input combinations corresponding to the binary numbers from 0 to 7. We want one output for each of these eight combinations. The truth table looks like this:

A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

We can implement this truth table with AND gates and inverters. For example,
$Y2 = A\bar{0} * A1 * A2$.

A diagram for the three-to-eight decoder is shown on figure 4.1.

You can have decoders of any size. If the decoder has *n* inputs then it can have 2n outputs.

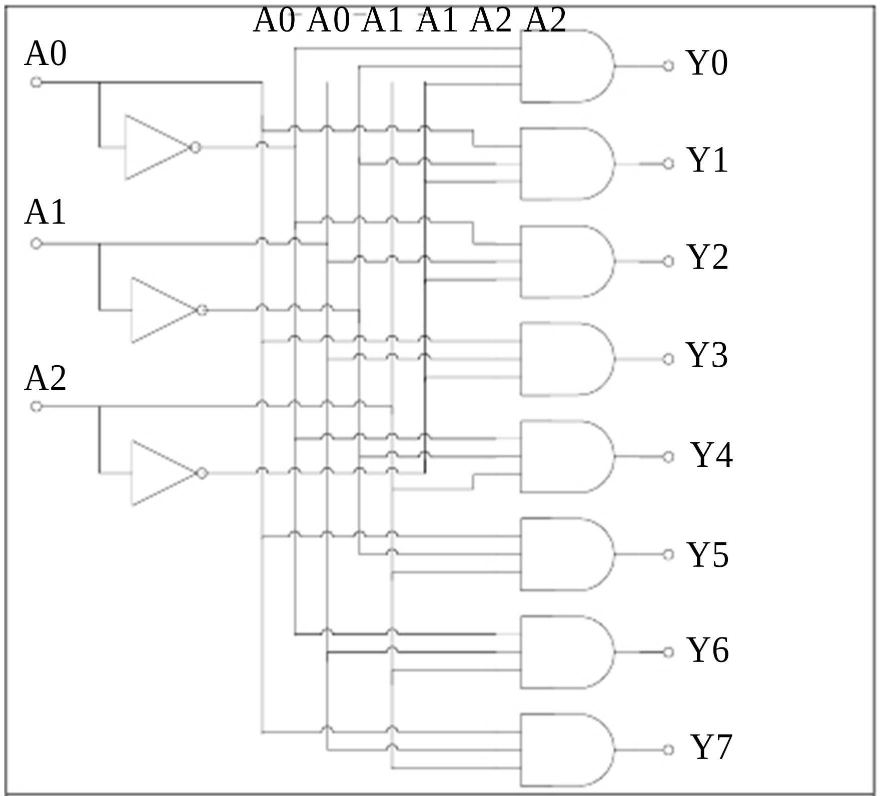


Fig. 4.1. Three to eight decoder

4.2. Encoders

An encoder is the opposite of a decoder. It has more inputs than outputs. For example, if you have a keyboard with many keys and the user is pressing one key, you want the key number as a binary code.

The truth table for an encoder with eight inputs may look like this:

A0	A1	A2	A3	A4	A5	A6	A7	Y2	Y1	Y0	V
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
x	1	0	0	0	0	0	0	0	0	1	1
x	x	1	0	0	0	0	0	0	1	0	1
x	x	x	1	0	0	0	0	0	1	1	1
x	x	x	x	1	0	0	0	1	0	0	1
x	x	x	x	x	1	0	0	1	0	1	1
x	x	x	x	x	x	1	0	1	1	0	1
x	x	x	x	x	x	x	1	1	1	1	1

A complete truth table with eight inputs would need 28 = 256 lines in the table. We have joined some of the lines together to save space by writing x, meaning *don't care*. The output is the same regardless of the values of the x inputs.

As you can see from the table above, the output (Y2,Y1,Y0) indicates the highest input that is active. The extra output V (valid) is used for distinguishing between no input and input A0.

4.3. Seven segment decoders

A seven-segment decoder is used for showing numbers on a seven-segment display.

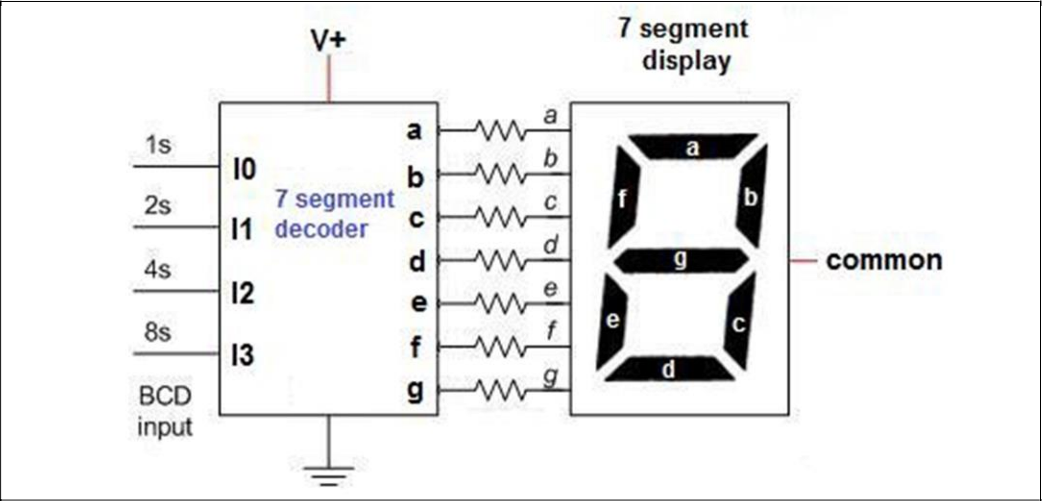
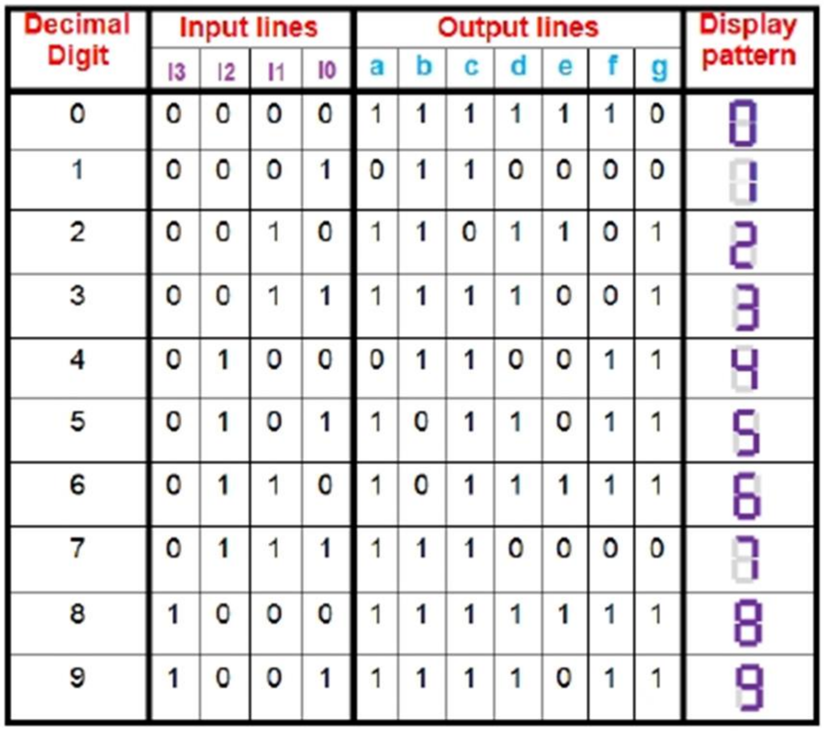


Fig. 4.2. Use of 7-segment decoder

The truth table looks like this:

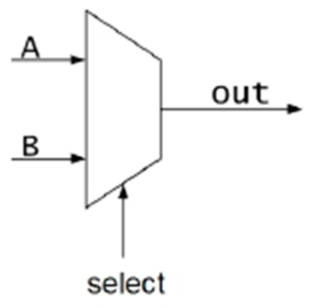


Decimal Digit	Input lines				Output lines							Display pattern
	I3	I2	I1	I0	a	b	c	d	e	f	g	
0	0	0	0	0	1	1	1	1	1	1	0	0
1	0	0	0	1	0	1	1	0	0	0	0	1
2	0	0	1	0	1	1	0	1	1	0	1	2
3	0	0	1	1	1	1	1	1	0	0	1	3
4	0	1	0	0	0	1	1	0	0	1	1	4
5	0	1	0	1	1	0	1	1	0	1	1	5
6	0	1	1	0	1	0	1	1	1	1	1	6
7	0	1	1	1	1	1	1	0	0	0	0	7
8	1	0	0	0	1	1	1	1	1	1	1	8
9	1	0	0	1	1	1	1	1	0	1	1	9

Input values above 9 do not necessarily produce any meaningful patterns on the display.

4.4. Multiplexers

A multiplexer is a kind of switch that chooses between two or more inputs. The symbol and the truth table for a two-input multiplexer looks like this:



select	A	B	out
0	x	x	A
1	x	x	B

Fig. 4.3. Two-to-one multiplexer

The multiplexer can be implemented as shown in figure 4.4:

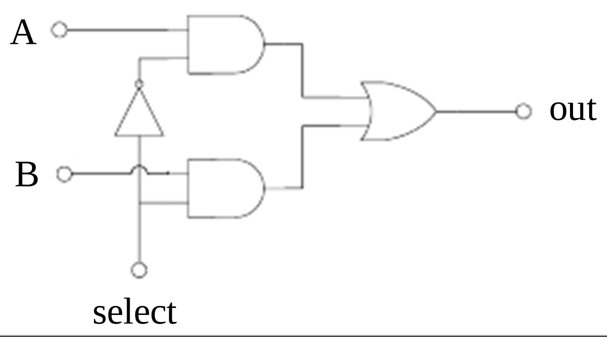
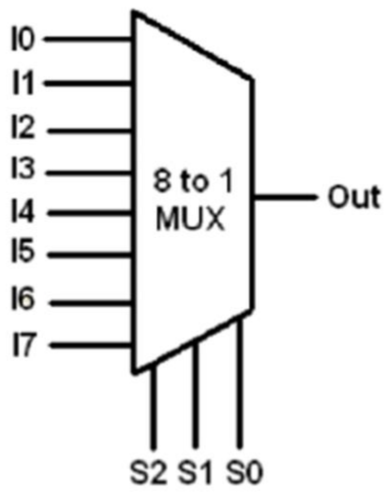


Fig. 4.4. Two-to-one multiplexer implementation

An eight-input multiplexer:

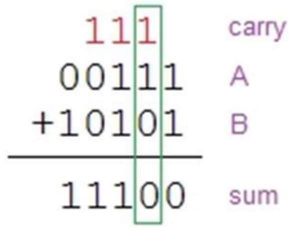


S2	S1	S0	I0-I7	out
0	0	0	x	I0
0	0	1	x	I1
0	1	0	x	I2
0	1	1	x	I3
1	0	0	x	I4
1	0	1	x	I5
1	1	0	x	I6
1	1	1	x	I7

Fig. 4.5. Eight-to-one multiplexer

4.5. Adders

We learned on page 6 how to add binary numbers by hand. Now we will construct a logical circuit that can do it for us. Let us look at the same example again:



We can divide this calculation into columns. The rightmost column, or column 0, is the 1's place. The second column from the right, or column 1, is the 2's place. Column 2 is the 4's place, and so on. Column *i* is the 2i's place.

Each column *i* has three inputs: carryi, Ai, Bi, and two outputs: sumi and the carry to the next column, carryi+1.

If we want to write the truth table for this, we simply add the three inputs carryi, Ai, Bi. The sum can be any number from 0 to 3. We write this sum as a binary number from 00 to 11 with the least significant bit in sumi and the most significant bit in the next carry, or carryi+1.

For example, the 2's place indicated by the green frame in the figure above has the inputs

carry1 + A1 + B1 = 1+1+0 = 2 = 10(base 2), where '+' means plus.

The binary sum is 10. These two bits are used so that the 0 goes to the sum in the 2's place: sum1 = 0. The 1 is the carry for the 4's place: carry2 = 1.

carry$_i$	**A$_i$**	**B$_i$**	**carry$_{i+1}$**	**sum$_i$**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The sumi output can be expressed as an exclusive-or of the three inputs:

sumi = Ai ⊕ Bi ⊕ carryi

The carry output is high if at least two of the inputs are high:

carryi+1 = Ai*Bi + Ai*carryi + Bi*carryi

We can build this with gates as shown in figure 4.6.

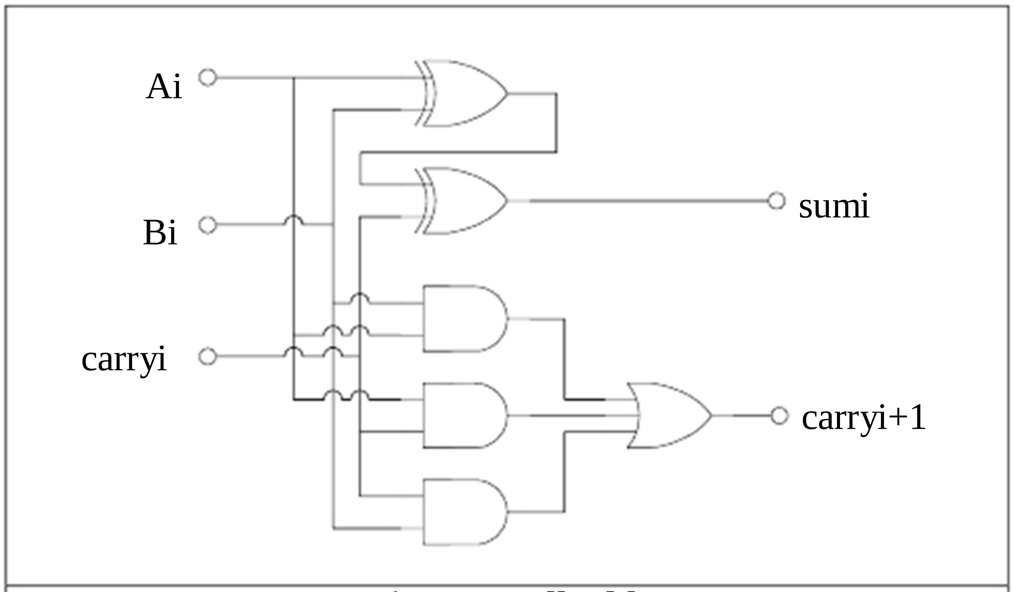


Fig. 4.6. Full adder

This circuit is called a full-adder. We need one full-adder for each column in our addition scheme. The carry output from each column goes to the carry input for the next column:

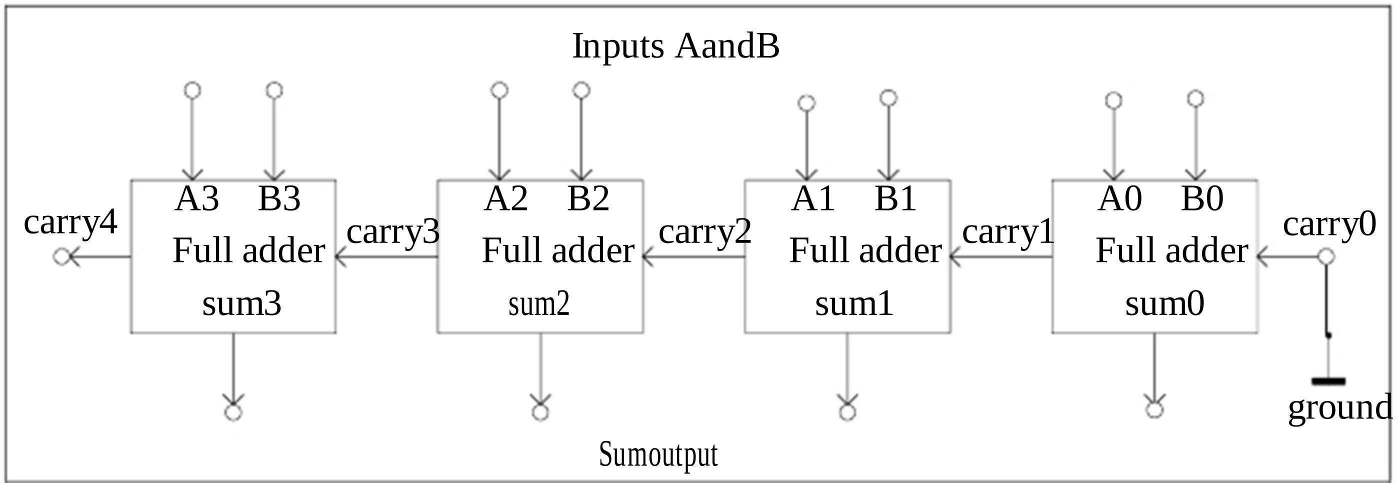


Fig. 4.7. Four bit adder

The carry input for the first full adder, carry0, is connected to ground so that it will be zero. The output
from the last full-adder, carry4, may be used for indicating overflow for unsigned addition.

4.6. Subtracting binary numbers

The four-bit adder that we made above corresponds to the number wheel on figure 1.1 (page 8). You add a value B by going B places clockwise on the wheel. This adder circuit also works for signed numbers in two's complement representation because the two's complement system relies on the same number wheel.

Now we want to subtract numbers. Remember how we changed the sign of a number? This is explained on page 9: invert all bits and add 1. So, we can calculate A - B as A + (-B), where

(-B) is generated by inverting all bits in B and adding 1. We can convert the adder in figure 4.7 to a subtractor by putting inverters on all the B input bits. We still have to add 1, and here the unused carry input comes in handy. We can simply add 1 by connecting the carry input to the positive voltage supply.

Figure 4.8 shows how we can use the adder as subtractor. This subtractor works for both unsigned numbers and for signed numbers in two's complement representation. The carry4 output indicates unsigned overflow.

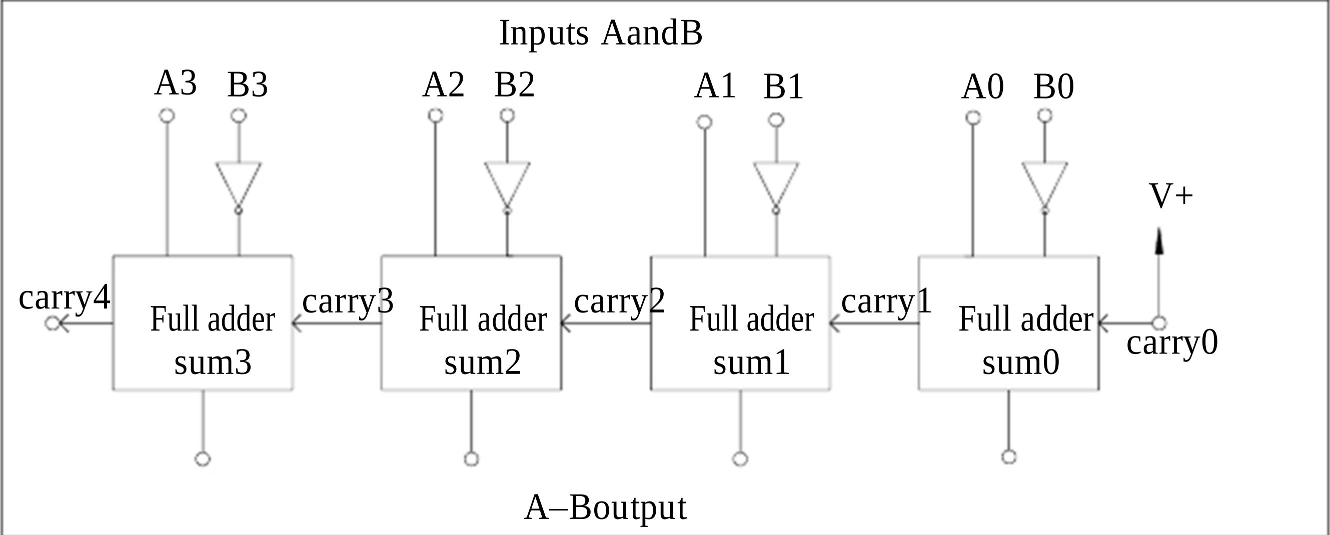


Fig. 4.8. Four bit subtractor

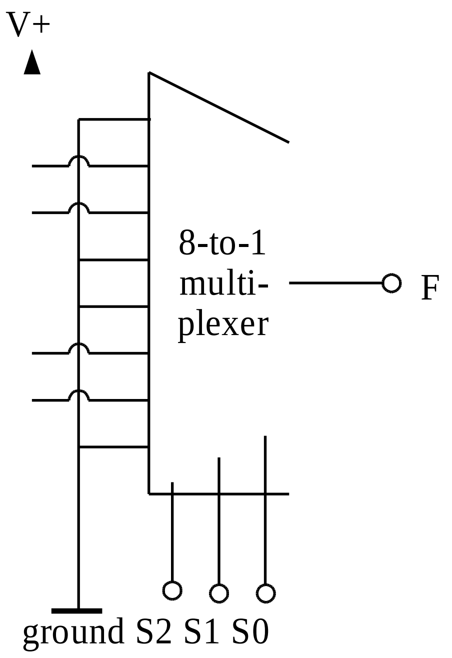
4.7. Exercises

Exercise 4.1.

Construct a two-to-four decoder. Draw the truth table and a diagram of gates.

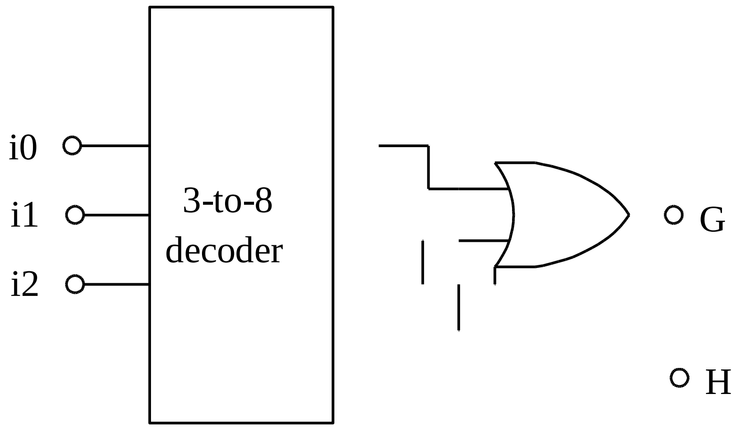
Exercise 4.2.

It is possible to construct any 3-input Boolean function by using an 8-to-1 multiplexer. Write the reduced Boolean equation for this multiplexer circuit:



Exercise 4.3.

It is possible to construct Boolean functions from decoders and OR gates. Write the reduced Boolean equations for this decoder circuit:



Exercise 4.4.

Test an adder on a breadboard. The integrated circuit 74HC283 contains a 4-bit adder like the one on figure 4.7.

You can use a DIP switch with at least 8 bits for the inputs. Each switch must have a pull-down resistor to ground in order to make sure the input is low when the switch is off:

Fig. 4.9. DIP switch and pull-down resistor

Connect light-emitting diodes (LED) to the outputs. The voltage over a LED is approximately 2V. The power supply is 5V, so we need a resistor in series with each LED to generate a voltage drop of 5V - 2V = 3V. A suitable current for a LED is 10 mA, so you can use a resistor of 3V / 10 mA = 300 Ω.

Test if the adder works as you expect when adding binary numbers.

Exercise 4.5.

Modify the adder from exercise 4.4 into a calculator that can both add and subtract. An additional input named SUB controls the function so that your circuit calculates A + B when SUB = 0, and A - B when SUB = 1.

Tip: you can use an XOR gate 74HC86 to invert the B inputs when SUB = 1 because $B \oplus SUB$ is equal to B when SUB = 0 and equal to $\overline{B}$ when SUB = 1.

SUB should also be connected to the carry input.

5. Flip-Flops

The output depends only on the inputs on digital circuits that have no memory. Such a circuit is called *combinational* because the output is a combination of the inputs. The circuits we have seen in the previous chapters were all combinational.

The output may depend not only on the inputs, but also on the past history, on circuits that contain memory elements. Such a circuit is called *sequential* because the output depends on a sequence of inputs. Now we will see how to create memory elements so that we can construct sequential circuits.

5.1. Basic feed back circuit

A flip-flop is a memory circuit that can remember one bit. It has two states, 0 and 1. Flip-flops are circuits with positive feedback. Consider this circuit built of two inverters:

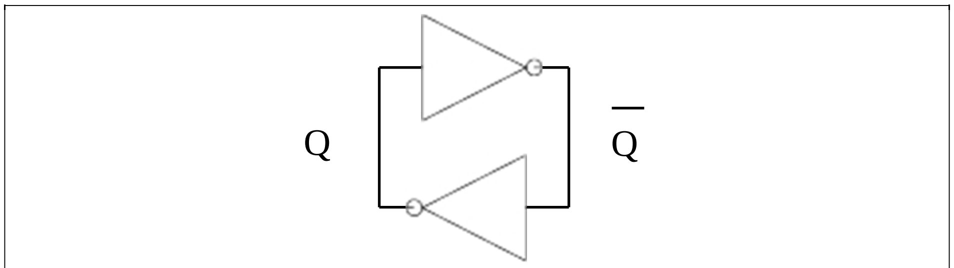


Fig. 5.1. Positive feedback circuit with two stable states

This circuit can be in one of two states. Either it is high on the left side (Q) and low on the right side ($\overline{Q}$), or vice versa. When you turn on the power, it will quickly go to one of these two stable states, because any state in between is unstable. You cannot predict which state it will end up in, because it is symmetrical.

5.2. SR flip-flop

The circuit above is not very useful because we cannot change the state. Now we will replace the inverters by NOR gates so that we can change the state. The input S is used for setting Q high. The input R is used for resetting Q to low.

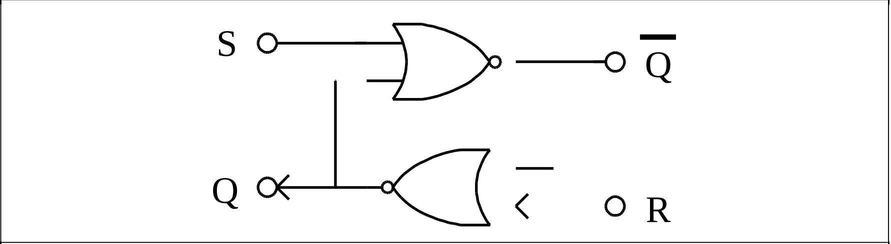


Fig. 5.2. SR flip-flop

We have the same feedback as before as long as the two inputs S and R are both low. Either Q will be low and $\overline{Q}$ will be high, or Q will be high and $\overline{Q}$ will be low. We can force $\overline{Q}$ to be low by setting S high because the output of a NOR gate is always low when at least one of the inputs is high. We assume that R is still low, so Q will be high. In other words, we can change the circuit to the state where Q is high by setting S high and low again. The opposite happens when we set R high and low again. This will make Q low and $\overline{Q}$ high.

This circuit is called an SR flip-flop. The S stands for set, and R stands for reset. We can write the truth table:

S	R	Q	$\overline{Q}$
0	0	remember last value	remember last value
0	1	0	1
1	0	1	0
1	1	0	0

The SR flip-flop works as a memory as long as the inputs S and R are both low. We can set it to state 1 (Q = 1) by setting S high and low again. And we can set it to state 0 (Q = 0) by setting R high and low again. It is not useful to set S and R both high at the same time.

Usually, we prefer to have inputs on the left and outputs on the right. We can rewrite the diagram of the SR flip-flop to get this:

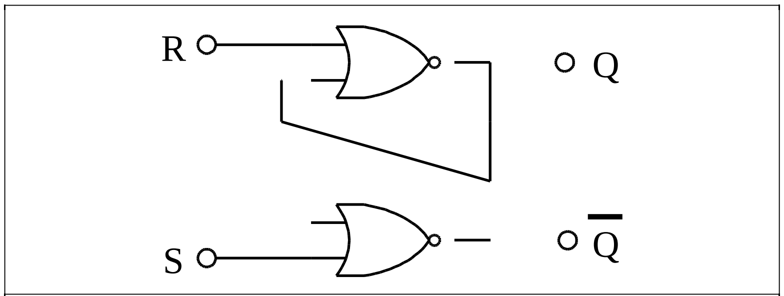


Fig. 5.3. SR flip-flop

5.3. SR flip-flop with enable

We can put an enable input (E) on the SR flip-flop so that it can only change when E is high:

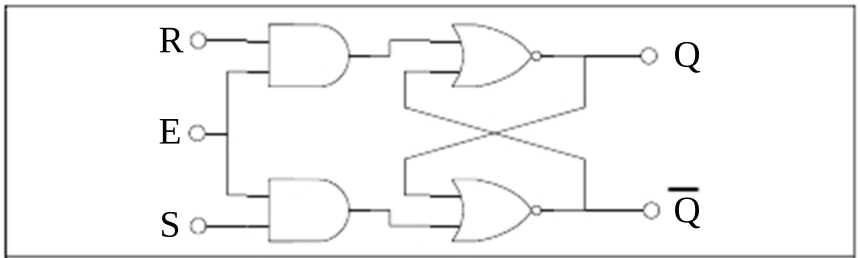


Fig. 5.4. SR flip-flop with enable

The truth table is now:

E	S	R	Q	$\overline{Q}$
0	x	x	remember last value	remember last value
1	0	0	remember last value	remember last value
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

5.4. D latch

Now, we can set R = $\overline{S}$ because R and S should never both be high at the same time. We can set the E input low when we want the circuit to remember a value.

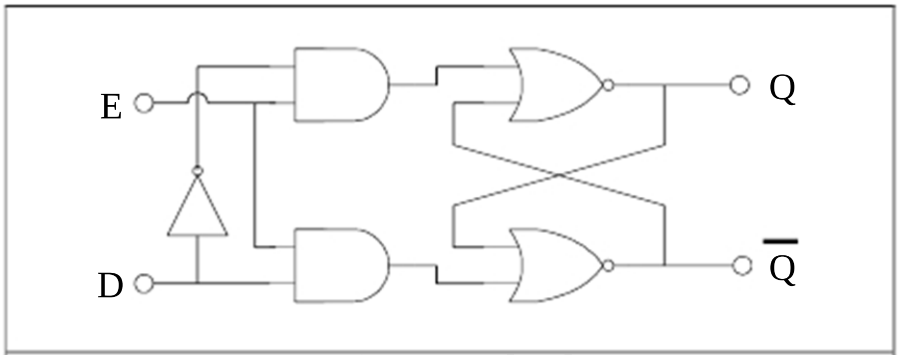


Fig. 5.5. D latch

The S input has been renamed to D (for data), while the R input is connected to $\overline{D}$. The state can only change when E is high. It will change to state 1 when D = 1, and state 0 when D = 0.

The D input is the data bit we want to save. The value of D is loaded into the D-latch when E is high and stored when E is low, as shown in this truth table:

E	D	Q	$\overline{Q}$
0	x	remember last value	remember last value
1	0	0	1
1	1	1	0

The D-latch is useful as a memory circuit.

5.5. Edge-triggered D flip-flop

Hold on! Now it is getting tricky. The D-latch can change as long as the E input is high. Now we want a memory circuit that can change only at specific points in time. This can be very useful when we want to handle data in a sequence, as we shall see below.

We can obtain this by connecting two D-latches after each other and invert the enable input of the first one. The first latch is open when the enable input is low and the second latch is open when the enable input is high:

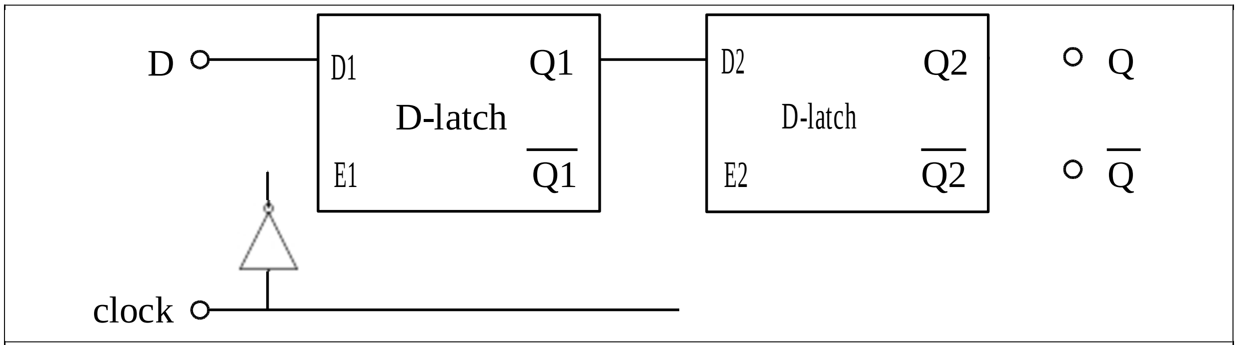


Fig. 5.6. Edge-triggered D flip-flop

The D input contains the data bit that we want to save. The joined enable input is named clock. First, the clock input is low. This will make E1 high and E2 low, so that D is loaded into the first latch, while
the second latch is on hold. Now we let the clock input go high. This will make E1 low and E2 high. Now the first latch is on hold while the second latch is loading the value of Q1. This will be the value that the D input had immediately before the clock input went high. When the clock goes low again, the
second latch will be on hold so that the same value will be preserved until the clock goes high again.

The result is that the Q output will remember the value that the D input had at the time when the clock changed from low to high. This is called an *edge-triggered* flip-flop. We say that the value of D is saved on the *rising edge* of the clock input.

Let me explain this with an example. The timing diagram in figure 5.7 shows how the signals change in a time sequence. We are assuming that Q1 and Q2 are zero when we start.

The clock input goes up and down all the time. The first latch, Q1, follows the D input when the clock is low (green) and stays constant when the clock is high (blue). The second latch, Q2, follows Q1 when the clock is high (blue) and stays constant when the clock is low (green). The result is that Q2 can change only when clock goes from low to high. The output value that we see on Q2 is the value that the D input had immediately before the clock went from low to high.

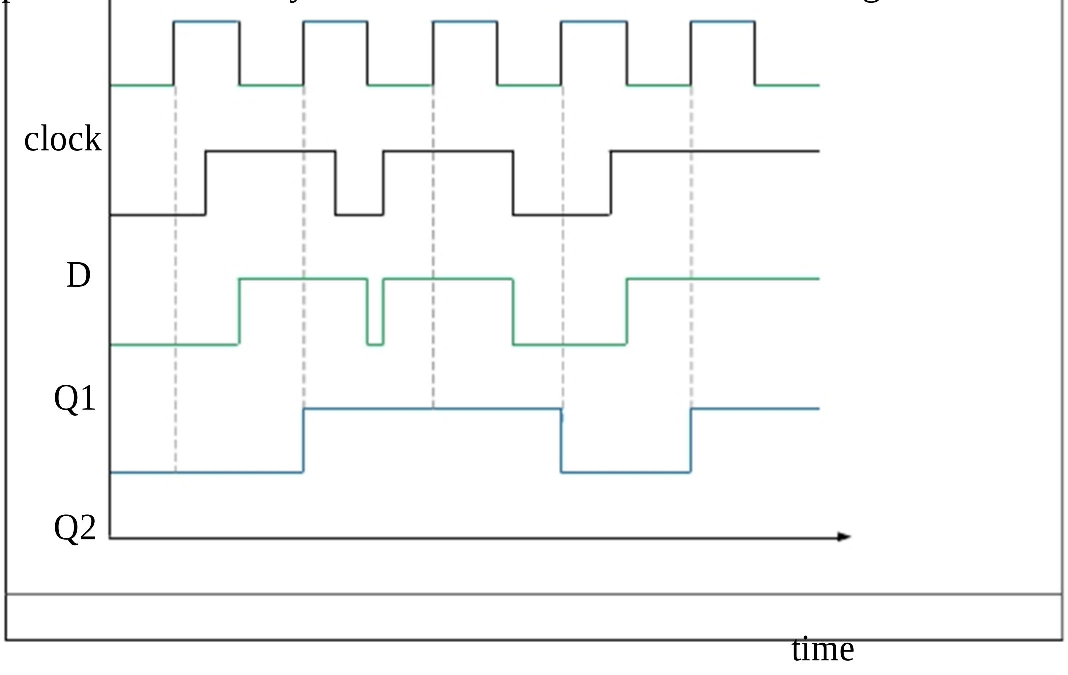


Fig. 5.7. Timing diagram for edge-triggered D flip-flop

Edge-triggered D-flip-flops are widely used in digital electronics. We can buy an integrated circuit containing one or more such flip-flops. The signal Q1 is used only internally and is not available as an output from the D-flip-flop. The signal Q2 is output with the name Q. The inverted signal $\overline{Q}$ is sometimes also available as output.

Truth table for an edge-triggered D-flip-flop:

clock	D	Q
0	x	remember last value
1	x	remember last value
rising edge	0	0
rising edge	1	1

The symbol for an edge-triggered flip-flop has a triangle or wedge at the clock input to indicate that this input is edge-triggered:

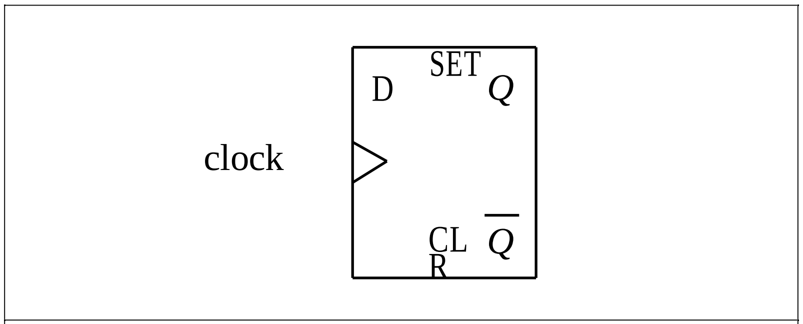


Fig. 5.8. Symbol for edge-triggered D flip-flop

5.1. Edge-triggered D Flip-Flop with set and reset

A flip-flop can have extra inputs to set and reset it, just like the SR flip-flop in figure 5.3. These inputs are used for setting the flip-flop to a desired state without giving it a clock pulse.

Truth table for an edge-triggered D flip-flop with asynchronous set and reset:

clock	D	S	R	Q
0	x	0	0	remember last value
1	x	0	0	remember last value
rising edge	0	0	0	0
rising edge	1	0	0	1
x	x	1	0	1
x	x	0	1	0
x	x	1	1	1

Figure 5.9 shows the symbol for an edge-triggered D-flip-flop with set and reset:

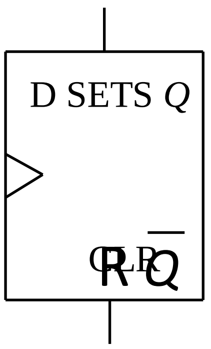


Fig. 5.9. Edge-triggered D flip-flop with set and reset

5.2. Toggle flip-flop

You can make an edge-triggered D flip-flop change at every clock pulse by connecting the inverted output, Q̄, to the D input:

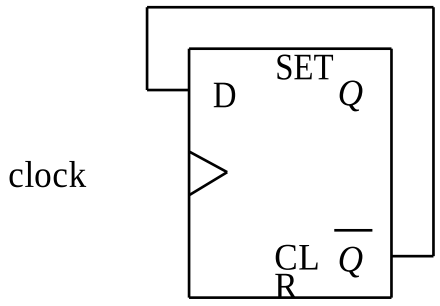


Fig. 5.10. Toggle flip-flop

This can be illustrated with a timing diagram:

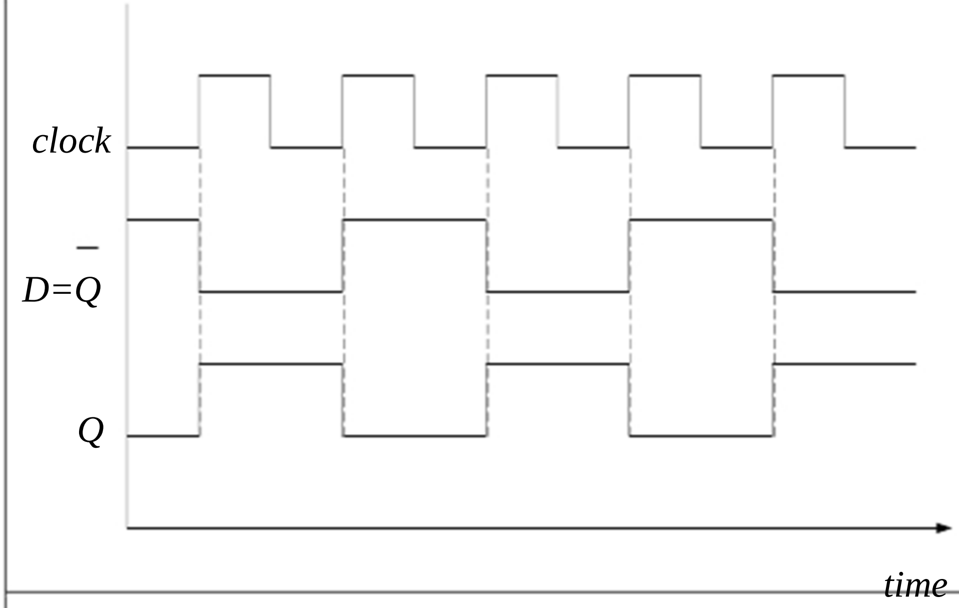


Fig. 5.11. Timing diagram for toggle flip-flop

If you put a square wave into the clock input of a toggle flip-flop, you get a square wave out with half the frequency.

5.1. Ripple counter

You can put multiple toggle flip-flops in a row so that each one will produce a square wave with half the frequency of the previous one:

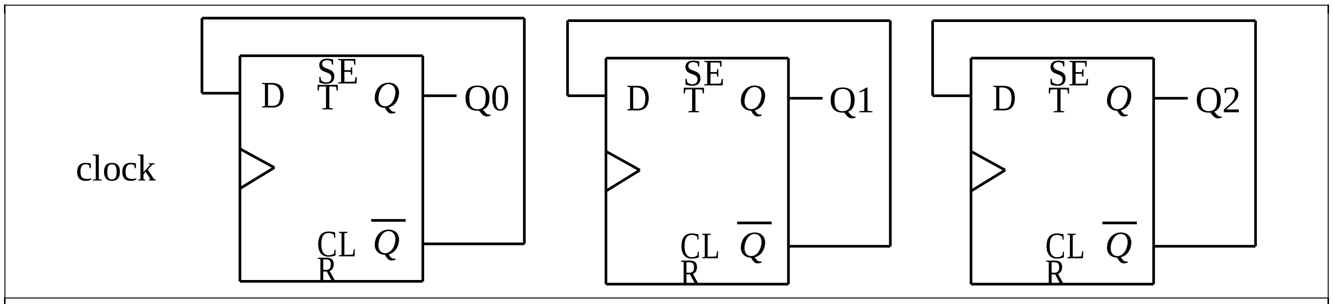


Fig. 5.12. Ripple counter with three bits

The timing diagram below shows that Q0 is changing on every rising edge of the clock input. Q1 changes every second time, and Q2 changes every fourth time.

The bit pattern (Q2,Q1,Q0) is actually the binary numbers from 000 to 111. After 111 it starts over again from 000. We can use this as a counter that counts the binary numbers from zero to seven. We can get more bits in the counter by adding more toggle flip-flops.

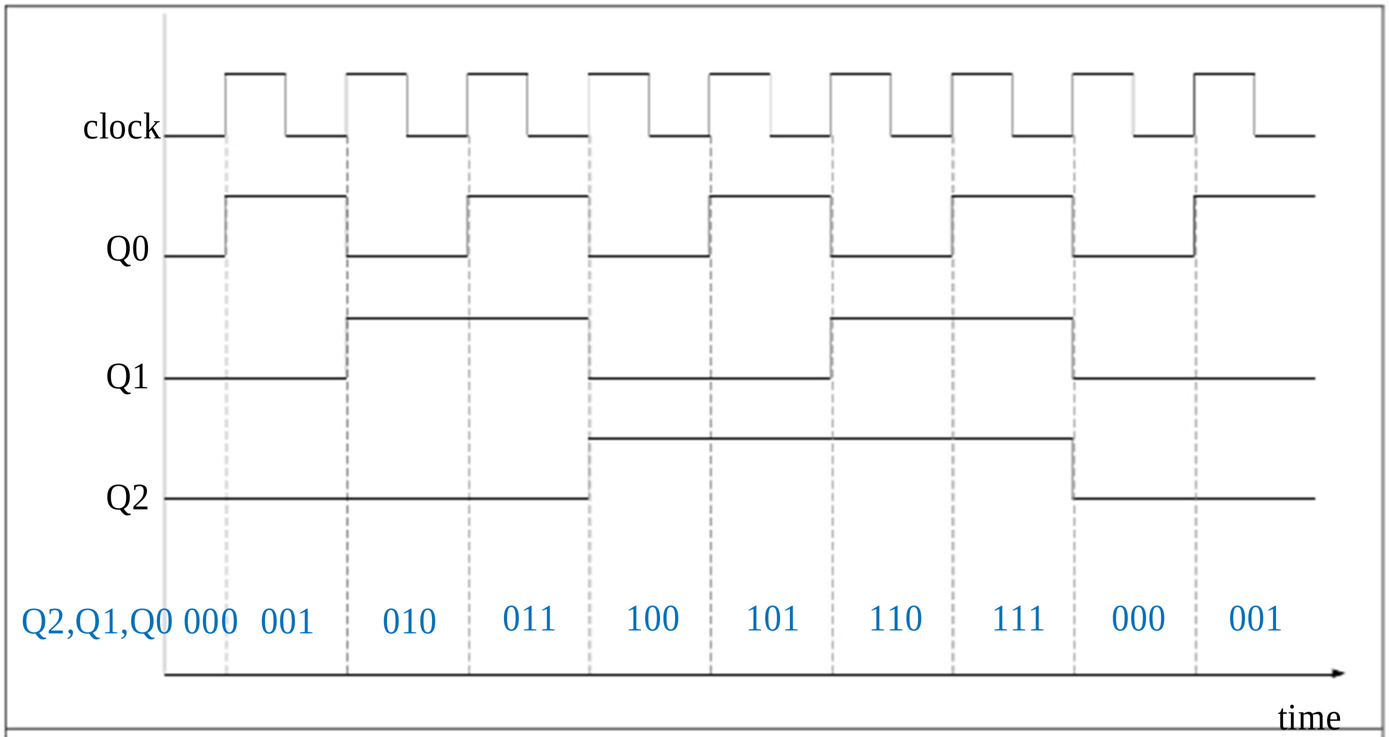
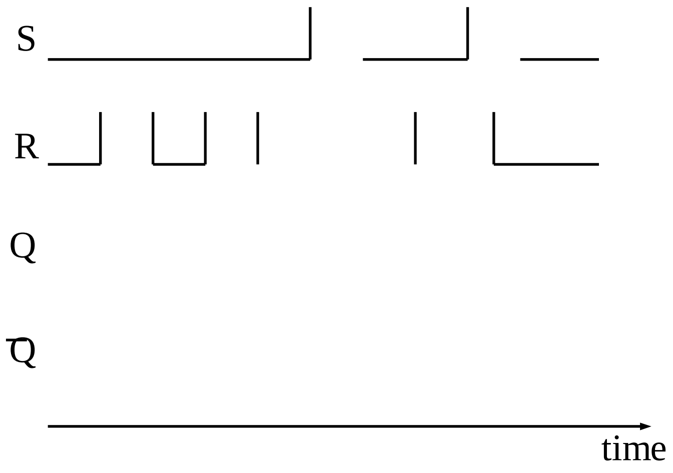


Fig. 5.13. Timing diagram for ripple counter

5.2. Exercises

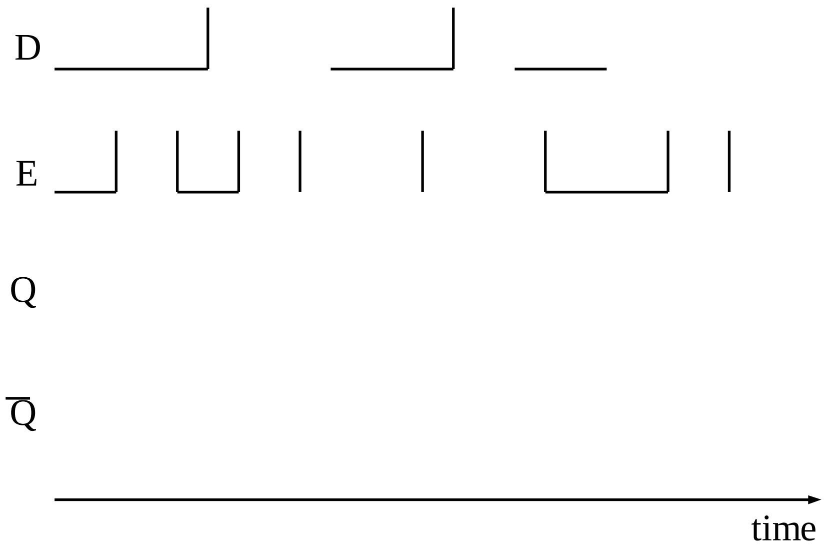
Exercise 5.1.

Show the values of Q and $\overline{Q}$ for an SR flip-flop on this timing diagram.



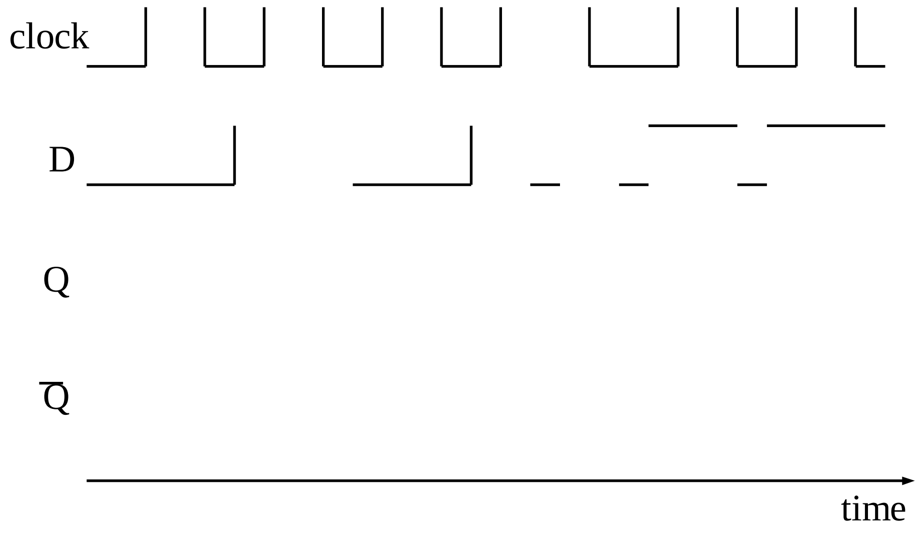
Exercise 5.2.

Show the values of Q and $\overline{Q}$ for a D-latch on this timing diagram.

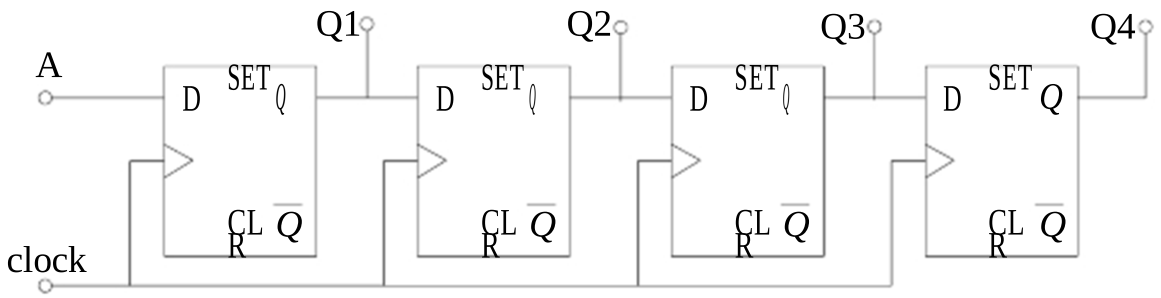


Exercise 5.3.

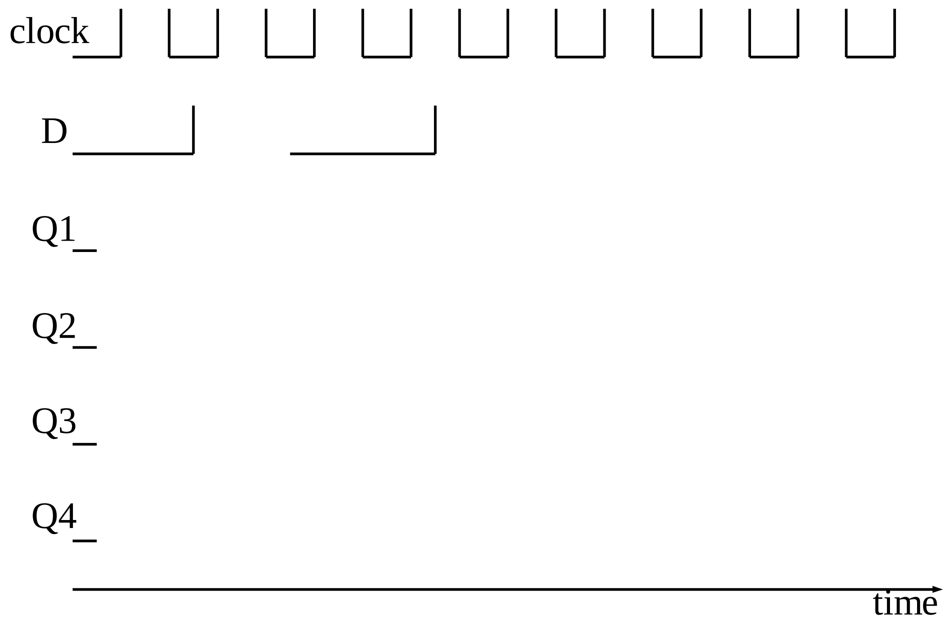
Show the values of Q and $\overline{Q}$ for an edge-triggered D flip-flop on this timing diagram.



Exercise 5.4.



A series of edge-triggered D flip-flops with the same clock is called a *shift register*. Show the outputs Q1, Q2, Q3, Q4 on the timing diagram for this shift register.



Exercise 5.5.

How can you make a toggle flip-flop with enable? The Q output should change on the rising edge of the clock only if an enable input (E) is high. The truth table is shown below. Show with a diagram how
you can implement this with an edge-triggered D flip-flop and gates.

clock	E	Q
rising edge	0	no change
rising edge	1	change to the opposite of last value

Exercise 5.6.

Make a traffic light that changes in the sequence

RED → RED + YELLOW → GREEN → YELLOW.

We can use a counter with two bits to switch between the four states, as shown in this truth table

X1	X0	RED	YELLOW	GREEN
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	0	1	0

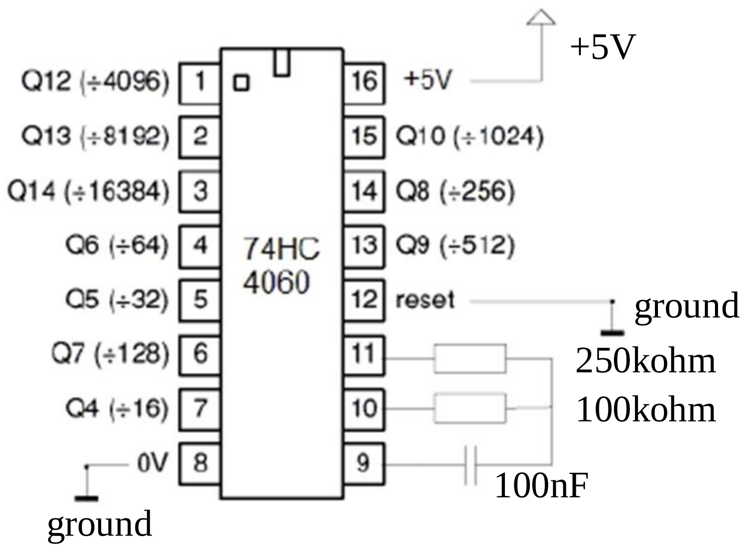
Show the Boolean equations for the three outputs RED, YELLOW, GREEN as functions of X1 and X0.

Draw a diagram of these functions using gates.

Exercise 5.7.

Make the traffic light of exercise 5.6 on a breadboard. You can use the integrated circuit 74HC4060 which contains an oscillator and a ripple counter with 14 toggle flop-flops. Connect it as shown below.
Use two consecutive outputs, for example Q8 and Q9, as X0 and X1. Find the gates you need in the list of digital ICs, page 75.

Use three LEDs in the colors red, yellow, and green as outputs. Connect a 300 Ω resistor in series with each LED.



Q12 (÷4096) 1
Q13 (÷8192) 2
Q14 (÷16384) 3
Q6 (÷64) 4
Q5 (÷32) 5
Q7 (÷128) 6
Q4 (÷16) 7
0V 8
ground
74HC 4060
16 +5V
+5V
15 Q10 (÷1024)
14 Q8 (÷256)
13 Q9 (÷512)
12 reset
ground
11
250kohm
10
100kohm
9
100nF

6. State machines

A state machine is an apparatus that can be in a number of different states and can change its state depending on some inputs. It also has outputs that can be used for controlling something.

An example is an elevator. It can be at different floors and it can move up and down depending on its current position and on the buttons pressed.

Another example is a washing machine. It goes through different states: filling, heating, prewash, wash, rinse, spin. The transition from one state to the next depends on the buttons pushed and on various sensors. The outputs of the electronics control motors, valves, heater, and pump.

State machines can be constructed in software or hardware. In this chapter we will learn how to build a state machine in hardware, using flip-flops and gates.

Let us look at a simple example. Here, we have two edge-triggered D flip-flops connected to the same clock:

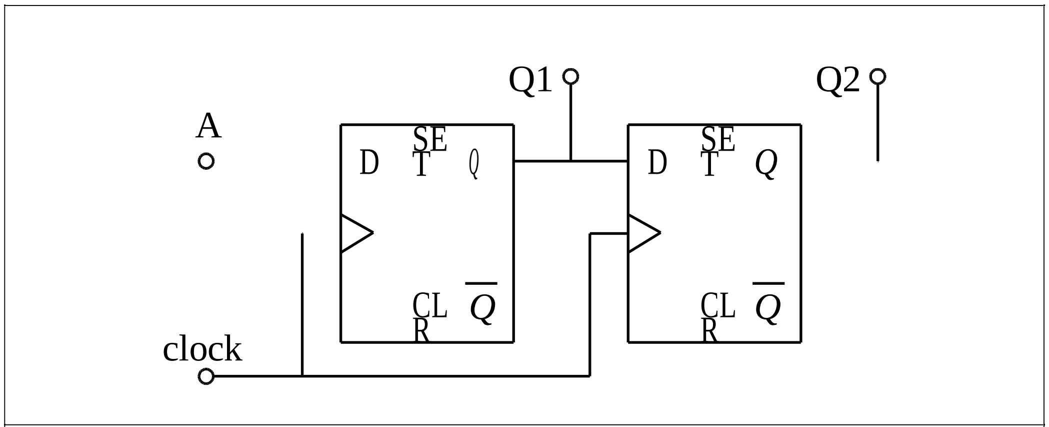


Fig. 6.1. Two bit shift register

A rising edge of the clock will put the value of the A input on Q1 and put the previous value of Q1 on Q2. This circuit has four possible states, defined by (Q1,Q2) = 00, 01, 10, 11. A clock pulse will put it in a new state, depending on the A input. We can write a table of the possible state transitions. This is called a *state table*.

A	current state Q1,Q2	next state Q1*,Q2*
0	00	00
1	00	10
0	10	01
1	10	11
0	01	00
1	01	10
0	11	01
1	11	11

The state transitions can be illustrated in a *state diagram*:

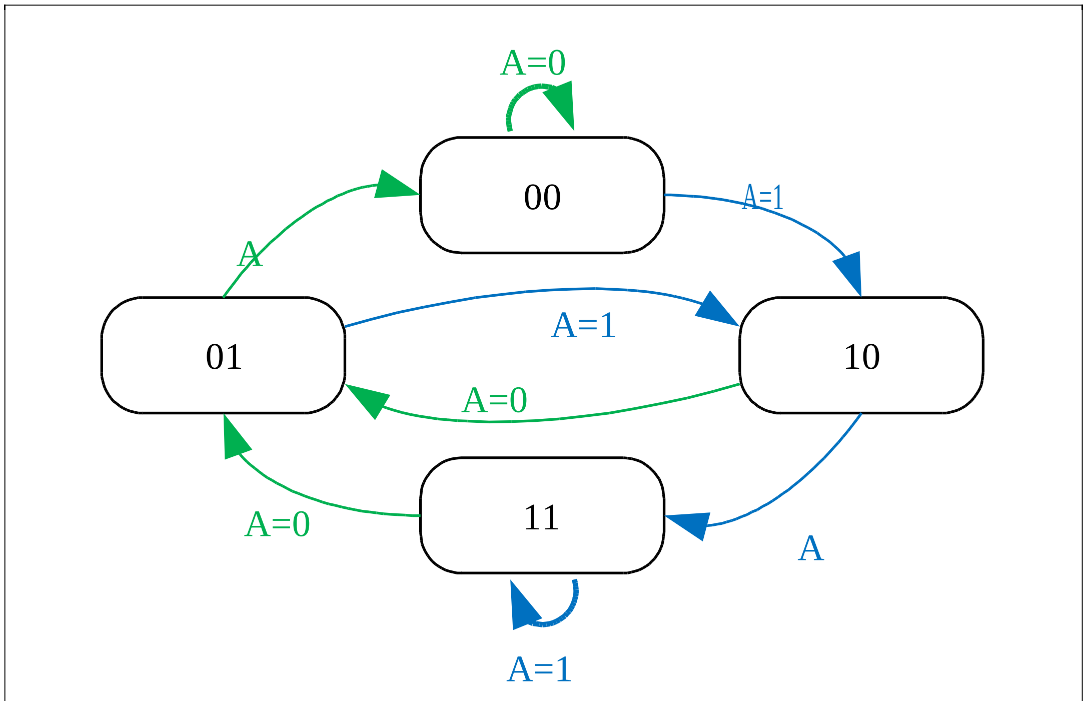


Fig. 6.2. State diagram for shift register in figure 6.1.

The ovals on the state diagram define the states. There are four states named 00, 01, 10, 11. The arrows indicate the transitions. The text on the arrows define the conditions for each transition.

The state diagram above is interpreted like this: If we are in state 00, we will go to state 10 on the next clock pulse if A = 1 (blue arrow) or stay in state 00 if A = 0 (green arrow). If we are in state 10, we will go to state 11 on the next clock pulse if A = 1 (blue arrow) or to state 01 if A = 0 (green arrow).
And so on.

In the next example, we will start with a desired state diagram and then construct a state machine corresponding to this state diagram. This example will be a 2-bit counter with synchronous reset. We want it to count in the sequence of binary numbers 00, 01, 10, 11. Furthermore, we want it to go to state 00 when a reset input, named R, is high while the clock has a rising edge. The R input does nothing unless there is a rising edge of the clock. We can draw a state diagram for this:

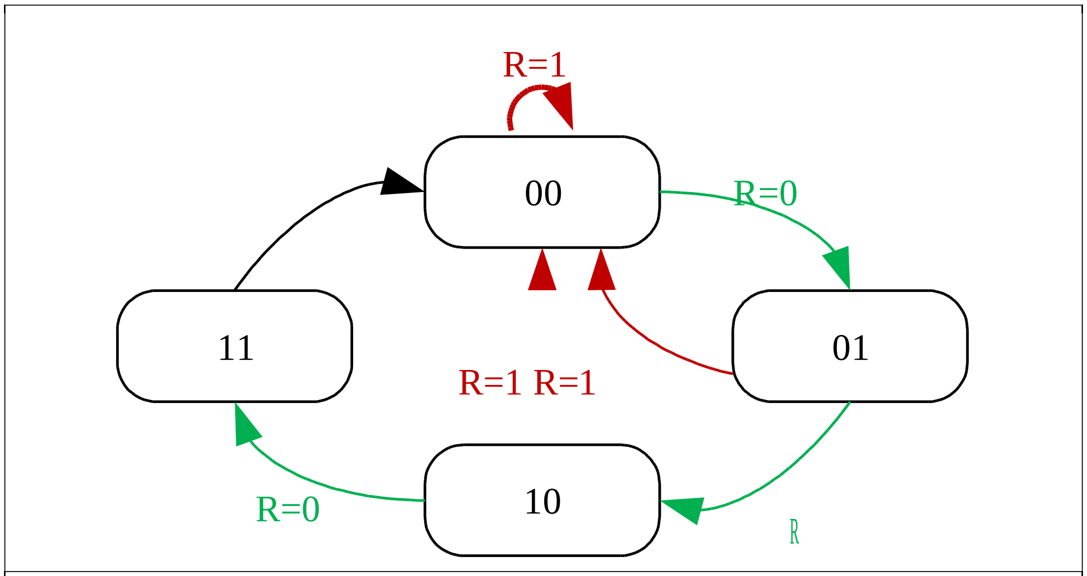


Fig. 6.3. State diagram for 2-bit counter with synchronous reset

This state diagram shows that the counter follows the sequence 00 → 01 → 10 → 11 when R = 0. It goes from any state to state 00 when R = 1. The condition for each transition is written on the arrows. The black arrow has no condition because it goes from state 11 to state 00 regardless of R. We can write a state table based on the state diagram:

input	current state		next state	
R	**Q1**	**Q0**	**Q1***	**Q0***
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	x	x	0	0

This table shows how the next state (Q1*,Q0*) depends on the current state (Q1,Q0) and the input (R). Now we can find the Boolean expressions for Q0* and Q1* using the sum-of-products method described on page 14.

$$Q0* = Q0 * \bar{R}$$

$$Q1* = Q0 * \bar{Q}1 * \bar{R} + Q0 * Q1 * \bar{R} = (Q0 \oplus \mathrm{Q1}) * \bar{R}$$

Now we can build a circuit that implements this functionality. We need two flip-flops for Q0 and Q1, respectively. The Boolean expressions for the next state, Q0* and Q1*, are built with logical gates. These are connected to the inputs D0 and D1 so that they will generate the next state when there is a clock pulse.

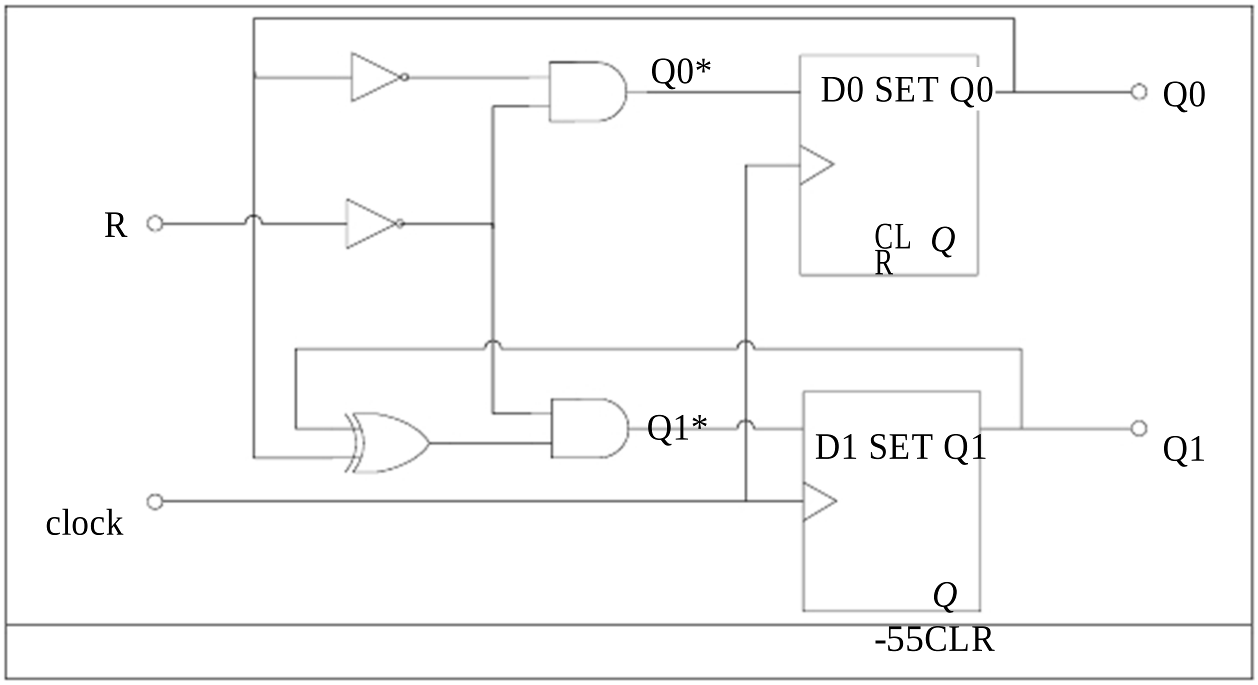


Fig. 6.4. Implementation of 2-bit counter with synchronous reset

A state machine can have various outputs. In the next example, we will make a two-bit counter with two outputs called max and min, indicating when the counter has reached its maximum and minimum value, respectively. Furthermore, we will make the counter so that it counts up when an input U is high, while it counts down when U is low. We will make the counter *saturating*, so that it does not wrap around from 11 to 00, but stays at the maximum value when it is counting up, and stays at the minimum value when it is counting down. This is shown in the following state diagram.

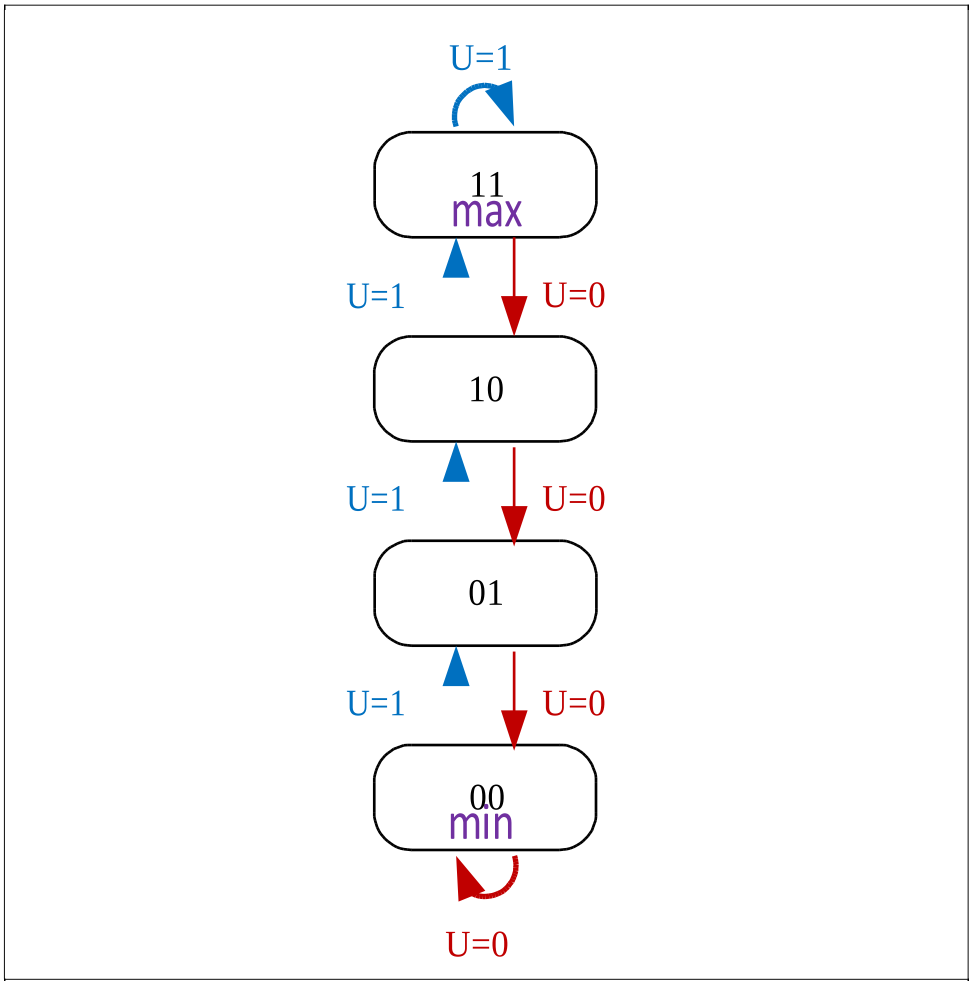


Fig. 6.5. State diagram for saturating up/down counter

This state diagram shows that it is counting up when U = 1 (blue arrows) and stays at the maximum when we keep trying to count up. Likewise, it counts down when U = 0 (red arrows) and stays at the minimum when we keep trying to count down. The outputs max and min (violet) are indicated in the states where they are active.

Let us make a truth table for this saturating up/down counter:

input	current state		next state		outputs	
U	**Q1**	**Q0**	**Q1***	**Q0***	**max**	**min**
0	0	0	0	0	0	1
1	0	0	0	1	0	1
0	0	1	0	0	0	0
1	0	1	1	0	0	0
0	1	0	0	1	0	0
1	1	0	1	1	0	0
0	1	1	1	0	1	0
1	1	1	1	1	1	0

We need the Boolean equations for the next state: Q0* and Q1*, and for the outputs: max and min. Using the sum-of-products method we get the expressions from the truth table:

$Q0* = U * \overline{Q0} * \overline{Q1} + \overline{U} * \overline{Q0} * Q1 + U * Q0 * Q1 + U * Q0 * Q1$

$Q1* = U * Q0 * \overline{Q1} + U * Q0 * Q1 + \overline{U} * Q0 * Q1 + U * Q0 * Q1$

$\text{max} = \overline{U} * Q0 * Q1 + U * Q0 * Q1$

$\text{min} = \overline{U} * \overline{Q0} * \overline{Q1} + U * Q0 * Q1$

We can reduce these equations using the program at www.32x8.com or the program Karnaugh Map Minimizer. The reduced expressions are

$Q0* = \overline{Q0} * Q1 + U * Q0 + U * Q1$

$Q1* = Q0 * Q1 + U * Q0 + U * Q1$

$\text{max} = Q0 * Q1$

$\text{min} = \overline{Q0}\,\overline{Q1}$

Now we can make a diagram for the saturating up/down counter:

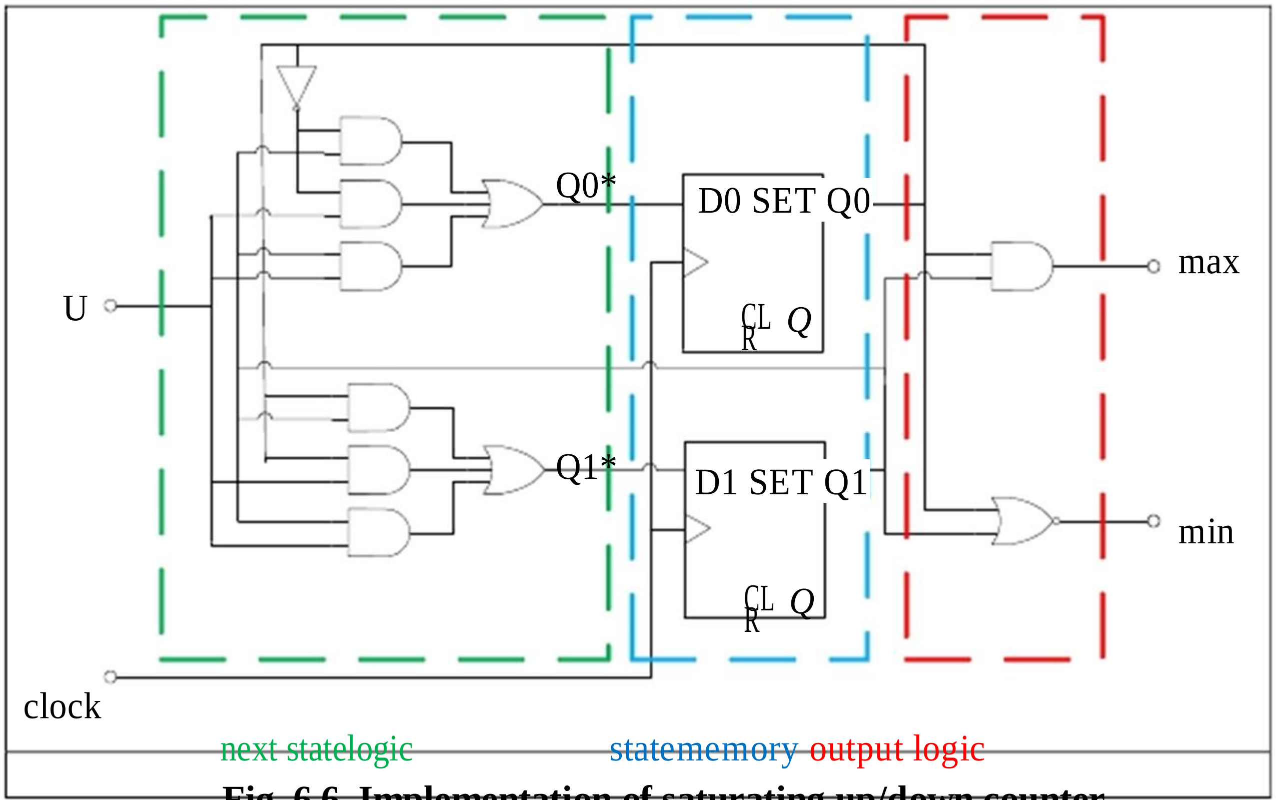


Fig. 6.6. Implementation of saturating up/down counter

This diagram illustrates the general form of a state machine. It consists of three blocks marked with colors on the diagram: *next state logic* (green), *state memory* (blue), and *output logic* (red).

The state memory consists of edge-triggered D flip-flops. The two other blocks consist only of gates and inverters. The next state logic defines the next state (Q0*, Q1*) as a function of the current state

(Q0, Q1) and the input (U). The state memory is the flip-flops that define the current state. The output logic defines the outputs as functions of the current state.

The general form of a state machine can be summarized in the following block diagram.

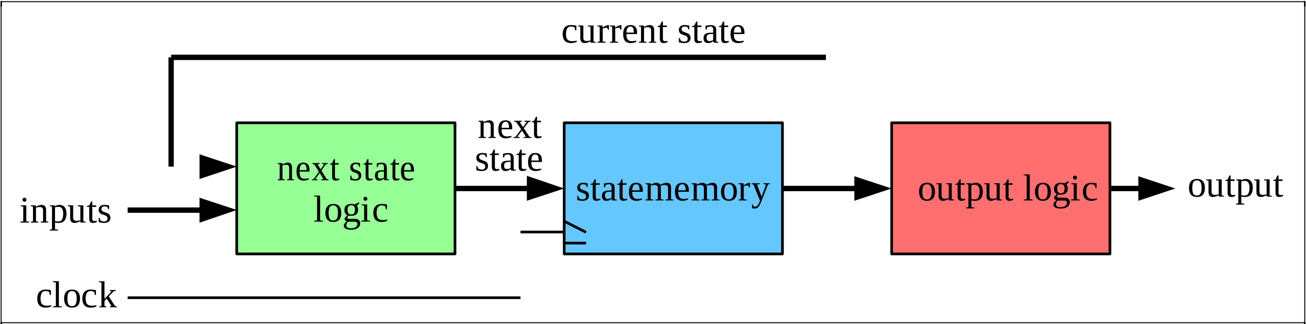


Fig. 6.7. Block diagram showing the general structure of a state machine

As you can see, the next state depends on the current state and the inputs. The output depends on the current state in this example. An output may actually depend on inputs as well. Such an output is called a Mealy output. A Mealy output can change if the input changes, even if there is no clock pulse.

Let us summarize what we have learned about state machines now. It is useful to start with a state diagram if we want to make a state machine. The states are indicated with circles or ovals on the state diagram. The state transitions are indicated by arrows, where the conditions for going to another state are written on the arrows. Outputs are written in the states where they are active.

We can make a state table based on the state diagram. The state table is a truth table where we have the current state and the inputs on the left side and the next state and outputs on the right side. We are using the state table to find the Boolean equations for the next state as a function of the current state and the inputs. We also have to find the Boolean equations for the outputs.

Now, we can build the state machine. The state machine consists of three parts: 'next state logic', 'state memory', and 'output logic'. The 'next state logic' is made of gates that generate the next state as a function of current state and inputs. The 'state memory' consists of edge-triggered D flip-flops. The 'output logic' is made of gates that generate the outputs as functions of the current state.

The number of flip-flops that you need is determined by the number of states. If you have *n* flip-flops, then you have 2n possible states. Choose *n* so that 2n ≥ number of states. You may have some unused states if 2n is bigger than the desired number of states. The state machine may start in one of the unused states by accident when you turn on the power. Therefore, it is recommended that you make the state table so that all unused states will go immediately to a valid state.

It is useful to have a reset input on a state machine so that you can set it to the desired state after turning on the power. Remember that it can start in a random state if you have no reset function. There are two kinds of reset: synchronous and asynchronous. The synchronous reset works only when there is a clock pulse. It is implemented as in figure 6.4. An asynchronous reset works immediately, regardless of the clock. It is implemented by using flip-flops with asynchronous reset as explained on page 38.

6.1. Synchronous design

A digital circuit with flip-flops can be synchronous or asynchronous. Synchronous means that all flip-flops have the same clock, while the flip-flops can be connected to different clocks in an asynchronous design. We will now explain why a synchronous design is more reliable.

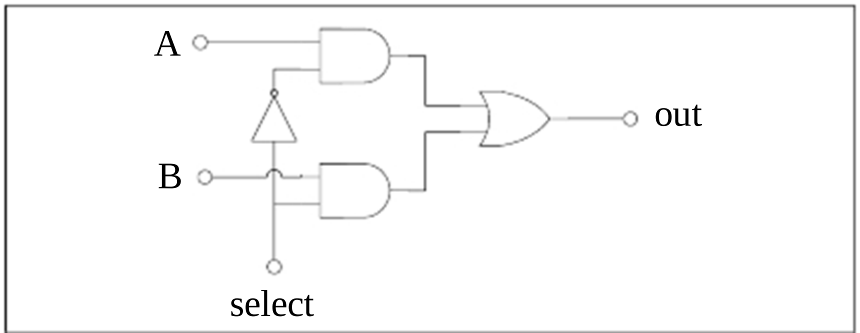


Fig. 6.8. Two-to-one multiplexer

Let us look at the multiplexer in figure 6.8 (same as figure 4.4). Assume that A and B are both high, and the select input goes from high to low. The signals will now change as shown in the following timing diagram.

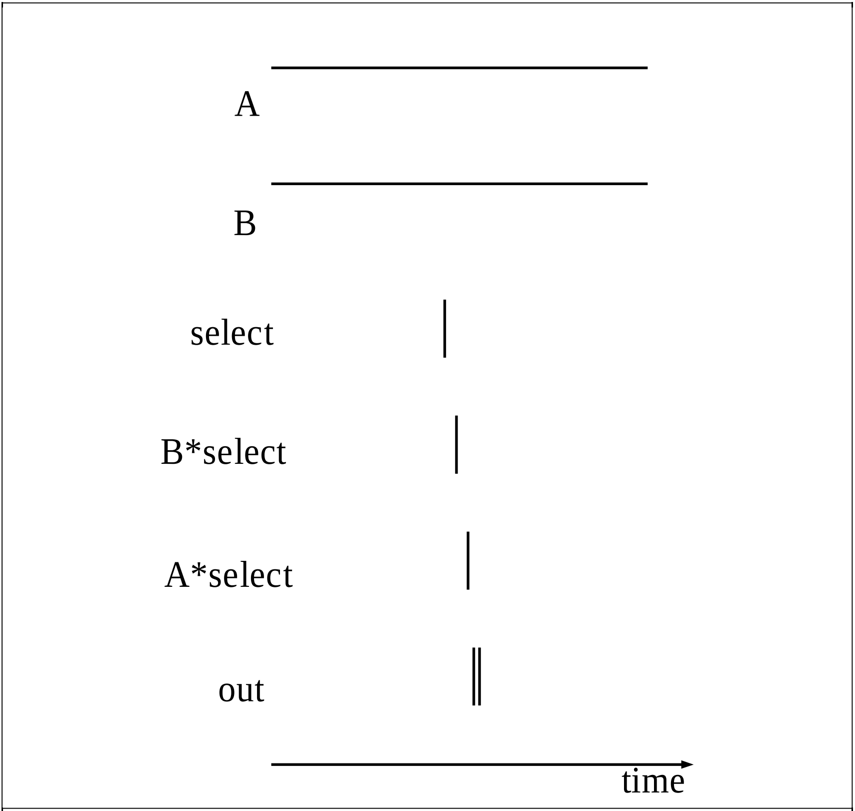


Fig. 6.9. Timing diagram for multiplexer of fig. 6.8

In this timing diagram, we have made a very fine time scale so that we can see the delays in the gates. The $A * select$ signal goes up a few nanoseconds later than the $B * select$ goes down because of the extra delay in the inverter. Now there is a very short moment of time where both of these signals are low. The 'out' signal, which is the OR combination of these two signals, will have a short blip where it is low, even though it is supposed to be high both before and after the change in the select input. This kind of noise in the signal is called a *glitch*. Such a glitch can cause problems if the signal goes to some other circuit that has memory, such as a latch or a flip-flop. For example, we may get an extra count if this signal goes to the clock input of a counter.

It can be difficult to detect this kind of errors because a glitch may be too short to see on an oscilloscope, but not too short to change the state of a flip-flop.

A good remedy against such timing problems is to have everything controlled by the same clock, as we did in the design of state machines above. The 'next state logic' in a state machine can produce many glitches, but this will not cause any errors because all glitches will have died out before the next clock pulse. The only thing we have to care about is that the clock period must be longer than the worst-case delay in the circuits. A digital circuit where all signals are controlled by the same clock is called a *synchronous design*.

The counters in figure 6.3 and 6.5 were synchronous designs. These are called synchronous counters. In the previous chapter we had a ripple counter in figure 5.12. This is an asynchronous counter because the flip-flops are not connected to the same clock.

Let me explain why the asynchronous counter can be less safe to use. The output of the ripple counter in figure 5.12 is a three-bit binary number (Q2,Q1,Q0). It counts from binary 000 to 111 and then starts over at 000.

Assume that we want a signal, P, every time the counter reaches 000. We can make this with a NOR gate: $P=\overline{Q0+Q1+Q2}$. Now, the three output bits from the counter do not change exactly simultaneously. For example, let us see what happens when the counter goes from binary 011 (=3) to 100 (=4). First Q0 goes low, then Q1 goes low, and then Q2 goes high. There will be a short moment of time after Q1 goes low, and before Q2 goes high, where the outputs are actually 000, and we will see a short glitch in the signal P. This may cause problems if P goes to a circuit with memory.

The asynchronous counter is cheap, and it is good enough for many purposes, but the synchronous counter is better if you want a stable output.

Figure 6.10 shows another example of an asynchronous design. Here we have an AND gate on the clock input of the second flip-flop so that it is only clocked when the input E (enable) is high. This design is unsafe because there is a short delay in the AND gate. The second flip-flop is clocked slightly later than the first flip-flop. We do not know whether the second flip-flop is getting the old value that Q1 had before the rising edge of the clock or the new value that Q1 has after the clock. It may be that it sometimes gets the old value of Q1 and sometimes the new value of Q1. Such an unreliable design is dangerous because it may work correctly when we test it, but fail later.

Another problem is that the second flip-flop gets clocked at the rising edge of E if the clock is high. This is perhaps not the behavior we want.

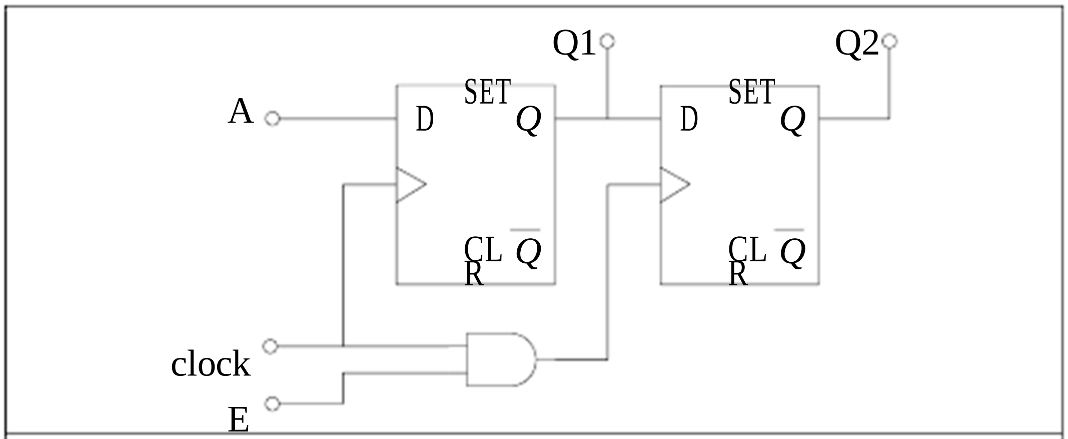


Fig. 6.10. Unsafe circuit with gated clock

A better way to enable and disable a flip-flop is a synchronous design:

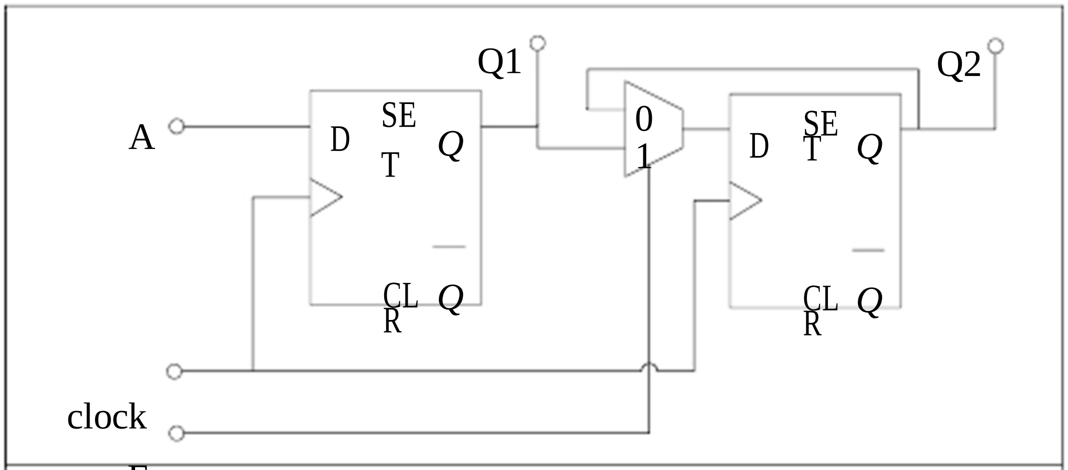


Fig. 6.11. Synchronous alternative to fig. 6.10

Here, the two flop-flops have the same clock. The multiplexer connects the D input of the second flip-flop to Q1 then E is high, while the value of Q2 is recycled when E is low. The effect is that changes in Q2 are enabled only when E is high. There is no undesired clocking at the rising edge of E.

In general, it is safer to use a synchronous design because you avoid a lot of delicate timing problems. A synchronous design is not completely safe, though, if an input changes simultaneously with the clock.

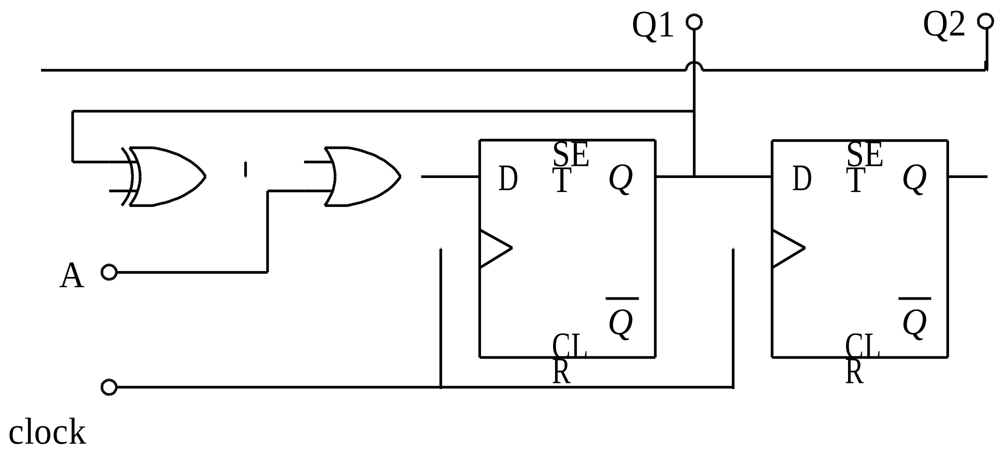
6.2. Exercises

Exercise 6.1.

Which of these circuits are combinational and which are sequential?

- ☐ inverter
- ☐ decoder
- ☐ multiplexer
- ☐ D latch
- ☐ counter
- ☐ adder
- ☐ edge-triggered D flip-flop
- ☐ shift register
- ☐ state machine
- ☐ each of the three blocks in figure 6.7

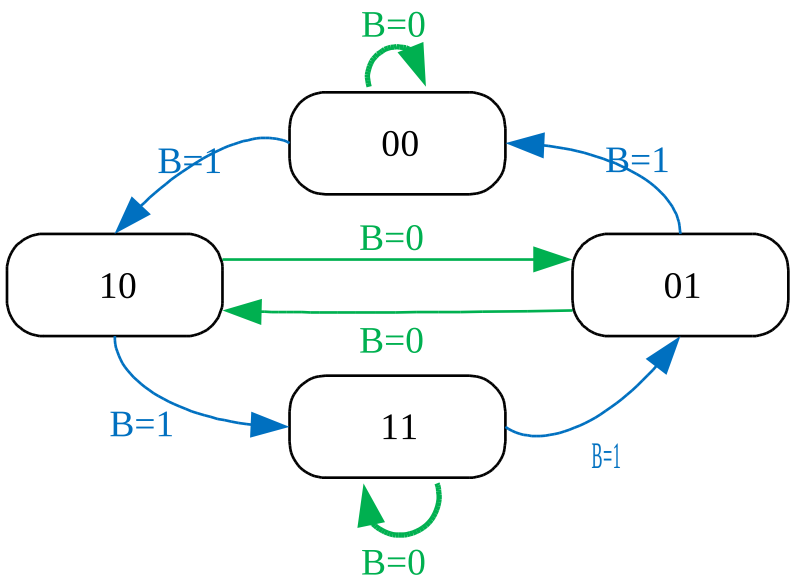
Exercise 6.2.



Make the state table and the state diagram for this state machine.

Exercise 6.3.

We want to design a state machine with the state diagram shown here.



Make the state table. Write the Boolean equations for the next state as a function of the current state and the B input. Reduce these equations using the rules of Boolean algebra.

Draw a diagram of this state machine, using as few gates as possible.

7. Interfacing digital circuits

This chapter gives some practical advice on how to connect inputs and outputs to digital circuits.

7.1. Pushbuttons

The input of a MOSFET digital circuit is controlled by the voltage at the input, not the current. The input resistance of a MOSFET is extremely high and the input current is virtually zero. While an ordinary switch can turn on and off the current for a lamp, it cannot turn on and off the input of a MOSFET circuit because there is no current. An input that is not connected to anything will have a random and unpredictable voltage that is influenced by even the smallest amount of electromagnetic noise. If you connect a switch to an input then you need a *pull-down resistor* to make sure the input voltage is low when the switch is off. The value of the resistor is not important.

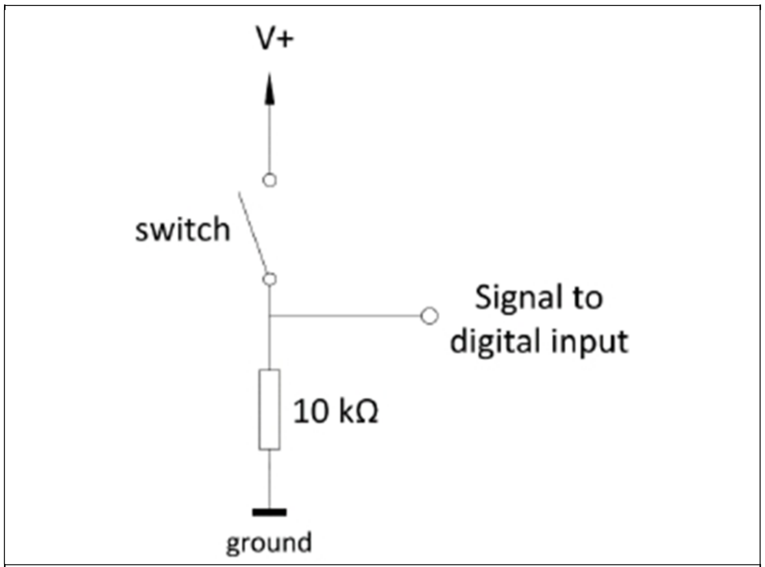


Fig. 7.1. Switch with pull-down resistor

You can invert the signal by connecting the switch to ground and a pull-up resistor to the positive supply:

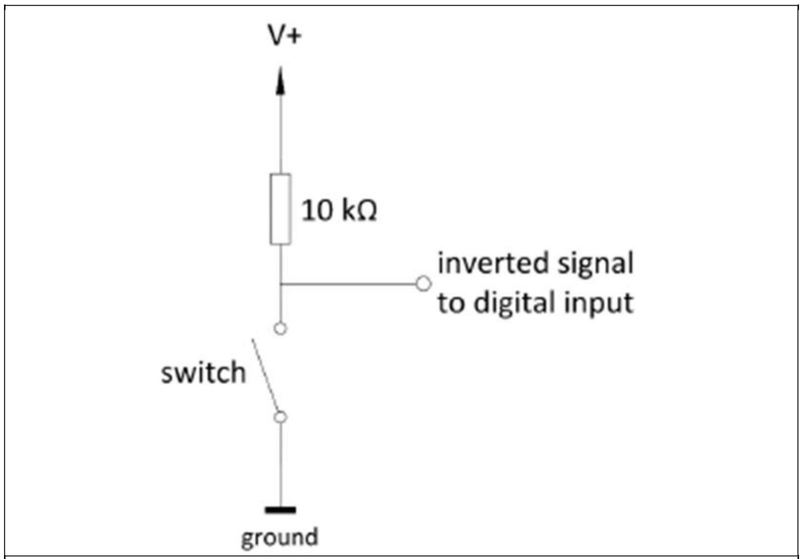


Fig. 7.2. Inverted switch with pull-up resistor

7.2. Button with debouncing

A mechanical switch will always produce noise when it is pressed or released. If you connect the output of the circuit in figure 7.1 to an oscilloscope it will typically look like this:

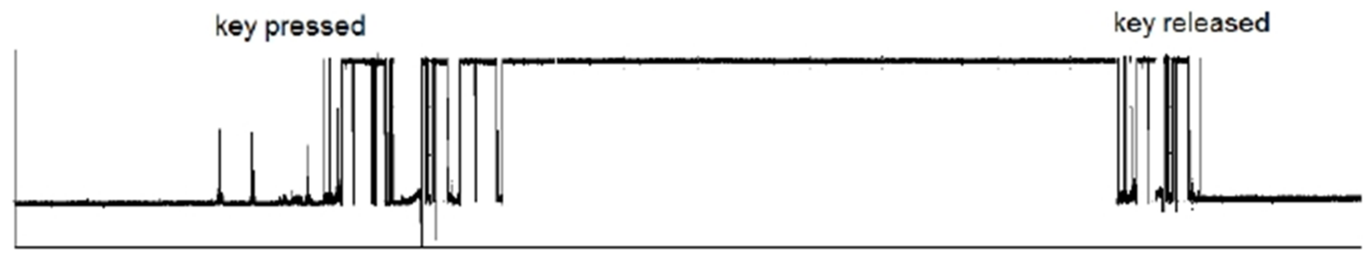


Fig. 7.3. Oscilloscope picture of key bounce

This noise comes from mechanical vibrations in the switch. The click sound you hear when you press a switch is actually the sound of such mechanical vibrations. The signal goes up and down many times during a few milliseconds when the switch or button is pressed or released. The electrical noise from switches and pushbuttons is called *key bounce*. Relay switches and other mechanical devices can also produce bouncing noise.

The key bounce is no problem when the button is connected to the input of an SR flip-flop. The SR flip-flop will be set whether the S input receives one pulse or a hundred pulses. But the key bounce is a big problem if the input is connected to an edge-triggered clock input or anything else that requires a clean signal. For example, if you connect a pushbutton to a counter in order to count how many times the button is pressed, you will see it counting maybe twenty times or more every time the button is pressed, and it may also count when the button is released.

The noise will typically make the signal go up and down for a period of a few milliseconds for a new switch and up to 20 ms, or perhaps 50 ms, for an old and dirty switch. We have to remove this noise if the circuit cannot accept noise. Removing noise from a switch is called *debouncing*. We want to ignore the noise and accept a change in the signal only after a period of, for example, 50 ms.

There are several ways to remove key bounce. Some different methods are described here.

Method 1. Low-pass filter and Schmitt trigger

We can use a low-pass filter to separate the signal from the noise because the noise has higher frequency than the signal from the switch. The diagram below shows a pushbutton with a pull-down resistor (R1) and a low-pass filter (R2 and C1). The low-pass filter will remove most of the noise, but it
will also make the signal change slowly. An edge-triggered flip-flop requires that the clock input has sharp edges so that it is rising and falling very fast. We will use a so-called Schmitt trigger for this.

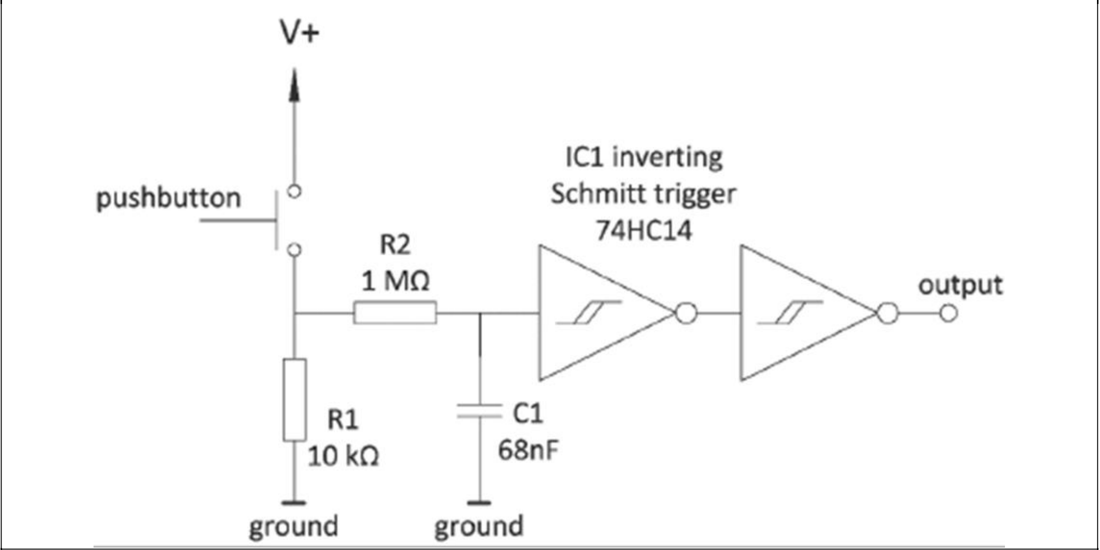


Fig. 7.4. Switch debouncing with low-pass filter and Schmitt trigger

A Schmitt trigger has an input with two threshold values. The output goes high when the input passes the upper threshold, and it goes low again when the input passes the lower threshold. The output stays in the same state when the input is between the two thresholds. This makes sure that the output of the Schmitt trigger always changes fast.

The integrated circuit 74HC14 contains six Schmitt triggers. There is an inverter on the output of each Schmitt trigger. In the diagram above, we have used an extra Schmitt trigger as inverter to get a non-inverted output. Alternatively, we could have inverted the input by connecting the pushbutton to ground, as in figure 7.2.

The low-pass filter has a delay of $\tau = R2C1 ln2 = 47ms$. This should be suitable in most cases. If the delay is too short you will get bounce problems when the pushbutton becomes old and worn. If the delay is too long you will get no response if the user pushes the button very fast.

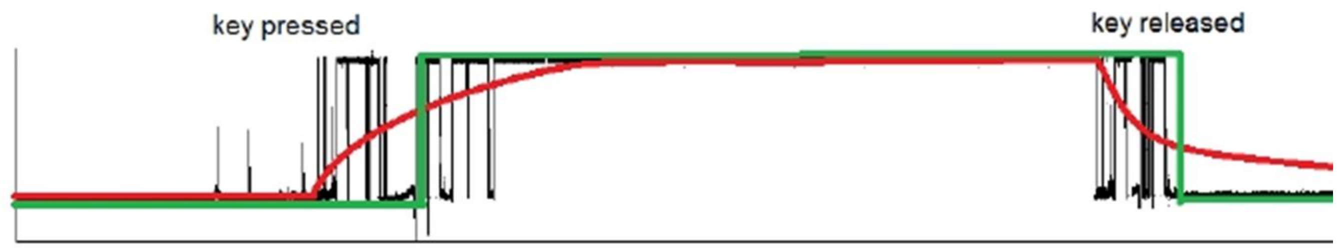


Fig. 7.5. Waveforms of circuit on fig. 7.4

The black curve shows the noisy signal from the pushbutton. The red curve shows the signal after the low-pass filter. The green curve shows the signal after the Schmitt trigger.

Method 2. Use a double-throw switch

Debouncing is easier if we have a double-throw switch (SPDT switch). In this diagram, we have made an SR flip-flop from NOR gates. This flip-flop will be set when the switch is up and reset when the switch is down. The flip-flop will stay in the same state during bounce periods and while the switch is in transition.

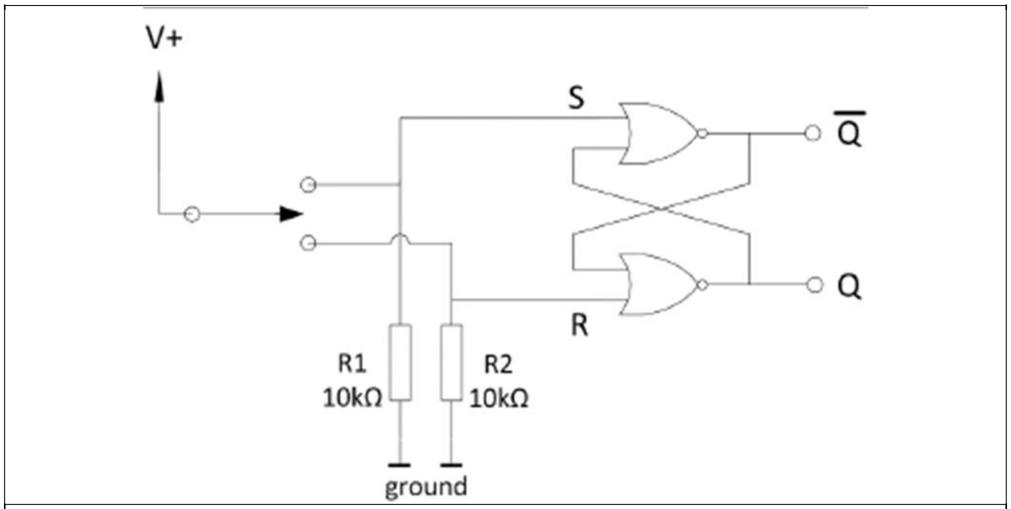


Fig. 7.6. Debouncing of double-throw switch using SR flip-flop

Method 3. Toggle flip flop with debounce

Figure 7.7 shows a nice trick that makes it possible to build a toggle switch with debounce from a pushbutton and two CMOS NOR gates. The NOR gates are coupled as a flip-flop with positive feedback through R2. It is switched to the opposite state by the pushbutton due to the negative feedback that charges C1 through R1. The output will toggle every time the pushbutton is pressed. The debounce delay is $\tau = R1C1ln2 = 47ms$. R3 and C2 serve to reset the flip-flop when the power is turned on.

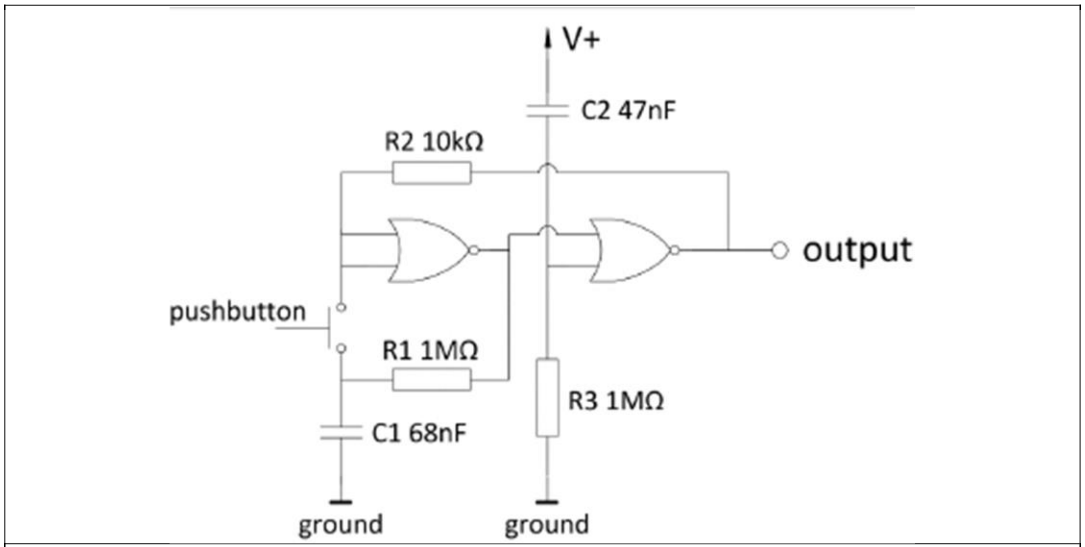


Fig. 7.7. Pushbutton with debouncing toggle flip-flop

Method 4. Shift register

Debouncing can be made by connecting the noisy signal to the D input of a shift register with two or more stages, as shown in figure 7.8. Connect a clock of approximately 100 Hz to the clock input. This will sample the signal every 10 ms. An SR flip-flop is set when all sample values are high and reset when all sample values are low. This method is used in programmable devices (FPGAs) where you do not have access to analogue filters.

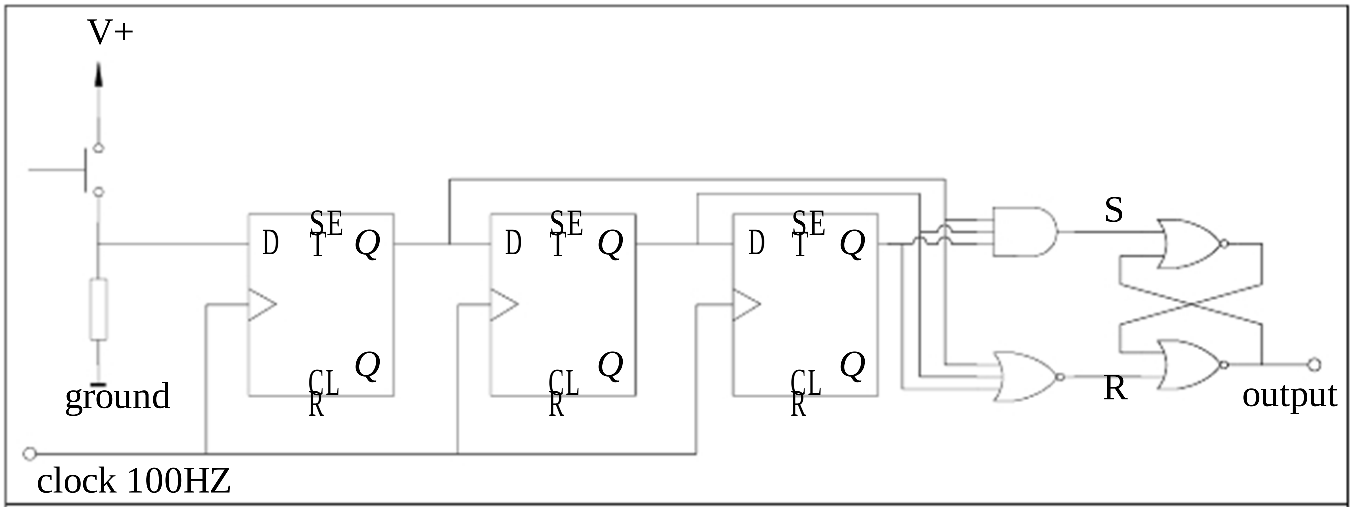


Fig. 7.8. Switch debouncing with shift register

Method 5. Software

Software is generally cheaper than hardware. Key debouncing is therefore often made in software if the apparatus includes a microcontroller anyway. The software will sample the signal two or more times with 10 ms or more between, and accept a change only if the signal remains stable over several samples.

7.3. Automatic power-on reset

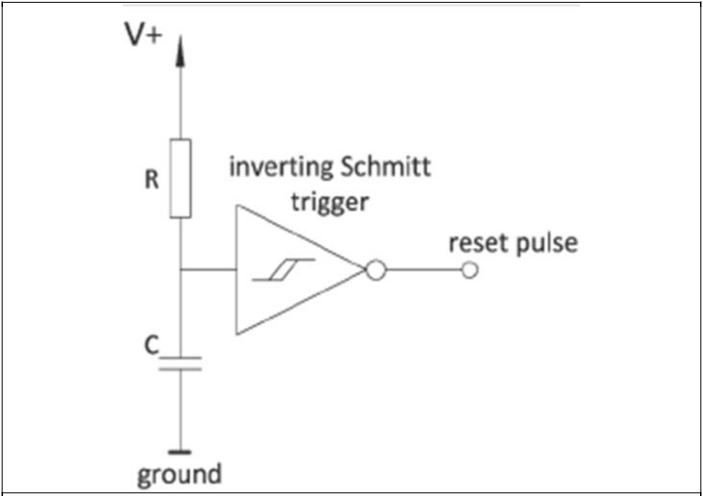


Fig. 7.9. Automatic power-on reset

A system containing flip-flops should be set to a well-defined state when the power is turned on. This is done by sending a pulse to the reset inputs of all flip-flops. Figure 7.9 shows a circuit that generates a pulse when the power is turned on. The Schmitt trigger makes sure the signal has sharp edges. The length of the reset pulse is $\tau = R \cdot C \cdot \ln(2)$.

7.4. LED output

A light-emitting diode (LED) has a voltage-current characteristic like this:

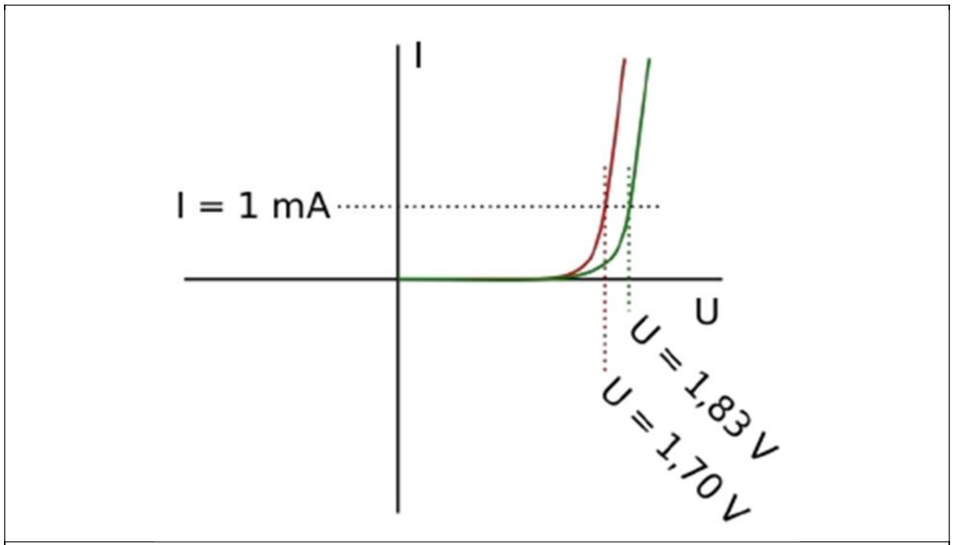


Fig. 7.10. Voltage-current characteristic of LED

Figure 7.10 shows that the LED needs a certain voltage before it can produce light. This voltage corresponds to the energy of the photons it emits. The photon energy for visible light ranges from 1.8 eV for red to 3.9 eV for violet. The operating voltage of a LED corresponds to the color, so that red, yellow and green LEDs operate at approximately 2 V while blue and white LEDs require 3-4 V.

You need to put a resistor in series with a LED in order to reduce the voltage and control the current, as shown in figure 7.11. The light intensity is controlled by the current. A current of 10 mA is suitable for a LED that is used as an indicator lamp.

The resistor is calculated by Ohm's law. For example, if the digital signal is 5V and the LED needs 2V, you need a voltage drop across the resistor of 5V - 2V = 3V. If you want a current of 10mA you need a resistor of R = 3V / 10mA = 300Ω.

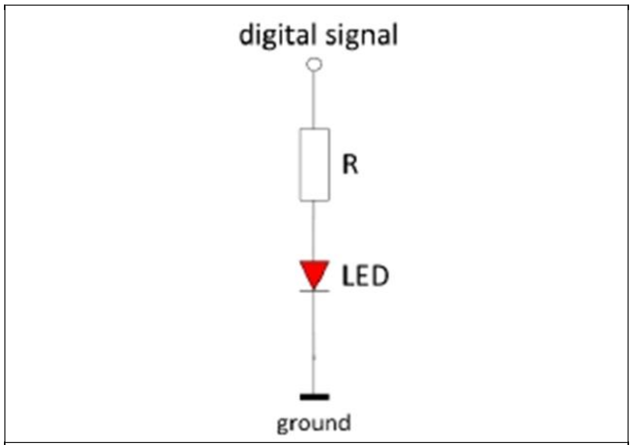


Fig. 7.11. LED with series resistor

The longest pin on the LED is the positive electrode (anode). The pin that holds the semiconductor chip inside the colored plastic house is the negative electrode (cathode) as shown in figure 7.12.

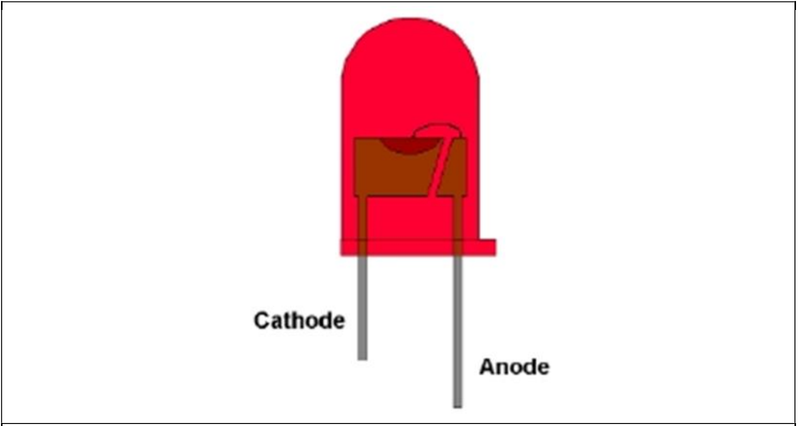


Fig. 7.12. LED terminals

7.5. Relay output

Figure 7.13 shows a relay driver. The transistor amplifies the current for the relay. This is needed if the relay coil requires more power than the digital circuit can provide.

Note that the diode is connected in the direction where it does not conduct the current from V+. This diode protects the transistor against the inductive current that occurs when the relay is turned off.

The same circuit can be used for driving motors, lamps, and other things. The diode is needed if the device contains coils or long wires or anything else that gives it a significant self-inductance.

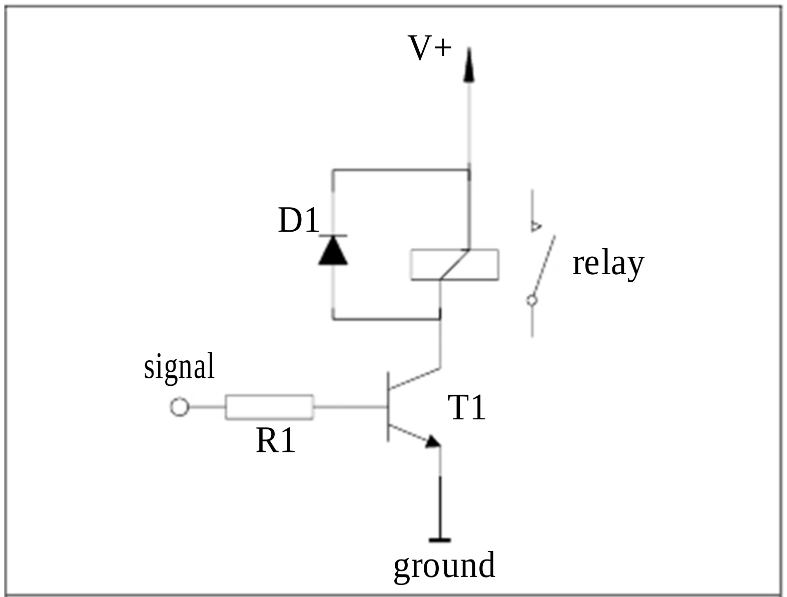


Fig. 7.13. Relay driver

7.6. Optocouplers

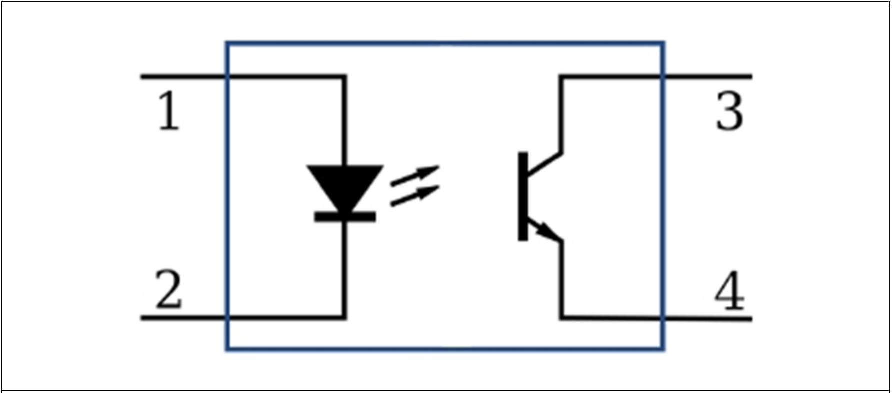


Fig. 7.14. Optocoupler

An optocoupler consists of a light-emitting diode (LED) and a phototransistor mounted together in a non-transparent housing. There is no electrical connection between the two components - they are connected only through the light. The phototransistor is conducting when the LED throws light on it.

An optocoupler is useful for transmitting signals between two circuits that do not have the same voltage level. The two circuits may have separate power supplies, separate voltage levels, and separate ground levels. An optocoupler is also useful if two circuits need to be isolated from each other for security reasons, for example if one circuit has high voltage and the other circuit is in contact with humans.

The LED of the optocoupler is connected as in figure 7.11. The phototransistor is connected like a switch.

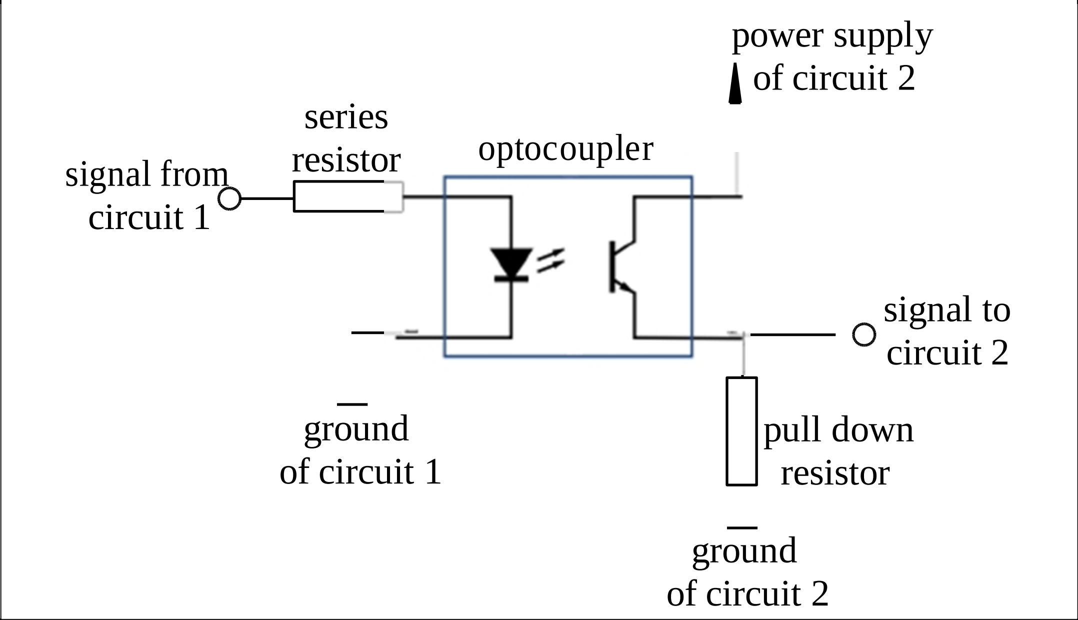


Fig. 7.15. How to connect an optocoupler

7.7. Digital-to-analog converters

A digital-to-analog converter is a circuit that converts a binary digital signal to an analog signal. The input signal consists of multiple bits that are either 0 or 1. The output is a single line with a voltage that is proportional to the binary value of the input bits.

A digital-to-analog converter can be constructed conveniently with a so-called R-2R ladder, as shown in figure 7.16. The resistors marked R all have the same value, for example 10 kΩ. The resistors marked 2R have exactly the double resistance, which would be 20 kΩ in our example. The binary bits are represented in the diagram as switches that connect a wire to ground when the bit is 0, and to a certain voltage *V* when the bit is 1. In reality, these are not mechanical switches but typically MOSFET transistors that connect a wire to ground or to the positive power supply for the logical values of 0 and 1, respectively.

The voltage that comes out of the R-2R ladder can be calculated by repeated use of Thévenin's theorem. Let us start with the first node marked v0 on the drawing and disconnect it from the next node v_1. The Thévenin equivalent of v_0 has a voltage $V_0 = \frac{1}{2} b_0 V$ and an equivalent resistance equal to R.

Now we can calculate the Thévenin equivalent of the next node, v1, when it is connected to v0 but not to v_2. The voltage is $V_1 = \left(\frac{1}{4} b_0 + \frac{1}{2} b_1\right) V$ and the resistance is R.

We continue with v_2 disconnected from v_3: The voltage is $V_2 = \left(\frac{1}{8} b_0 + \frac{1}{4} b_1 + \frac{1}{2} b_2\right) V$ and the resistance is R.

Finally, $V_3 = \left(\frac{1}{16} b_0 + \frac{1}{8} b_1 + \frac{1}{4} b_2 + \frac{1}{2} b_3\right) V$ with the equivalent resistance R.

We can add an operational amplifier to amplify the signal v3 by the factor $\frac{16}{V}$. This gives [illegible] = $\frac{16}{V} v3 =$ $d0 + 2d1 + 4d2 + 8d3$. This voltage is indeed the same as the binary number formed by the four input bits. We can add more steps in the ladder if the input has more bits.

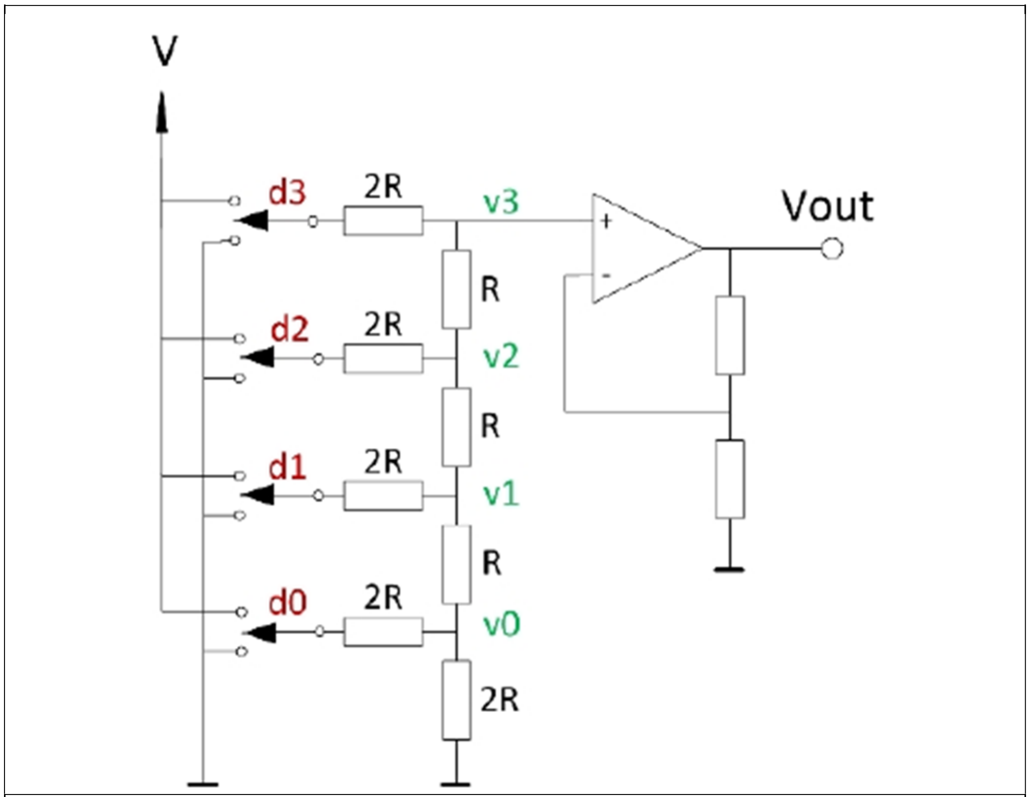


Fig. 7.16. Digital-to-analog converter with R-2R ladder

7.8. Analog-to-digital converters

An analog-to-digital converter is the opposite of a digital-to-analog converter. The input is a single wire with an analog signal. The output is a binary signal consisting of multiple bits. The precision or resolution is determined by the number of output bits. With n bits we can get a resolution of the total range divided by 2n.

We can make a simple analog-to-digital converter with a counter and a digital-to-analog converter. The counter counts up until the converted binary number exceeds the analog input value. The principle is outlined in figure 7.17.

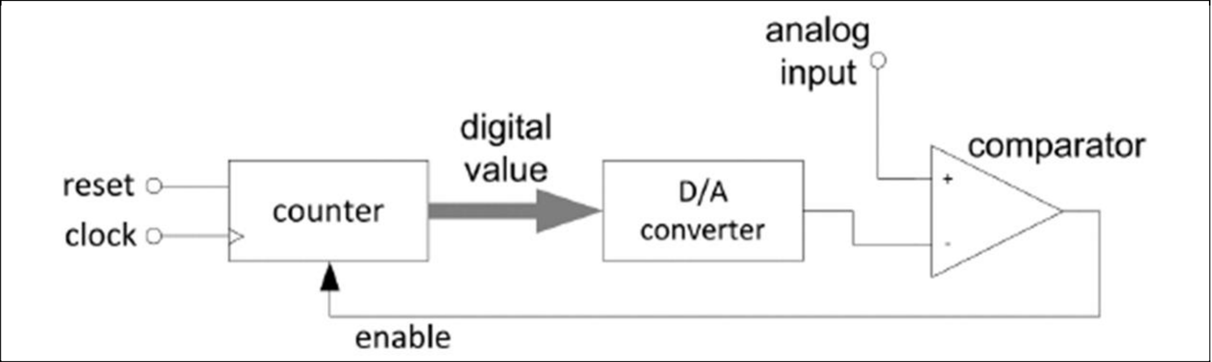


Fig. 7.17. Analog-to-digital converter with counter

This method is slow because it will take up to 2n clock pulses before we have reached the desired value.

We can get a faster conversion by using the *successive approximation* method. This method finds the n-bit binary value by first dividing the interval from 0 to 2n into two sub-intervals of half the size, from 0
to 2n-1 and from 2n-1 to 2n. The system finds out whether the input is in the lower or the upper of these
two half-intervals by comparing the analog input to 2n-1. Now that it knows which half-interval the number is in; it divides this half-interval into two intervals of 1/4 size by comparing the input to the middle value of the subinterval. It continues to divide the intervals into halves n times. The first comparison gives the most significant bit of the result, and the last comparison gives the least significant bit. The successive approximation method can be described by the flow chart in figure 7.18.

This flow chart can be implemented as a state machine that replaces the counter in figure 7.17. The successive approximation method is much faster than the counter method because it takes n clock cycles where the counter method uses up to 2n clock cycles.

If you need an extremely fast analog-to-digital converter, you can use a flash converter. The flash converter contains 2n comparators, one for each possible output value. The flash converter becomes very big and expensive if n is big. Therefore, it is used only for low resolutions.

There are several other methods for analog-to-digital conversion. I will not explain all the different types, but you can find more information on Wikipedia if you are interested.

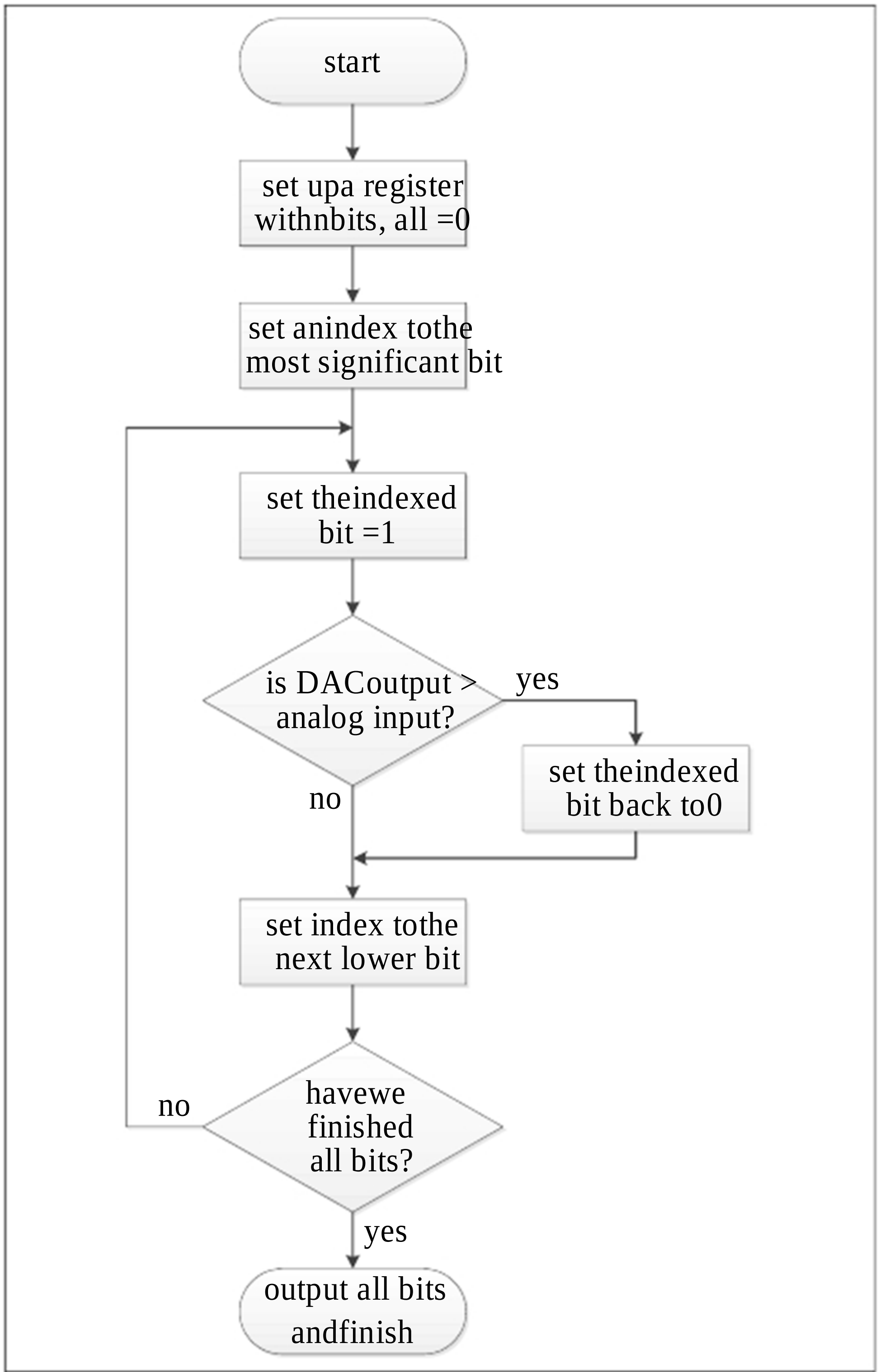

start
set upa register withnbits, all =0
set anindex tothe most significant bit
set theindexed bit =1
is DACoutput > analog input?
yes
set theindexed bit back to0
no
set index tothe next lower bit
havewe finished all bits?
no
yes
output all bits andfinish

Fig. 7.18. Flow chart for successive approximation analog-to-digital converter

7.9. Exercises

Exercise 7.1.

Make a circuit on a breadboard with a counter that counts every time a pushbutton is pressed. You can show the output as a binary number on LEDs, or as a decimal number on a 7-segment LED display if you prefer. Find the integrated circuits you need in the list of digital ICs, page 75.

Test the circuit with and without debouncing.

8. Microprocessors and microcontrollers

Figure 8.1 shows the basic construction of a computer. The central processing unit (CPU) is the brain of the computer. The CPU communicates with different kinds of memory and input and output devices. The different units are all connected through three buses called *address bus*, *data bus*, and *control bus*. A bus is a bundle of wires that connects multiple units.

A big computer has the CPU on a separate silicon chip and the RAM and other devices on a number of other chips. A microcontroller is a small computer that has everything on a single chip.

There are different kinds of memory. RAM stands for *random access memory*. This means that the CPU can read from the RAM and write to the RAM in random order. In principle, the RAM memory consists of a large number of flip-flops. The contents of the RAM memory is lost when the power is turned off.

ROM means read-only memory. The ROM may contain some important code that the computer needs for start-up. The contents of the ROM memory is permanent.

A newer type of memory is FLASH. FLASH memory is often used instead of ROM because the contents can be changed, and it is not lost when the power is turned off. FLASH memory cannot replace RAM because writing to FLASH is slower and more complicated. FLASH memory is used in telephones, cameras, USB memory sticks, microcontrollers, and solid-state hard discs.

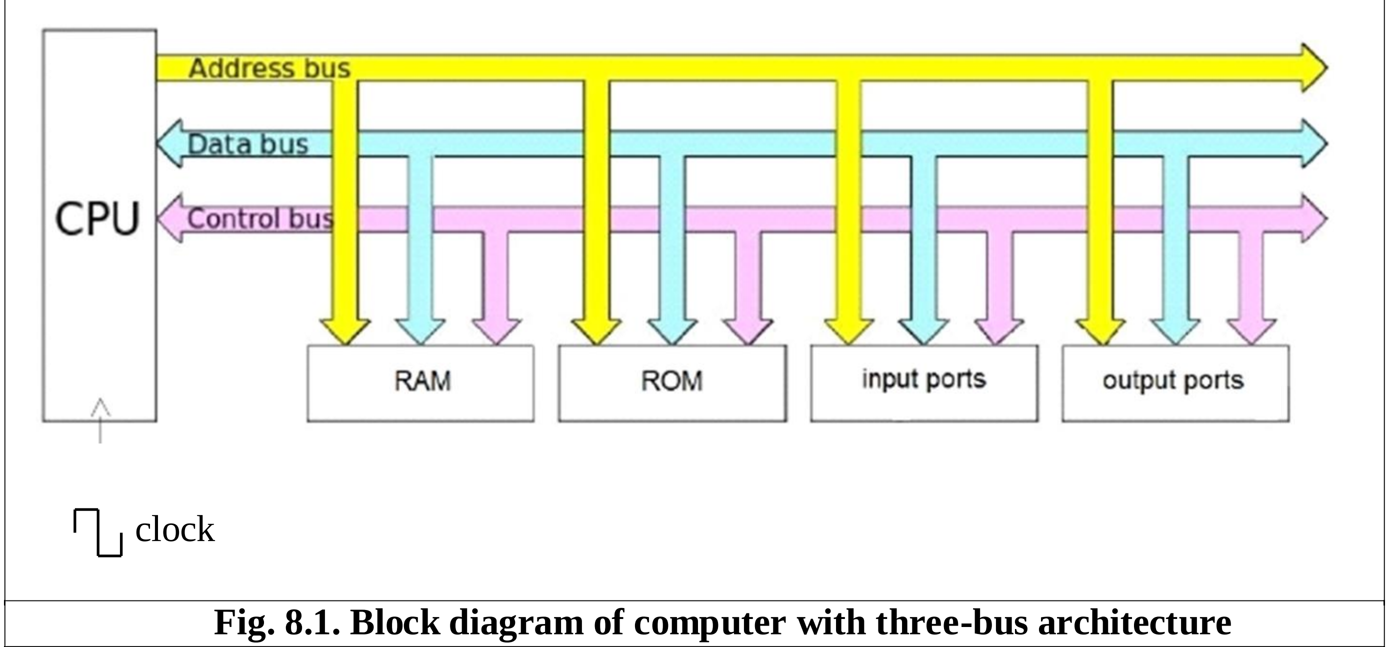


Fig. 8.1. Block diagram of computer with three-bus architecture

The CPU needs two kinds of data: (1) the program code and (2) the variables and other data that the program works on. The program code can be stored in RAM, ROM, or FLASH memory. The program data is stored in RAM memory.

The memory is organized into blocks containing eight bits each. A block of eight bits is called a *byte*. Each byte in the memory has an *address*. The first byte has address 0, the next byte has address 1, and so on. Small microcomputers have thousands of memory bytes. Bigger computers have millions, or even billions, of memory bytes.

The address of the byte we want to read or write is placed on the address bus. This is the yellow line on figure 8.1. The address bus consists of a number of wires to represent the address as a binary number. The number of wires in the address bus depends on how much memory you have. With 10 address wires, you can have 210 = 1024 different addresses and 1024 bytes of memory. 1024 bytes is called a kilobyte. With 20 address wires you can have 220 = 1048576 bytes = 1 megabyte. With 30 address wires you can have 230 = 1073741824 bytes = 1 gigabyte.

The program code is written in a programming language, for example C++, and then compiled. The compiler translates the C++ code into machine code. The machine code is a long list of simple machine instructions. Each machine instruction consists of a binary code telling the CPU what to do, for example to add two numbers. The machine instruction may also contain the address of the data that it is working on.

The CPU is a very big state machine that typically reads and executes one machine instruction for each clock cycle. The CPU consists of gates and D flip-flops just like the state machines we learned about in chapter 6.

The CPU is fetching the machine instructions, one by one, from the memory (RAM or ROM). It does this in the following way: It puts the address of the machine instruction on the address bus (yellow line
on figure 8.1). Then it puts a read signal on the control bus (pink line on figure 8.1). If one of the memory blocks recognizes the address as belonging to itself, it puts the value of the byte on the data bus (blue line). The CPU reads this byte, interprets it, and does whatever this instruction tells it to do.

The CPU can read program data from RAM memory in the same way. The CPU can also write program data to RAM memory. It does this by putting the address of the memory cell on the address bus in the same way as before. Then the CPU puts the value on the data bus and a write signal on the control bus.

Note that the data can go both ways on the data bus, while the address bus and the control bus go only one way, from the CPU to the memory and other devices. Only one unit at a time can put data on the data bus. All the other units must leave the data bus untouched. This is like in class when the teacher says the name of one student who is allowed to talk, while everybody else must keep silent. Each memory cell must have a unique address just as each student must have a unique name if you want reliable communication.

The data bus has at least eight wires so that it can read or write one byte at a time. It may have 16, 32, 64, or more wires so that it can read or write multiple bytes at the same time.

The control bus needs at least two wires, one for a read signal and one for a write signal.

8.1. Input ports and output ports

External devices such as screen, display, keyboard, mouse, hard disk, and network are connected to the data bus via input ports and output ports. Each input or output port has a unique address just like each byte-cell in the memory has a unique address.

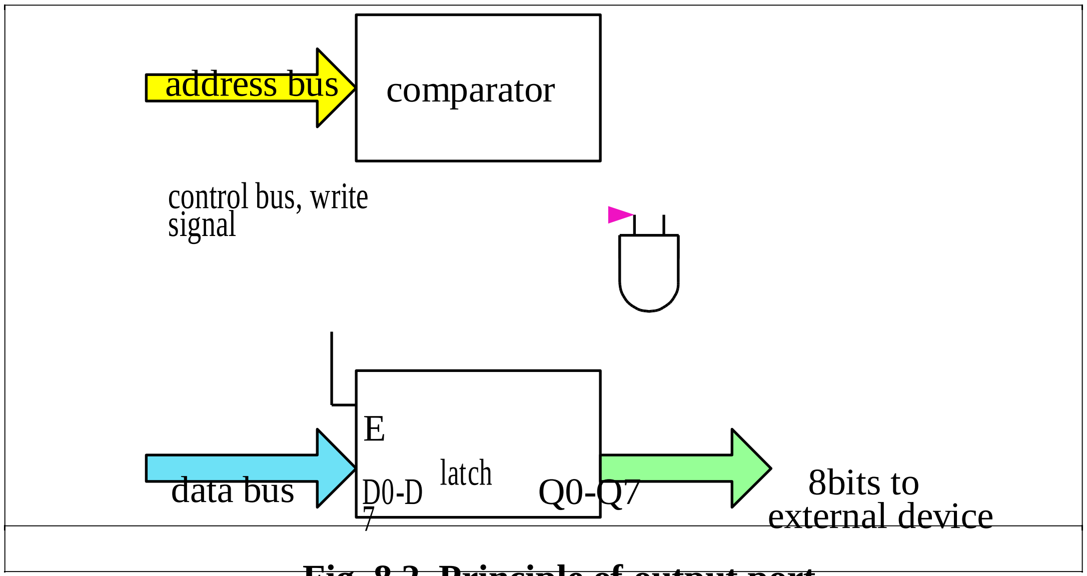


Fig. 8.2. Principle of output port

An output port contains a comparator that compares the fixed address of the port with the value on the address bus. An 8-bit latch stores the value from the data bus if the address is equal to the port address and there is a write signal on the control bus, as shown on figure 8.2.

An input port contains a so-called three-state buffer. This is a device where the output can be in one of three states: low, high, or disconnected. Each of the eight inputs from the external device goes through a three-state buffer before it is connected to the data bus. The three-state buffer is enabled when the value on the address bus is equal to the fixed address of the input port and there is a read signal on the control bus. The three-state buffer is disabled when the address is not matching. A three-state buffer is not disturbing the data bus when it is disabled – the output of the three-state buffer is simply disconnected so that the voltage on the data bus can be set high or low by some other device.

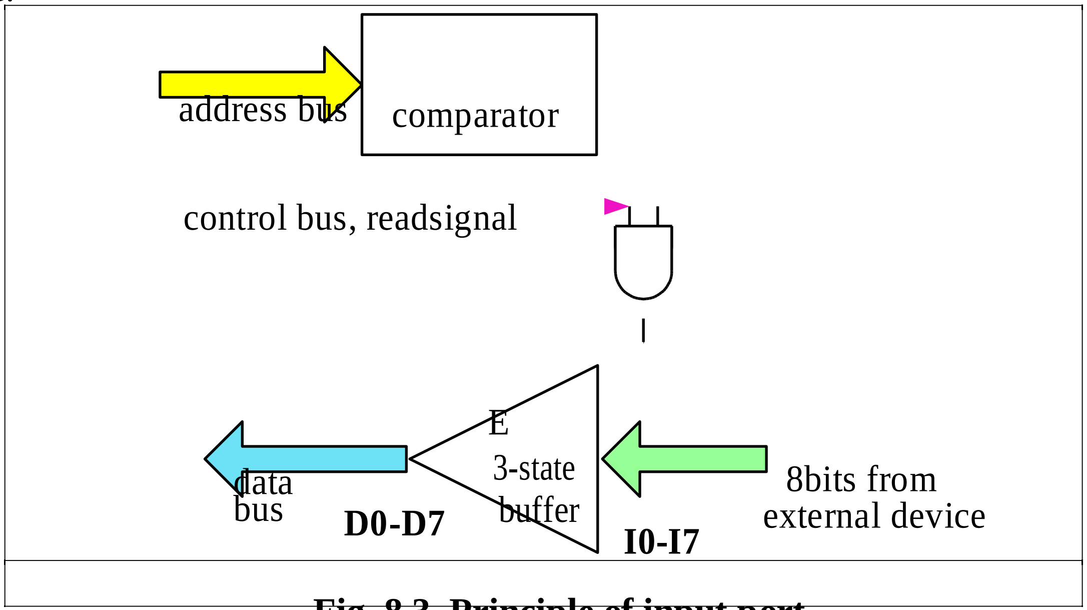


Fig. 8.3. Principle of input port

Figure 8.4 shows how a three-state buffer is constructed. It has a p-channel MOSFET and an n-channel MOSFET just like the inverter on figure 3.3 (page 19). The p-channel MOSFET will make the
output high by connecting it to V+ when its gate is low. This happens when the data input is high and

enable is high. The n-channel MOSFET will make the output low by connecting it to ground when its gate is high. This happens when the input is low and enable is high. Both MOSFETS are off when the enable signal is low.

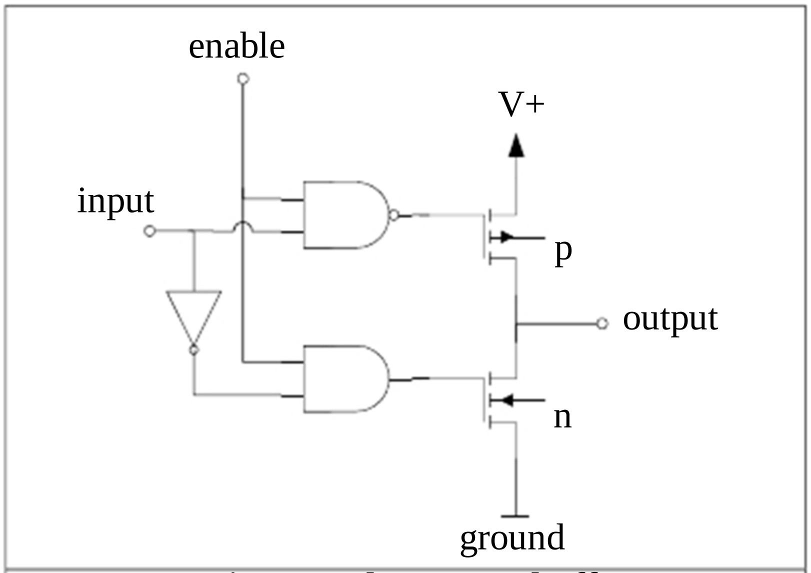


Fig. 8.4. Three-state buffer

The truth table for a three-state buffer looks like this:

input	enable	output
0	1	0
1	1	1
0	0	Z
1	0	Z

The 'Z' in the truth table means that the output is disconnected. This state is called 'high impedance'. The difference between 0 and Z is that the output is connected to ground in state 0 but not connected to anything in state Z.

8.2. Interrupt

If we press a key on a keyboard or we move a mouse we want to see the response immediately on the computer screen, even if the computer is busy doing something else. This problem can be solved with a mechanism called *interrupt*. One task is interrupted by an external event where the computer has to do some other task that has higher priority. It will return to the first task when the high-priority task has been completed.

It is very inefficient if the software code has to check all the time if a key has been pressed. Instead, computers have a hardware mechanism to support interrupts. This is illustrated in figure 8.5.

The instructions are executed one by one during the normal operation of the CPU. The sequence of program instructions in memory are shown as the green boxes on figure 8.5. An interrupt can happen at any time regardless of which instruction is being executed. The CPU does not execute the next instruction in the sequence when an interrupt has been detected. Instead, it jumps to another sequence of instructions called an interrupt service routine (yellow boxes on the figure). It remembers where it came from so that it can return and continue where it left when the interrupt service routine has finished. The interrupt service routine must save all registers that it uses and restore them to their original value before it returns to the main program so that all registers have the same value that they had before the interrupt.

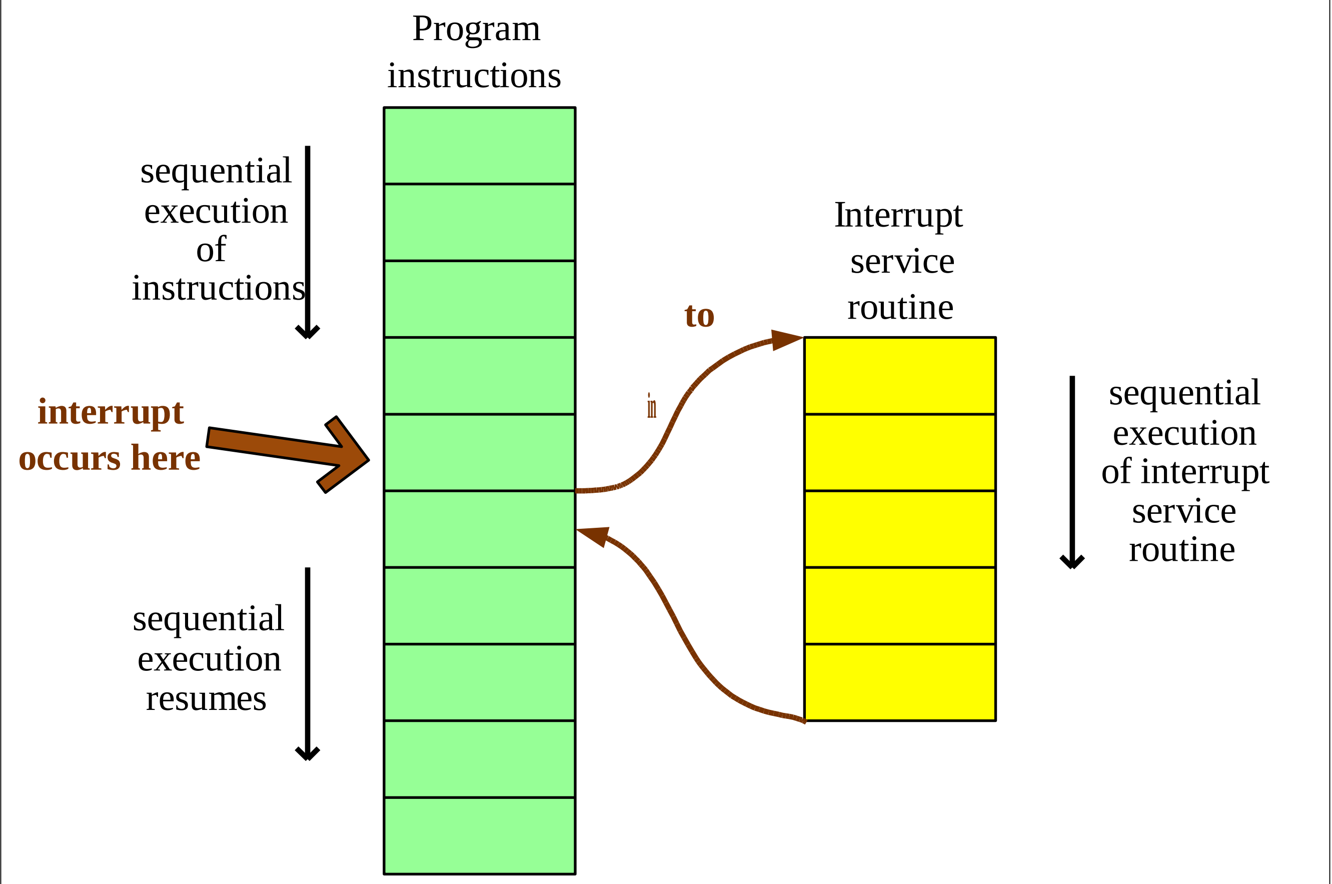

Program instructions
sequential execution of instructions
interrupt occurs here
sequential execution resumes
to
Interrupt service routine
sequential execution of interrupt service routine

Fig. 8.5. Principle of interrupt

We need a mechanism that allows the CPU to remember where it was interrupted so that it can return to the right place when the interrupt service routine has finished. The address of the instruction that it has to return to is stored in a piece of RAM memory called a *stack*. The stack stores information in a first-in-last-out basis. The first-in-last-out scheme is necessary in case the interrupt service routine is interrupted again by something else with a higher priority.

The stack can also be used for returning from function calls and for temporary storage of local data in a function.

9. Appendix A: List of digital integrated circuits

There are several families of digital integrated circuits:

4000 series: CMOS. 3–15 V supply. Output 0.5 mA.

74HC00 series: CMOS. 2–6 V supply. Output 5 mA. Faster than the 4000 series.

74LS00 series: TTL. 5V supply. This is an older series with asymmetric signal levels and higher power consumption. Do not use the TTL series unless you have a special reason to do so.

The 4000 series and 74HC00 series are compatible with each other. An output from a 4000 series circuit cannot drive an input of a 74LS00 series. An output from a 74LS00 series circuit must have a pull-up resistor if it is connected to an input of a CMOS circuit. The output of a 4000 series cannot drive a LED.

There are many other variants of the 74xx00 series with different letters in the name. Types with C in the name are CMOS types, all others are TTL types. The 74HCT00 series has CMOS technology but TTL signal levels. These are compatible with both the TTL and CMOS types.

Below is a list of selected digital ICs suitable for small projects.

Gates:

Gates per package	Inputs per gate	AND	NAND	OR	NOR	XOR
4	2	74HC08	74HC00	74HC32	74HC02	74HC86
3	3	74HC11	74HC10	74HC4075	74HC27	
2	4	74HC21	74HC20	74HC4072	74HC4002	
1	8		74HC30	74HC4078	74HC4078	

Buffers, inverters, Schmitt triggers, miscellaneous

Buffers per package	Type number	Description
6	74HC04	Hex inverter
6	74HC4050	Hex buffer
6	74HC4049	Hex inverting buffer
6	74HC14	Hex inverting Schmitt trigger
6	74HC367	Hex tri-state buffer
8	74HC244	Octal tri-state buffer
8	74HC243	Octal transceiver
4	74HC4066	Quad analog switch
6	4504	Hex voltage level converter

Decoders, encoders, multiplexers, arithmetic circuits

Number per package	Type number	Description
2	74HC139	Dual 1-of-4 decoder with inverted outputs
1	74HC138	1-of-8 decoder with inverted outputs
1	74HC238	1-of-8 decoder
1	74HC42	1-of-10 decoder with inverted outputs
1	74HC154	1-of-16 decoder with inverted outputs
1	74HC4511	BCD to 7-segment decoder, for common cathode
1	74LS247	BCD to 7-segment decoder, for common anode
1	74HC4543	BCD to 7-segment decoder, for LCD displays
4	74HC157	Quad 2-input multiplexer
2	74HC153	Dual 4-input multiplexer
1	74HC151	8-input multiplexer
1	74HC283	4 bit adder
1	74HC85	4 bit magnitude comparator
1	74HC688	8 bit magnitude comparator

Flip-flops and latches

Number per package	Type number	Description
2	74HC74	D Flip-flop with inverted set and reset
4	74HC175	D Flip-flop with inverted reset
6	74HC174	D Flip-flop with inverted reset
8	74HC273	D Flip-flop with inverted reset
4	74HC75	D-latch
8	74HC373	D-latch

Counters

Counter states	Type number	Description
2^{12}	74HC4040	12 stage ripple counter with asynchronous reset
2^{14}	74HC4060	14 stage ripple counter with RC oscillator and asynchronous reset
10	74HC4017	Counter with 10 decoded outputs and asynchr. reset
10, 100	74HC390	Dual BCD counter with asynchronous reset
10	74HC160	4 bit synchronous BCD counter with count enable, asynchronous reset and synchronous load
16	74HC161	4 bit synchronous binary counter with count enable, asynchronous reset and synchronous load
16	74HC163	4 bit synchronous binary counter with count enable, synchronous reset and synchronous load
10	74HC190	up/down BCD counter w. count enable and asyn. reset
16	74HC191	up/down binary counter w. count enable and asyn. reset

Shift registers

Number of bits	Type number	Description
8	74HC4094	Serial in parallel out shift register with output latch and 3-state output
8	74HC595	Serial in parallel out shift register with clocked output register and 3-state output
4	74HC194	4 bit universal bidirectional shift register with synchronous load
4	74HC195	4 bit universal shift register with synchronous load
8	74HC164	8-bit serial in parallel out shift register
8	74HC165	8-bit parallel in serial out shift register