Understanding Microprocessors

Authors acknowledgement

The authors wish to thank T. Jeffrey Burton Associates for their help in preparing this book.

UNWIN BROTHERS LIMITED
OLD WOKING SURREY

Understanding Microprocessors

By The Staff of Motorola Inc., Semiconductor Products Division

General Editor:
DANIEL QUEYSSAC

Based on the series "Understanding Microprocessors" published by *Electronics Weekly*

Authors acknowledgement

The authors wish to thank T. Jeffrey Burton Associates for their help in preparing this book.

UNWIN BROTHERS LIMITED
OLD WOKING SURREY

CONTENTS

			PAGE
Chapter	1	What is a microprocessor?	5
	2	Binary number systems (Parts 1, 2 & 3)	7
	3	Basic microprocessor system concepts	15
	4	Microprocessor hardware	18
	5	Random access memory	22
	6	Read only memory	25
	7	Input/output techniques	27
	8	Data transfer by DMA	29
	9	The interface	32
	10	Discussion of interrupts	36
	11	Solving an I/O problem	38
	12	Interfacing a keyboard	49
	13	Microprocessor programming	54
	14	Program execution	59
	15	Software support tools	69
	16	Support programs	72
	17	MPU programming languages	75
	18	Programming techniques	81
	19	Should you use an MPU?	84
	20	Selecting your MPU	87
	21	System design process	93
	22	System design considerations	99
	23	Memory technologies	102
	24	MPU market review	105
		Glossary of terms	111

CHAPTER 1

What is a microprocessor?

THE MICROPROCESSOR is perhaps the most important development the electronics industry has seen for at least the last decade. It was introduced to meet the need for a universal large scale integrated circuit caused by the fairly high cost and narrow application of most specialist LSI circuits.

Semiconductor manufacturers have learned how to incorporate thousands of transistors on a single chip of silicon. Unfortunately, before the microprocessor, as the number of transistors on a chip increased, the more dedicated that chip became to a particular application and the smaller the potential market became for it.

Since the cost of an integrated circuit is related inversely to the production volume, LSI circuits for other than common applications tended to be more expensive than necessary because of the restricted market.

VARIETY OF FUNCTIONS

The microprocessor, with its ability to perform a wide variety of different functions, is the answer. It can be obtained at low cost, because its almost unlimited range of applications makes volume production economical.

In use, the microprocessor can be coupled, via suitable interface circuits, to a wide variety of external devices which both provide its input signals and are controlled by its outputs.

The microprocessor, at the centre of this activity, responds to inputs and produces outputs in a manner determined wholly by a sequence of instructions which are stored in some form of memory connected to the microprocessor. This sequence of instructions is referred to as its program.

Immediately one is introduced to microprocessing concepts, parallels with minicomputers are inevitably drawn. Is the microprocessor on a par with the minicomputer? What are the basic differences? The answer to these questions can be fairly complex.

However, it is generally true to say that the MOS microprocessors on the market today operate in a similar way to minicomputers, but are slower and not as powerful in that the instruction sets are less comprehensive. The differences are, however, quickly diminishing.

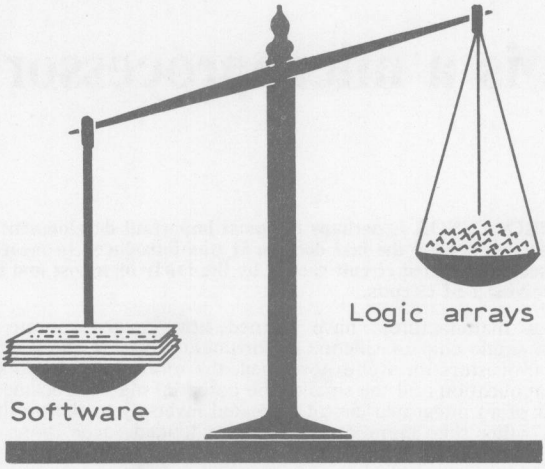
HARD-WIRED LOGIC

In general terms, the microprocessor is a replacement for hard-wired logic in most applications where hard-wired logic is still used and in applications where more complex logic could have been used to advantage if the cost had not been prohibitive.

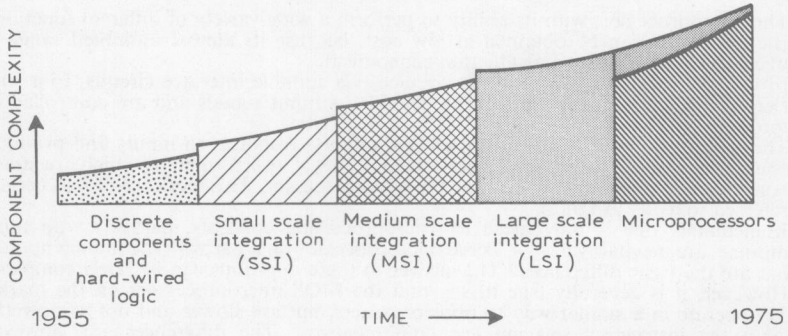
The main advantages offered by microprocessing systems are lower cost, fewer components, increased reliability and versatility. Existing systems can be modified at will by changing the program of instructions, often merely by exchanging a ROM device. However, do not be misled, program development is time consuming, and changes may not always be as simple as they sound.

Designing with a microprocessor involves exchanging logic arrays for software and there is an optimum balance between the two which provides the best performance in a given system most cost effectively. The systems designer using microprocessors requires both logic design and software expertise.

Fortunately, the qualities required of a good programmer are basically similar to those possessed by logic designers.



The optimum balance—between software and hardware.



The development of semiconductor technology from discrete components and hard-wired logic to microprocessors. The MPU represents a breakthrough in versatility at a decreasing cost per unit.

CHAPTER 2

Binary number systems

PART 1

THE MICROPROCESSOR breaks down the conventional divisions between software and hardware—the new definition is firmware—and presents the electronics engineer with the unfamiliar task of programming.

Programming forms a very important part in the development of a microprocessor based system, and it is essential that engineers intending to use a microprocessor are fully conversant with binary and related number systems.

EIGHT-BIT WORDS

The majority of microprocessing systems available today are based on a binary word that is eight bits long. In this series of articles we will use the eight-bit word (or byte, as it is called) as the norm; however, most of what is written also applies in principle to four and 16-bit machines.

To the microprocessor, a byte is merely a set of eight electrical signals, or logic levels. Each signal can have one or two values, or states.

The machine itself does not know whether these electrical states represent a binary, or any other kind of number.

The designer of the internal logic of the microprocessor has given particular bit patterns significance by the way he has organised the logic.

In his turn, the programmer can make the bytes represent anything he likes by the way in which he writes his program. For example, a byte, or pattern of eight bits, could represent:

1. A binary pattern between 00000000 and 11111111
2. A number between 0 and 255_{10} in natural binary code
3. A number between −128 and +127 in binary 2's complement
4. A number between +127 and −127 (signed binary)
5. A number between 00 and 99 (binary coded decimal)
6. An octal number between 000 and 377
7. A hexadecimal number between 00 and FF
8. A letter of the alphabet, the numerals 0 to 9 or punctuation marks (ASCII code)
9. Half a 16-bit memory address
10. An instruction
11. Whatever else the programmer wants it to represent

As far as the binary representation of numerical values is concerned, in microprocessing the programmer has a choice of natural binary, signed binary, 2's complement binary, and binary coded decimal.

Hexadecimal and octal can be described as shorthand methods of writing binary numbers or patterns. Each of the four main binary numerical systems mentioned above can be described by means of a series of short statements.

A *word* is a set of binary digits which are operated on collectively and which form one number.

A *byte* is a binary word and comprises eight binary digits, for example, 10101010 is a byte.

The *left-hand* bit, the most significant, is referred to as bit 7 (b_7).

The *right-hand bit*, the least significant, is referred to as bit 0 (b_0).

Intermediate bits are numbered between bit 1 to bit 6 (b_1 to b_6) from right to left. In *natural binary* a number is always positive.

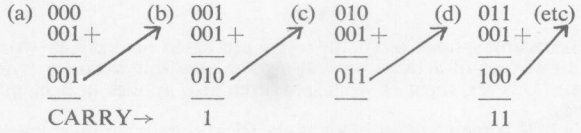
A *natural binary word* can have a numerical value of $2^n - 1$, where n is the number of bits, for example:

$$111 = 2^3 - 1 = 7_{10}$$
$$11111111 = 2^8 - 1 = 255_{10}$$

where 7_{10} and 255_{10} signify decimal numbers.

Therefore a byte (8-bits) can have any value between zero and 255_{10} in natural binary.

Counting can be described as repeatedly adding one unit. In natural binary counting proceeds as follows:



Explanation:

(a) $1 + 0 = 1$, $0 + 0 = 0$, $0 + 0 = 0$ Result 001

(b) $1 + 1 = 0$ carry 1,
$0 + 0 + 1$ carry $= 1$, $0 + 0 = 0$ Result 010

(c) $1 + 0 = 1$, $0 + 1 = 1$, $0 + 0 = 0$ Result 011

(d) $1 + 1 = 0$ carry 1,
$0 + 1 + 1$ carry $= 0$ carry 1,
$0 + 0 + 1$ carry $= 1$ Result 100

As in decimal addition, when the sum of two digits exceeds the capacity of a digit position, a one is carried to the next most significant position to be included in the next addition. In the following example, as:

$$8_{10} + 5_{10} = 3_{10} \text{ carry } 1_{10}$$

then $1_2 + 1_2 = 0_2$ carry 1_2

where 1_2 indicates a binary number

PART 2

WE CONTINUE by considering the basic arithmetic operations of addition and subtraction which are used to describe logic functions.

The rules of binary addition are illustrated in the first truth table (Table 1). All the possible combinations of input states are listed and the true output states for each combination are given.

TABLE 1

Rules of binary addition

Truth table for a single bit of a binary adder

X	+	Y	Carry IN[1]	=	Z	Carry OUT[2]
0	+	0	0	=	0	0
1	+	0	0	=	1	0
0	+	1	0	=	1	0
1	+	1	0	=	0	1
0	+	0	1	=	1	0
1	+	0	1	=	0	1
0	+	1	1	=	0	1
1	+	1	1	=	1	1

[1] This is the carry (IN) from the previous column.
[2] This is the carry (OUT) to the next more significant column.

X and Y are input states and Z is the output state.

The rules of binary addition are applied to the addition of two bytes below:

```
          00101101
          10110110 +
          --------
          11100011
          --------
Carry→     1111
```

In binary subtraction, as in decimal subtraction, if it is necessary to subtract a larger number from a smaller one in a particular column, then a digit has to be "borrowed" from the next more significant column, for example:

	Decimal	Binary
	21	10101
	14	01110
	---	-----
	07	00111
	---	-----
Borrow→	1	111

The rules of binary subtraction appear in the second truth table (Table 2). It is interesting to note that the result columns (Z) are identical for both subtraction and addition, although the "carry" and "borrow" bits generated are not identical.

TABLE 2

Truth table for a single bit of a binary subtractor

X	—	Y	Borrow IN[1]	=	Z	Borrow OUT[2]
0	—	0	0	=	0	0
1	—	0	0	=	1	0
0	—	1	0	=	1	1
1	—	1	0	=	0	0
0	—	0	1	=	1	1
1	—	0	1	=	0	0
0	—	1	1	=	0	1
1	—	1	1	=	1	1

[1] This is the borrow from the previous subtraction column
[2] This is the borrow from the next more significant column

So far this discussion has concentrated on positive binary numbers, but there are two commonly used binary notations that can be used to represent both positive and negative numbers. These are called 1's complement binary and 2's complement binary. Both systems use the left bit (b_7) of a word to indicate the sign of the number and the remaining bits to give the magnitude.

By convention, if the left-most bit is 0, the number is positive; if it is one, the number is negative. In 1's complement representation, changing the sign bit from 0 to 1 negates the whole number and its magnitude does not change:

$$\begin{aligned} 01101001 &= +105 \\ 11101001 &= -105 \\ 01111111 &= +127 \\ 11111111 &= -127 \end{aligned}$$

1's complement binary is hardly ever used by computer hardware because it requires relatively complex logic. Certain anomalies arise using 1's complementary binary, for example

$$\begin{aligned} &00000000 = \text{zero} \\ \text{but} \quad &10000000 = -\text{zero} \end{aligned}$$

This gives rise to a confusing situation.

Consider the following:

$$\begin{gathered} 00000000 = 0 \\ \text{subtract 1 from zero} \\ \text{giving } 11111111 = -1 \end{gathered}$$

The 8-bit positive limit is $01111111 = +127_{10}$ and the negative limit is $10000000 = -128_{10}$.

This is 2's complement representation. Counting, by repeatedly adding 1 to -128 (10000000) will give a correct result all the way through zero to $+127$ (01111111). The sign bit will automatically change as the result goes through zero. Equally, continuously subtracting 1 from $+127$ will give the correct result down to -128:

$$\begin{aligned} 01111111 &= +127 \\ 01111110 &= +126 \\ &\vdots \quad\quad \vdots \end{aligned}$$

$$
\begin{aligned}
00000001 &= +1 \\
00000000 &= 0 \\
11111111 &= -1 \\
11111110 &= -2 \\
&\vdots \\
10000000 &= -128
\end{aligned}
$$

In 2's complement representation the positive and negative limits to the magnitude of a given number of bits can he expressed as follows:

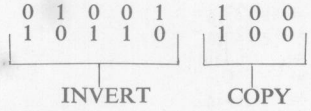
Positive limit is $(2^{n-1} - 1)$
Negative limit is $-(2^{n-1})$
where n is the number of bits.

It is very easy to change a positive number to its negative equivalent in 2's complement.

(1) Invert all bits (i.e. generate 1's complement)
(2) Add 1

$$
\begin{aligned}
& 00100101 = +37 \\
\text{Invert} \quad & 11011010 \\
\text{Add} \quad & \qquad 1 \\
& 11011011 = -37
\end{aligned}
$$

Another method is to take each bit in turn starting from the right. Write down each bit as it stands, up to and including the first "one" to be encountered, then invert all subsequent bits:



The rules governing the arithmetic manipulation of 2's complement numbers are identical with those for unsigned (positive) binary numbersin which it is assumed that the most significant bit is "0".

Since the most significant bit is implied, the permitted magnitude can range up to 255_{10} from zero. The design of the arithmetic logic unit is therefore much simplified.

PART 3

CALCULATIONS performed very simply by computers and microcomputers would present a tedious task to the engineer. The coding of numbers into binary, and the subsequent operation of the binary-coded numbers involves a great deal of tiresome arithmetic.

To counteract this problem, and to save effort, other binary-based numerical scales are very commonly used. These are the "octal" and the "hexadecimal" systems, with the "binary-coded decimal" notation as a compromise solution.

Instead of writing eight binary digits to represent the numbers operated on by the processor, it is possible to write three octal characters or two hexadecimal ones. Octal, as it suggests, is a number system with a base number 8, hexadecimal has a base of 16. Both systems save time and reduce errors which might otherwise be caused by a confusion of ones and noughts.

The table below illustrates how useful hexadecimal notation can be as a shorthand method of writing binary numbers.

$Binary_{(2)}$		$Decimal_{(10)}$		$Hexadecimal_{(16)}$
0000	=	0	=	0
0001	=	1	=	1
0010	=	2	=	2
0011	=	3	=	3
0100	=	4	=	4
0101	=	5	=	5
0110	=	6	=	6
0111	=	7	=	7
1000	=	8	=	8
1001	=	9	=	9
1010	=	10	=	A
1011	=	11	=	B
1100	=	12	=	C
1101	=	13	=	D
1110	=	14	=	E
1111	=	15	=	F

One hexadecimal digit represents a four digit (bit) binary word. Two hexadecimal digits represent a byte:

$01101011_2 = 6B_{16}$
$11111111_2 = FF_{16}$

Therefore:

$01111111_2 = 7F_{16} = +127_{10}$
$01111110_2 = 7E_{16} = +126_{10}$
$00000001_2 = 01_{16} = +1_{10}$
$00000000_2 = 00_{16} = 0_{10}$
$11111111_2 = FF_{16} = -1_{10}$
$11111110_2 = FE_{16} = -2_{10}$
$10000000_2 = 80_{16} = -128_{10}$

where the binary numbers are in 2's complement form. Counting in hexadecimal is straightforward:

$Decimal_{(10)}$	$Hexadecimal_{(16)}$
0	0
1	1
.	.
.	.
.	.
.	.
13	D
14	E
15	F
16	10
17	11
.	.
.	.
255	FF

Similarly, octal can be used as a shorthand version of binary coded numbers. A single octal number represents a three-digit binary number. Counting in octal proceeds as follows:

$Binary_{(2)}$	$Decimal_{(10)}$	$Octal_{(8)}$
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

Therefore

$01001001_{(2)} = 111_{(8)}$
$11111111_{(2)} = 377_{(8)}$

The microprocessor operates most efficiently with binary number systems. However, there are many occasions when it is required to operate with decimal numbers. This can be done using a notion called binary—coded decimal (BCD).

BCD is similar to hexadecimal, except the digits A to F are not used:

$Binary_{(2)}$	BCD	Hexa-decimal$_{(16)}$	$Decimal_{(10)}$
0000	0000	0	0
0001	0001	1	1
0010	0010	2	2
0011	0011	3	3
0100	0100	4	4
0101	0101	5	5
0110	0110	6	6
0111	0111	7	7
1000	1000	8	8
1001	1001	9	9
1011	—	A	10
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
1111	—	F	15

With a single byte it is possible to express a two-digit decimal number within the following limits:

$00000000(BCD) = 00_{(10)}$
$10011001(BCD) = 99_{(10)}$

Since the maximum capacity of BCD bytes is $99_{(10)}$ less than half of the information-carrying capacity of the byte is used (in natural binary, a byte can have a value up to 255).

BCD is, however, very useful and many microprocessors offer facilities for handling it. In fact, there is at least one microprocessor that deals exclusively with BCD numbers.

The size of the numbers which can be represented using the hexadecimal, octal and binary-coded decimal numbering systems is, however, limited to 255 for each byte. The resolution is also poor, equal to one part in 255 or approximately 0·39 per cent.

For greater accuracy, therefore, two or more bytes are used to represent numerical quantities and a mathematical technique known as "multiple precision" is adopted. A two-byte (16-bit) word has a resolution of more than one part in 6400.

CHAPTER 3

Basic microprocessor system concepts

THE MAIN object here is to describe how a microprocessor might be used to perform a particular task and to point out the similarities which exist between logic design and programming.

Every microprocessor has a memory in which a sequence of instructions can be stored. The memory, like a filing system, is divided into a series of locations and each location stores one byte. To enable these bytes to be retrieved, each location is allocated an address (like houses in a street) which does not change. Normally, in an eight-bit microprocessing system, 16 address bits (lines) are provided thereby providing coded access to $(2^{16} - 1)$ locations, i.e. $65{,}536_{10}$.

Binary address	Hexadecimal equivalent
00000000 00000000	0000
00000000 00000001	0001
00000000 00000010	0002
.	.
.	.
.	.
.	.
.	.
11111111 11111111	FFFF

The byte stored at a particular location, or address, may be an instruction (which causes the microprocessor to perform a task) or it may be data (perhaps a numerical value). The way in which the microprocessor uses a particular byte as an instruction or data will be described in a later article. Suffice to say now that a sequence of instructions is stored in the memory and can be retrieved by the microprocessor.

In traditional system design a block diagram can be drawn which shows the main components of the system and the way in which they are interconnected. Very often, in performing a particular task, many operations may be carried out simultaneously (in parallel).

With the microprocessor-based system, the approach is different because it can only perform one operation at a time. Therefore, the task to be undertaken must be represented as a series of operations in sequence.

Often, however, some tasks are performed by autonomous sub-systems such as input/output (I/O) controllers, while system co-ordination, control and the remaining tasks are performed by the microprocessor. The amount of additional circuitry used beyond the microprocessor sub-system is a matter for a trade-off between several factors; speed, hardware and software costs being the most important.