
Chapter 1

Computer System Fundamentals and Internal Data Transfer

A Summary...

Principles of microprocessor based systems. The data bus. Number systems, binary data representation, conversion and arithmetic. Boolean Algebra and laws. The bus architecture and parallel data transmission. Alpha-numeric representation and the ASCII / EBCDIC systems. The address bus system and addressing. Memory structure, operation and mapping. Microprocessor system internal master-slave relationships. Interrupt programming and internal system co-ordination and control.

Read This Chapter If...

- ◆ You are unfamiliar with the principles of modern computer architecture and you would like to come to terms with the basic concepts
- ◆ You need to refresh your memory on computer topics, terminology and acronyms that are in common use in data communications.

1.1 Microprocessor System Fundamentals

Microprocessor chips are the basic building blocks for nearly all of the "intelligent" control systems found in a modern manufacturing organisation. Smaller systems have a single microprocessor chip acting as the entire Central Processing Unit (CPU). This is typical of Personal Computers, Workstations and small industrial controllers. Larger computer-based systems use microprocessors as building blocks for entire boards, which may themselves act as CPUs or closed loop controllers.

Regardless of the architecture of intelligent systems, the principles by which communication occurs between a microprocessor chip and other associated semiconductor devices are essentially the same. We shall examine communications in a simple, single processor system to illustrate the key features involved.

Figure 1.1 shows a few of the fundamental elements in an intelligent system - the microprocessor chip, the memory chips and the data (communications) bus.

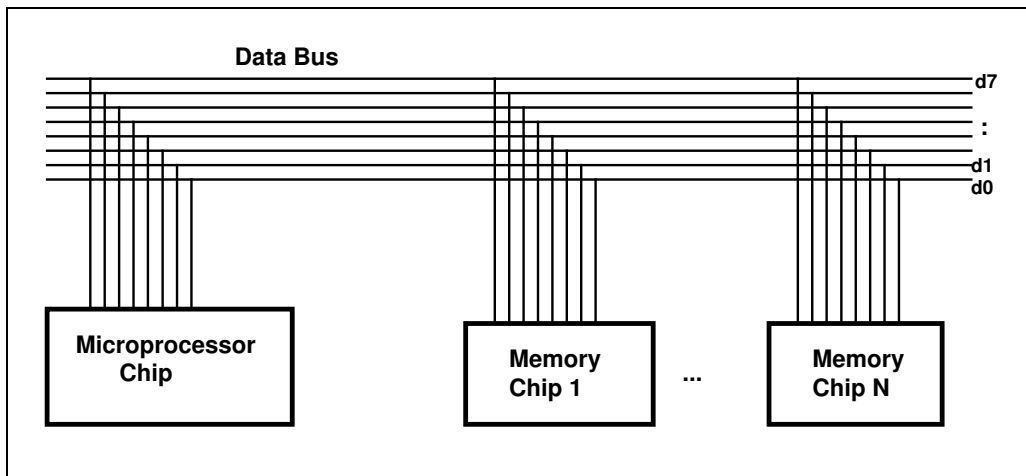


Figure 1.1 - Some Basic Elements in an "Intelligent System"

The data bus is a cluster of parallel conductors, which run as a group around the base-board (mother-board), on which the majority of chips (including the microprocessor) are mounted. The number of conductors (size of the data bus) depends upon the type of microprocessor chip on which the computer architecture is based. Typical data bus sizes may be 8, 16 or 32 bits.

The microprocessor chip can be envisaged as a machine that generates a number of internal voltage levels which together define the internal "state" of that machine. The internal state of the microprocessor changes at a rate determined by an external clock chip. The internal "state" voltage levels are decoded (by appropriate circuits) in order to:

- move data into or out of the microprocessor
- manipulate data within the microprocessor (add, subtract, etc.)
- move data from one internal storage location (register) to another.

Each cycle (tick) of the clock causes the microprocessor to jump from one internal state to another. The "next state" of the microprocessor is determined by a logical combination of its current internal state, together with the condition of all the various input lines connected to it. This is shown schematically in Figure 1.2. At each cycle of the system clock, a microprocessor executes only a very simple operation - read data in; write data out; store data in an internal register (storage location); add data contained in two internal registers; compare data in internal registers and so on.

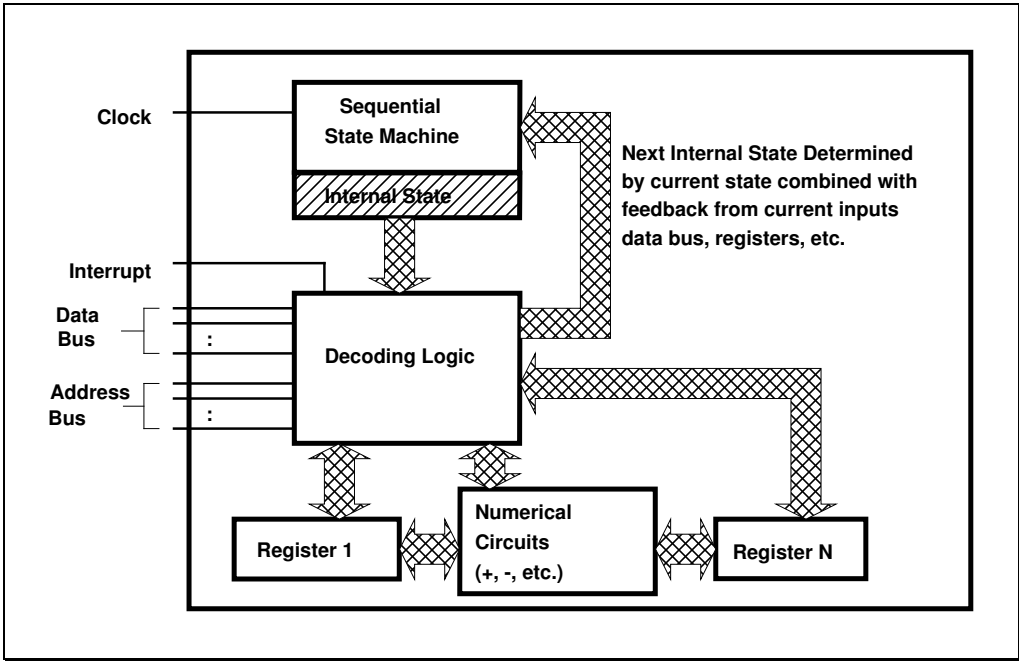


Figure 1.2 - Schematic of Building Blocks and Data Flow Within a Microprocessor Chip

Chips such as memory chips are considerably less sophisticated than the microprocessor and essentially have only two functions - read data in and write data out. Memory chips which perform both these functions are referred to as Random Access Memory (RAM) and those which can only write data out are referred to as Read Only Memory (ROM), because other devices can only read from them.

The architecture of semiconductor devices such as microprocessors, memory chips, etc., is based upon the use of only two voltages - low (false / off) or high (true / on). This is referred to as a "binary" or "base 2" system. Typically a voltage in the order of five volts is treated as high, and voltages of approximately zero are treated as low. The actual values depend upon the semiconductor technology used to fabricate a particular set of chips.

At any one point within a microprocessor chip, only the numbers 0 or 1 can be represented electronically at any instant in time. Similarly, the microprocessor's links to its outside world, the conducting, bus lines can also only have either a high or low voltage at any instant. Multiple conductors are therefore needed on a bus in order for the microprocessor to handle realistic numbers. A system with "n" conductors can therefore **directly** handle numbers ranging from:

$$0 \text{ to } (2^n - 1)$$

The voltage waveforms which appear, both on the bus lines of a system such as that in Figure 1.1, and within the semiconductor chips themselves, are approximated by square waves, as shown in Figure 1.3. Systems such as this are referred to as "digital", since we are only concerned with two conditions, high and low, rather than continuous (analogue) voltages.

In reality, few physical quantities such as voltage can switch from high to low and vice-versa in zero time. There is a finite time required in order for this electronic switching to occur and this transition period is in the order of pico-seconds. The voltage waveform of Figure 1.4 is more typical of what actually takes place in digital circuits.

Designers of digital hardware must take these transitions into account, but in more general discussions of computer systems and digital circuits we deal only with the idealised waveforms exemplified in Figure 1.3.

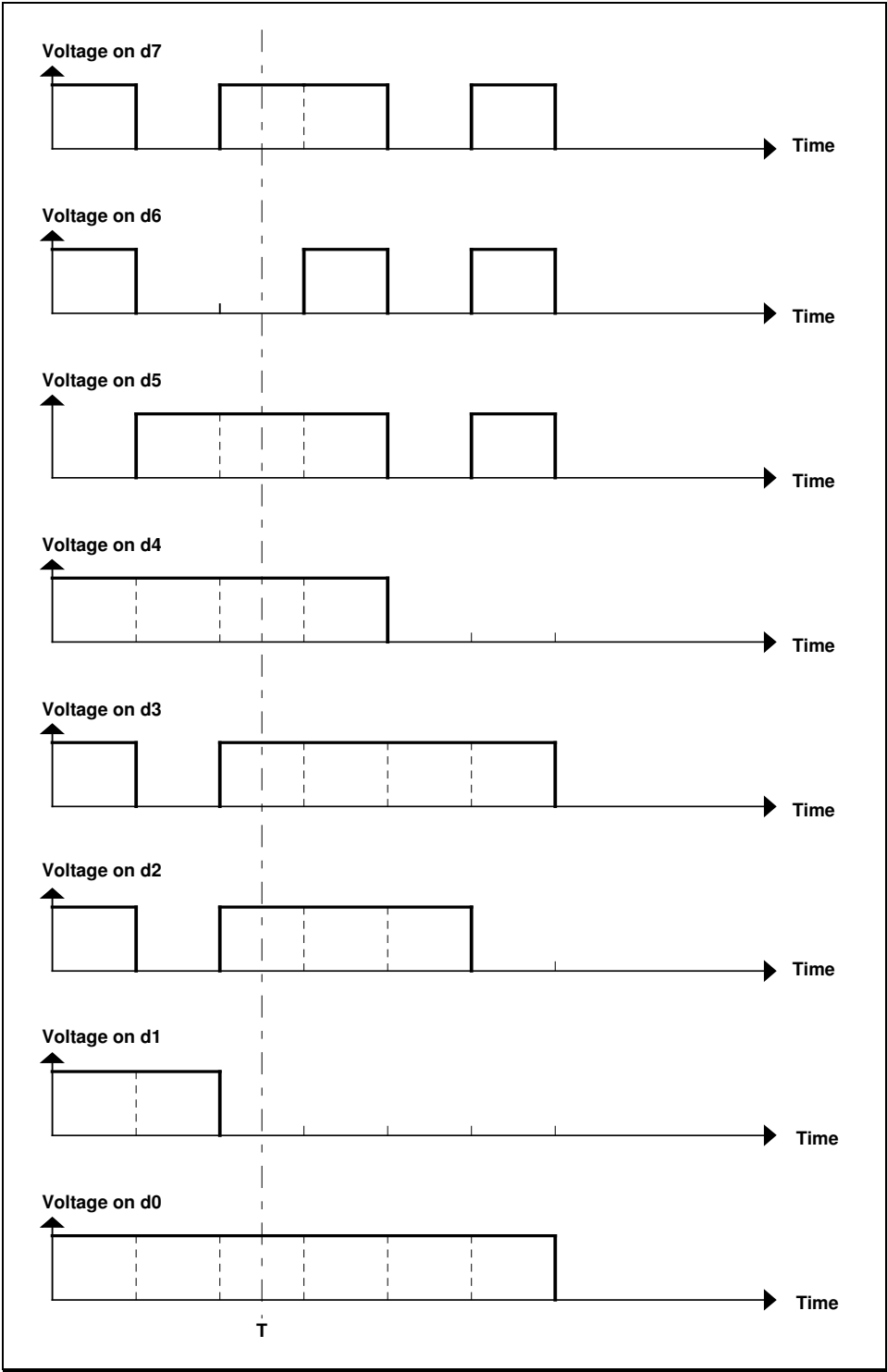


Figure 1.3 - Typical Data Bus Voltage Waveforms (approximate)

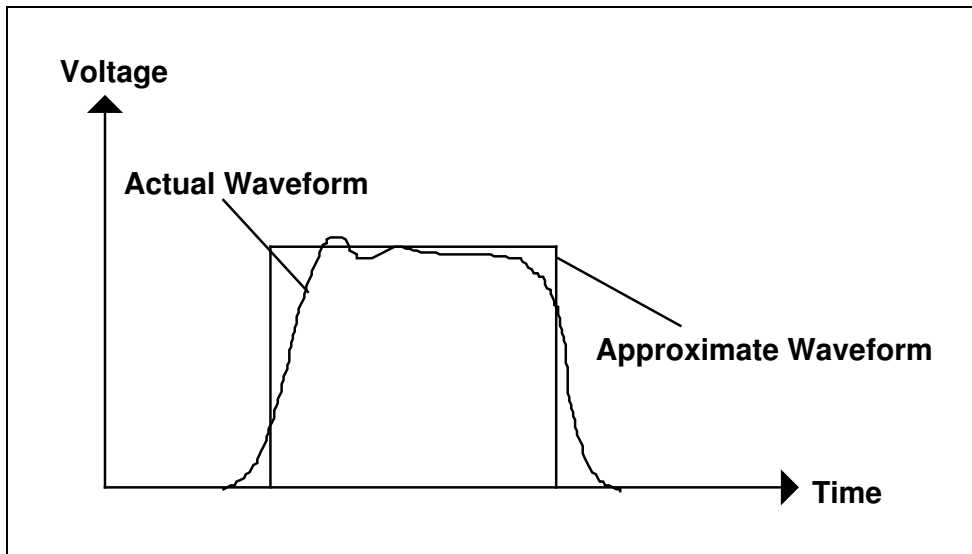


Figure 1.4 - Actual Switching from High-Low and Low-High

If we use the voltage waveforms as shown in Figure 1.3 as a mechanism for representation of the numbers 0 and 1, we can say that at a time "T", we have the following, "binary" number:

1 0 1 1 1 1 0 1

At any instant in time (neglecting transition periods), each point in a digital circuit represents one **binary digit**. This is abbreviated to the word "**bit**".

Since humans generally deal with the decimal (base 10) number system and alphabetic characters it should be evident that it is necessary for an intelligent digital system to continually convert data to and from the form used by humans. This is shown in Figure 1.5.

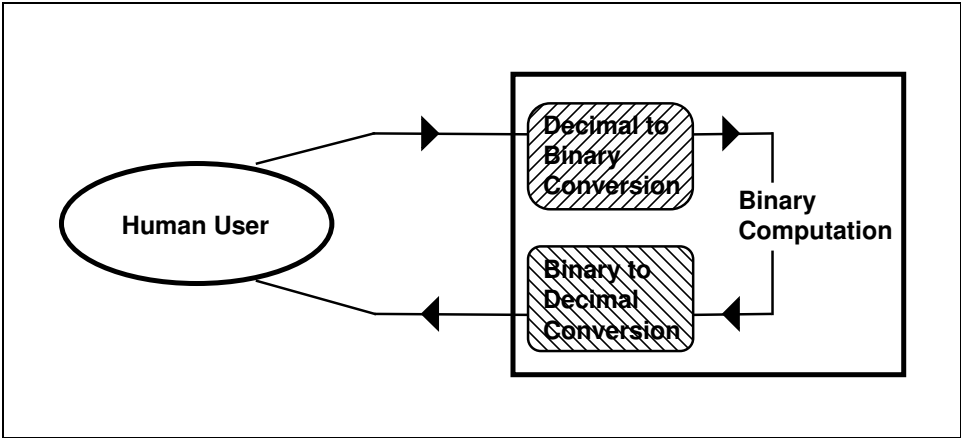


Figure 1.5 - Conversion from Computer to Human Form

1.2 Number Systems, Conversion and Arithmetic

In order to understand how data is transferred within computer systems, it is first necessary to understand how data is represented. We must therefore examine a range of different number systems. Firstly, in the decimal (or base 10) number system, the following is a count sequence:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
.									
.									
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109

Note the way in which we "carry" a digit each time we exceed the number "9". These representations are symbolic of the quantities that we actually wish to represent. For example, the decimal number 721 actually represents the following:

$$(7 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$$

Since the electronic circuitry in computer systems is designed to handle only two types of voltages (high and low), this representation is clearly inappropriate for our needs. There are however other commonly used number systems, which more closely relate to the needs of the computer, albeit indirectly. For example, the base 8, or "Octal" number system arises regularly. A count sequence in base 8 takes on the following form:

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77
100	101	102	103	104	105	106	107

The octal number 721 actually represents the following:

$$(7 \times 8^2) + (2 \times 8^1) + (1 \times 8^0)$$

which is equal to decimal 465 and not decimal 721.

When working with a range of different number systems, it is common practice to subscript numbers with the base of the number system involved. For example, we can validly write the following expression:

$$721_8 = 465_{10}$$

Another number system that is commonly used with computer systems is the base 16 or hexadecimal number system. Since we do not have enough of the ordinary numerals (0..9) to represent 16 different numbers with a single symbol, we "borrow" the first six letters of the alphabet (A..F). A count sequence in base 16 then takes on the following form:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
⋮															
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

To similarly convert the hexadecimal number 721 to decimal:

$$\begin{aligned} 721_{16} &= (7 \times 16^2) + (2 \times 16^1) + (1 \times 16^0) \\ &= 1825_{10} \end{aligned}$$

Finally we move on to the number system most closely related to the architecture of computer systems themselves, the binary number system, in which we can only count from 0 to 1 before performing a "shift" operation. The following is a base 2 count sequence:

0	1								
10	11								
100	101	110	111						
1000	1001	1010	1011	1100	1101	1110	1111		

If we look again at Figure 1.3, we can now see that the number represented by the voltage waveforms at time "T" is:

$$\begin{aligned} 10111101_2 &= \\ (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 189_{10} \end{aligned}$$

We have now seen that it is a relatively straightforward task to convert numbers from different bases to their decimal (base 10) equivalents. However it is also possible to convert from base 10 numbers into different number systems through a process of long division. In order to do this, the original decimal number is repeatedly divided by the new base (to which we wish to convert) and the remainders of each division are stored. The process is repeated until the original number is diminished to zero. The remainders then form the representation of the decimal number in the new base. This sounds complex, but in essence is relatively straightforward.

For example, if we wish to convert the decimal number 189 into its binary representation, the following long division quickly achieves the result:

2	189	1	Low order Remainder
	94	0	
	47	1	
	23	1	
	11	1	
	5	1	
	2	0	
	1	1	High order Remainder
	0		

Therefore $189_{10} = 10111101_2$ as proven earlier.

Conversion from the binary number system to the octal number system is a simple task, since each group of three bits directly represents an octal number. Binary numbers are partitioned into groups of 3 bits (binary digits), starting from the low order bits. Then each group of three digits is individually converted into its octal equivalent. For example, to convert the binary number 1011011110111 to its octal equivalent, the following procedure is used:

1	011	011	110	111
1	3	3	6	7

Therefore $1011011110111_2 = 13367_8$.

Conversion from the binary number system to the hexadecimal number system is similar to the binary-octal conversion, except that binary digits are placed into groups of four (since 4 bits represent 16 combinations). Each group of four is then individually converted into its hexadecimal equivalent. For example, to convert the binary number 1011011110111 to its hexadecimal equivalent, the following procedure is used:

1 0110 1111 0111

1 6 F 7

Therefore $1011011110111_2 = 16F7_{16}$.

Octal and hexadecimal numbers can also be readily transformed into their binary representation, simply by converting each digit individually to its equivalent 3 or 4 bit representation. This is the reverse operation to that shown in the previous two examples.

You should now observe that we have a simple and direct mechanism for converting from octal and hexadecimal numbers to binary, but that in order to convert from decimal to binary we need to perform the long division calculation, shown previously. In order to establish an analogous, direct relationship between binary and decimal, another number representation is also in use. This is referred to as the Binary Coded Decimal or BCD system.

In the BCD system, each decimal digit is represented in binary by four bits. For example, the BCD equivalent of the number 721 is given by:

0111 0010 0001

This is similar to the relationship between hexadecimal and binary, except that certain bit combinations can never occur, since the BCD system uses 4 digits (with 16 combinations) in order to represent the ten decimal digits, 0 to 9. Strictly speaking, BCD should not be regarded as a number system, but rather as a mechanism for directly converting human (decimal) input into an electronically usable binary form. It is most commonly used at a human to computer interface. For example, if a person pushes a number 7 (say) on a simple key-pad, then the appropriate voltages (low, high, high, high) can be generated.

To summarise the various number representations, most commonly associated with computers, Table 1.1 shows how each of the number systems represents the decimal numbers from 0 to 20.

<i>Decimal</i>	<i>Hexadecimal</i>	<i>Octal</i>	<i>Binary</i>	<i>BCD</i>
0	0	0	00000000	0000 0000
1	1	1	00000001	0000 0001
2	2	2	00000010	0000 0010
3	3	3	00000011	0000 0011
4	4	4	00000100	0000 0100
5	5	5	00000101	0000 0101
6	6	6	00000110	0000 0110
7	7	7	00000111	0000 0111
8	8	10	00001000	0000 1000
9	9	11	00001001	0000 1001
10	A	12	00001010	0001 0000
11	B	13	00001011	0001 0001
12	C	14	00001100	0001 0010
13	D	15	00001101	0001 0011
14	E	16	00001110	0001 0100
15	F	17	00001111	0001 0101
16	10	20	00010000	0001 0110
17	11	21	00010001	0001 0111
18	12	22	00010010	0001 1000
19	13	23	00010011	0001 1001
20	14	24	00010100	0010 0000

Table 1.1 - Representation of Decimal Numbers from 0 to 20

Numbers from different bases can be dealt with arithmetically in the same way as decimal numbers, except that a "shift" or "carry" has to occur each time a digit equals or exceeds the base value. The following are simple examples of addition, subtraction, multiplication and division using the binary number system:

- (i) Addition of 11101_2 and 1011_2

$$\begin{array}{r}
 1111 \text{ (Carry)} \\
 11101 \\
 + 1011 \\
 \hline
 101000 \\
 \hline
 \end{array}$$

- (ii) Subtraction of 1011_2 from 11101_2

$$\begin{array}{r}
 11101 \\
 - 1011 \\
 \hline
 10010 \\
 \hline
 \end{array}$$

- (iii) Multiplication of 11101_2 by 1011_2

$$\begin{array}{r}
 11101 \\
 \times 1011 \\
 \hline
 11101 \\
 111010 \\
 11101000 \\
 \hline
 100111111 \\
 \hline
 \end{array}$$

- (iv) Division of 11101_2 by 1011_2

$$\begin{array}{r}
 1011 \overline{) 11101} \\
 \underline{10110} \\
 10100011
 \end{array}$$

1.3 Representation of Alpha-numerics

It should be clear from the discussions of 1.1 and 1.2 that microprocessor based systems (and indeed digital systems) can only understand voltage waveforms which represent bit streams. They have no capacity for a direct interpretation of the alpha-numeric characters which humans use for communication.

In section 1.2, the Binary Coded Decimal system was cited as a means by which numbers, entered on a simple keypad, could be directly and electronically represented in a computer. This is however very restrictive as only 16 different numeric characters can be represented by a 4 bit scheme (and only 10 combinations are actually used in 4 bit BCD). In order to represent all the upper and lower case alphabetic characters on a typical computer keyboard, plus symbols, carriage-returns, etc., it is necessary to use strings of 7 or 8 bits, which then provide enough combinations for 128 or 256 alpha-numeric characters.

Two specifications for the bit patterns representing alpha-numeric characters are in common use. These are the 7 bit **ASCII** (American Standard Code for Information Interchange) and the 8 bit **EBCDIC** (Extended Binary Coded Decimal Interchange Code) systems. The ASCII system is by far the more prolific of the two specifications and it is used on the majority of Personal Computers. The EBCDIC system is used predominantly in a mainframe (notably IBM) computer environment.

The ASCII character set is listed in Table 1.2. This table shows each character beside its hexadecimal ASCII value, which explicitly defines the bit pattern representation for each character. For example, the character 'X' has the hexadecimal ASCII value of "58" that translates to a bit pattern of:

5	8
0101	1000

The corresponding EBCDIC hexadecimal values are also provided beside each character for comparison. Note that the EBCDIC system uses an 8 bit representation and therefore provides a much larger character set than the ASCII system. However some of the bit patterns in the EBCDIC system are unused.

CHAR	ASCII Value	EBCDIC Value	CHAR	ASCII Value	EBCDIC Value	CHAR	ASCII Value	EBCDIC Value
NUL	00	00	+	2B	4E	V	56	E5
SOH	01	01	,	2C	6B	W	57	E6
STX	02	02	-	2D	60	X	58	E7
ETX	03	03	.	2E	4B	Y	59	E8
EOT	04	37	/	2F	61	Z	5A	E9
ENQ	05	2D	0	30	F0	[5B	4B
ACK	06	2E	1	31	F1	\	5C	E0
BEL	07	2F	2	32	F2]	5D	5B
BS	08	16	3	33	F3	^	5E	--
HT	09	05	4	34	F4	_	5F	DF
LF	0A	25	5	35	F5	`	60	--
VT	0B	0B	6	36	F6	a	61	81
FF	0C	0C	7	37	F7	b	62	82
CR	0D	0D	8	38	F8	c	63	83
SO	0E	0E	9	39	F9	d	64	84
SI	0F	0F	:	3A	7A	e	65	85
DLE	10	10	;	3B	5E	f	66	86
DC1	11	11	<	3C	4C	g	67	87
DC2	12	12	=	3D	7E	h	68	88
DC3	13	13	>	3E	6E	i	69	89
DC4	14	3C	?	3F	6F	j	6A	91
NAK	15	3D	@	40	7C	k	6B	92
SYN	16	32	A	41	C1	l	6C	93
ETB	17	26	B	42	C2	m	6D	94
CAN	18	18	C	43	C3	n	6E	95
EM	19	19	D	44	C4	o	6F	96
SUB	1A	3F	E	45	C5	p	70	97
ESC	1B	27	F	46	C6	q	71	98
FS	1C	22	G	47	C7	r	72	99
GS	1D	--	H	48	C8	s	73	A2
RS	1E	35	I	49	C9	t	74	A3
US	1F	--	J	4A	D1	u	75	A4
SP	20	40	K	4B	D2	v	76	A5
!	21	5A	L	4C	D3	w	77	A6
"	22	7F	M	4D	D4	x	78	A7
#	23	7B	N	4E	D5	y	79	A8
\$	24	5B	O	4F	D6	z	7A	A9
%	25	6C	P	50	D7	{	7B	C0
&	26	50	Q	51	D8		7C	6A
'	27	7D	R	52	D9	}	7D	D0
(28	4D	S	53	E2	~	7E	A1
)	29	5D	T	54	E3	DEL	7F	07
*	2A	5C	U	55	E4			

Table 1.2 - ASCII and EBCDIC Character Representation

The ASCII system only uses 7 bits to represent characters with values from 0 to 7F (127) but most computers work with 8 bit units. In order to utilise the high order bit, an extended ASCII character set, using all 8 bits, displays special symbols on Personal Computers (PCs), but unfortunately there is no uniformity in definition. Frequently, PC software packages (such as ASCII-based word-processors) take advantage of the spare high-order bit to store additional character information such as bolding, underlining, etc.

The choice of bit patterns to represent characters and numerics is essentially an arbitrary one. For example, in both the ASCII and the EBCDIC system, the number characters '0' to '9' are not represented by their equivalent binary values. In ASCII, the character '0' is represented by hexadecimal 30, which has a bit pattern of 00110000 and so on.

The first 32 ASCII characters in the set are not defined as "printable characters", although they do display as symbols on some computers. These special characters are predominantly used for communications purposes and will have relevance to our discussions in later chapters. Table 1.3 lists the values, together with their names and common abbreviations. Some of the characters in this set (such as carriage return) can be generated by a single key-stroke on an ASCII keyboard. Others can be generated by holding down the control (CTRL) key and hitting the corresponding letter listed in the table.

We summarise our discussions by saying that within the computer system itself, there is no direct recognition of alpha-numeric characters, but only of the bit strings (patterns) which are chosen to represent them. These bit patterns are arbitrarily defined, with the most common definition being the ASCII code system.

<i>HEX VALUE (ASCII)</i>	<i>NAME</i>	<i>ABBREVIATED NAME</i>	<i>KEY CODE</i>
00	NULL	NULL	CTRL @
01	START OF HEADER	SOH	CTRL A
02	START OF TEXT	STX	CTRL B
03	END OF TEXT	ETX	CTRL C
04	END OF TRANSMISSION	EOT	CTRL D
05	ENQUIRY	ENQ	CTRL E
06	ACKNOWLEDGE	ACK	CTRL F
07	BELL	BEL	CTRL G
08	BACK SPACE	BS	CTRL H
09	HORIZONTAL TAB	HT	CTRL I
0A	LINE FEED	LF	CTRL J
0B	VERTICAL TAB	VT	CTRL K
0C	FORM FEED	FF	CTRL L
0D	CARRIAGE RETURN	CR	CTRL M
0E	SHIFT OUT	SO	CTRL N
0F	SHIFT IN	SI	CTRL O
10	DATA LINE (LINK) ESCAPE	DLE	CTRL P
11	DEVICE CONTROL 1 (XON)	DC1	CTRL Q
12	DEVICE CONTROL 2	DC2	CTRL R
13	DEVICE CONTROL 3 (XOFF)	DC3	CTRL S
14	DEVICE CONTROL 4	DC4	CTRL T
15	NEGATIVE ACKNOWLEDGE	NAK	CTRL U
16	SYNCHRONOUS IDLE	SYN	CTRL V
17	END OF TRANSMIT BLOCK	ETB	CTRL W
18	CANCEL	CAN	CTRL X
19	END OF MEDIUM	EM	CTRL Y
1A	SUBSTITUTE	SUB	CTRL Z
1B	ESCAPE (ESC)	ESC	CTRL [
1C	FILE SEPARATOR	FS	CTRL \
1D	GROUP SEPARATOR	GS	CTRL]
1E	RECORD SEPARATOR	RS	CTRL ^
1F	UNIT SEPARATOR	US	CTRL _

Table 1.3 - Special Functions of the first 32 ASCII Characters

1.4 Boolean Algebra

Boolean Algebra was named after the mathematician George Boole (1815-64) and is the simplest means by which we can convert human reasoning and tautology into a mathematical and electronic form for computation. Boolean Algebra deals with the logic of the binary numbers, "0" and "1" or the states "TRUE" and "FALSE".

The means by which we now generate electronic circuits on Small Scale Integrated (SSI), Medium Scale Integrated (MSI), Large Scale Integrated (LSI) and Very Large Scale Integrated (VLSI) semiconductors, to provide Boolean logic is beyond the scope of this book. Suffice to say that these are well-established technologies, which are in principle based on semiconductor switching of voltages from high to low and vice versa. The basic circuits used to provide Boolean logic within computer systems are referred to as "logic gates".

Table 1.4 shows the symbols for some common Boolean gates, together with their Boolean Algebra functions. Beside each gate is a table, generally referred to as a "truth table", which illustrates how the outputs from each logic circuit are related to the inputs. Note that the Boolean Operators "." and "+" do not represent multiplication and addition. Rather, they represent the logical functions "AND" and "OR" respectively. This can become a very confusing point, particularly where we use Boolean Algebra to make circuits that add or multiply binary numbers together. Note also that the symbolic circuit representation for logic gates, shown in table 1.4, is one of the commonest forms in use. However, it is not a universal standard and the symbolic representation of logic gates varies from country to country.

Using Boolean logic, we can convert statements such as:

"If both A and B are true or if C is true while D is false, then E will be true,"

into Boolean expressions, which can be handled by computers. Using gates such as those shown in table 1.4, we can effectively build simple circuits to provide the kind of logic that we require. The following is the Boolean representation of the above tautology:

$$E = (A.B) + (C.\bar{D})$$

In computer-based systems, Boolean Algebra is used at a number of different levels, ranging from hardware design right through to software design (programming). For example, in a Pascal computer program, we could implement the previous expression with the following syntax:

```
IF (A AND B) OR (C AND NOT D) THEN
  E := TRUE
ELSE
  E := FALSE
```

At a hardware level, microprocessor chips have built-in Boolean logic and can directly handle logical expressions internally. We therefore have a mechanism that can provide computers with a small measure of "human" reasoning.


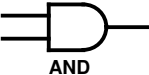
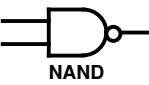
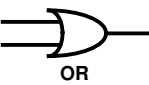
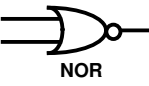
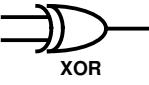
LOGIC GATE	BOOLEAN LOGIC	TRUTH TABLE			FUNCTION
		X	Y	Z	
 Inverter	Z is NOT X	0 1		1 0	$Z = \overline{X}$
 AND	Z is X AND Y	0 0 1	0 1 1	0 0 1	$Z = X.Y$
 NAND	Z is NOT (X AND Y)	0 1 1	0 1 1	1 1 0	$Z = \overline{X.Y}$
 OR	Z is X OR Y (Inclusive OR)	0 1 1	0 1 1	0 1 1	$Z = X + Y$
 NOR	Z is NOT (X OR Y)	0 1 1	0 1 1	1 0 0	$Z = \overline{X + Y}$
 XOR	Z is either X OR Y but not both (Exclusive OR)	0 1 1	0 1 1	0 1 0	$Z = X \oplus Y$

Table 1.4 - Common Boolean Logic Gates and Representation

Boolean expressions can often be unnecessarily complex. In situations where we are designing electronic circuits, from gate chips, to provide Boolean logic, we clearly need to minimise the number of chips and gates required. When we generate computer programs, unnecessarily complex expressions can slow down the operation of programs and decrease the legibility of the program code. For these reasons we need to establish the simplest form of a Boolean expression before committing it to an implementation phase.

There are a number of laws and postulates associated with Boolean Algebra. These are listed in Table 1.5. They are used as the basis for the simplification of Boolean expressions. A truth table can be constructed for each of the laws shown in Table 1.5 in order to verify that they are correct. Using these laws and postulates, we can now take a complex expression such as:

$$Z = \overline{\overline{A.B} \oplus C} \cdot (A \oplus B) \cdot D$$

and simplify it in the following manner:

$$\begin{aligned} Z &= \overline{\overline{A.B} \oplus C} \cdot (A \oplus B) \cdot D \\ &= \overline{A.B.C} \cdot (A \oplus B) \cdot D && \text{(By De Morgan's Theorem)} \\ &= A.B.\overline{C.D} \cdot (A \oplus B) && \text{(By law of commutation)} \\ &= A.B.C.D.A \oplus A.B.C.D.B && \text{(By law of association)} \\ &= A.A.B.\overline{C.D} \oplus A.B.B.\overline{C.D} && \text{(By law of commutation)} \\ &= \overline{A.B.C.D} \oplus \overline{A.B.C.D} && \text{(By law of tautology)} \\ &= \overline{A.B.C.D} && \text{(By law of tautology)} \\ &= \overline{A} \oplus \overline{B} \oplus \overline{C} \oplus \overline{D} && \text{(By De Morgan's theorem)} \end{aligned}$$

If we were designing a circuit or piece of software to achieve a given Boolean logic, then reduction techniques based on Boolean laws have the potential to simplify a design. However, not all complex expressions can be simplified and the technique shown above doesn't tell us when we have achieved the simplest form of an expression. For this reason a number of more systematic techniques (such as Karnaugh mapping) are also in use, but these are beyond the scope of this book.

Boolean algebra is introduced at this point because Boolean logic is the cornerstone of digital circuitry and structured program development. These elements will be of importance in later chapters, in the sense that they relate to data communications.

<i>Postulates of Boolean Algebra</i>		
<p>All variables must have either the value 0 or 1. If the value of a variable is not zero then it must be one and vice-versa. The following rules apply:</p>		
<i>OR</i>	<i>AND</i>	<i>NOT</i>
$1 + 1 = 1$	$1 \cdot 1 = 1$	$\overline{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\overline{0} = 1$
<i>Boolean Laws of Combination</i>		
$A \cdot B = B \cdot A$	(Laws of Commutation)	
$A + B = B + A$		
$A + B + C = A + (B + C)$	(Laws of Association)	
$(A \cdot B) \cdot C = A \cdot (B \cdot C)$		
$A \cdot (B + C) = A \cdot B + A \cdot C$	(Laws of Distribution)	
$A + A = A$	(Laws of Tautology - Idempotent Rule)	
$A \cdot A = A$		
$A + 1 = 1$		
$A \cdot 1 = A$		
$A + 0 = A$		
$A \cdot 0 = 0$		
$A + \overline{A} = 1$		
$A \cdot \overline{A} = 0$		
$A + \overline{A \cdot B} = A + B$		
$\overline{\overline{A}} = A$	(Laws of Double Complementation)	
$\overline{\overline{A + B}} = A + B$		
$\overline{\overline{A \cdot B}} = A \cdot B$		
$\overline{A \cdot B} = \overline{A} + \overline{B}$	(DeMorgan's Theorems)	
$\overline{A + B} = \overline{A} \cdot \overline{B}$		

Table 1.5 - Fundamental Principles of Boolean Algebra

1.5 Microprocessor System Program Execution & Communication

Now that we have examined how data is represented within a digital computer system, we can examine the mechanism for program execution within a microprocessor based system. We can also examine the communication that occurs between the various elements of the system.

A microprocessor chip responds to certain bit patterns entering through its data bus port as "instructions", which will cause it to perform some simple task. The definition of the bit patterns, corresponding to instructions, depends entirely on the particular microprocessor chip in use and is referred to as the "instruction set" of the microprocessor. Instruction sets for microprocessors may vary significantly from one processor to another but the basic instructions are essentially similar. Let us examine a few such instructions from a hypothetical processor. We will assume that we have an "8-bit" microprocessor, which means that its basic unit for data and instructions consists of 8 bits (ie: 1 byte). Table 1.6 lists the bit patterns corresponding to instructions for the hypothetical system.

<i>Binary Machine Code</i>	<i>Instruction</i>	<i>Abbreviation (Mnemonic)</i>
0000 0001	Load internal microprocessor register "A" with the number represented by the next byte appearing on the data bus.	LDA
0000 0010	Load internal microprocessor register "B" with the number represented by the next byte appearing on the data bus	LDB
0000 0011	Add the contents of register "A" to the contents of register "B" and store the sum in register "A"	ADD
0000 0100	Take each bit in register "B" and "OR" it with the corresponding bit which comes from the next byte appearing on the data bus	ORB
0000 0101	If the contents of register "A" are not zero then jump to the program instruction in memory specified by the next two bytes appearing on the data bus	JNZA

Table 1.6 - Some Basic Instructions for a Hypothetical Processor

Note the word "MNEMONIC", used in regard to low level programming, is an abbreviated means by which we can remember what each instruction (bit pattern) does.

If we wanted our hypothetical microprocessor to add the numbers 3 and 7 together, then the following code might be used:

```
00000001    (LDA)
00000011    (3)
00000010    (LDB)
00000111    (7)
00000011    (ADD)
```

There are several questions that need to be examined at this point. Firstly, where does the program reside before it is executed by the microprocessor? Secondly, how does the microprocessor fetch these instructions? Thirdly, how can a microprocessor differentiate between the bit pattern corresponding to the ADD instruction and that corresponding to the number 3 (since they clearly have the same bit patterns)?

The third question is perhaps the easiest to answer. The microprocessor cannot differentiate between data and instructions by any means other than the order in which they arrive. Each instruction within the microprocessor's set, must be accompanied by an "argument" (qualifier) of a pre-defined length. For example, the LDA command must be qualified by an 8 bit number. The ADD command on the other hand, has a qualifier of 0 bits in length (ie: no qualifier). Therefore, the microprocessor will treat the second line of the above program as data (3) for the LDA instruction and NOT as the ADD command.

The first and second questions are somewhat interrelated. In order to understand the answer to these questions and thereby, the mechanism by which program execution in a microprocessor based system occurs, we need to understand the principles by which memory chips operate. We also need to examine the concept of "addressing", which will also arise in our subsequent discussions of external data communications.

As a first step towards understanding program execution and internal data transfer, we need to look at a more detailed picture of a microprocessor based system than the one that we started out with in Figure 1.1. Figure 1.6 shows the same basic elements as Figure 1.1, but this time we have added another cluster of 16 conductors, which we term the "address bus". As with the data bus, the number of lines in the address bus depends entirely on the architecture of the microprocessor system. Typical address bus sizes may be 16, 32 or 64 bits. We have also added two other blocks to the system, which are shown as "address decoders". We shall properly introduce these a little later.

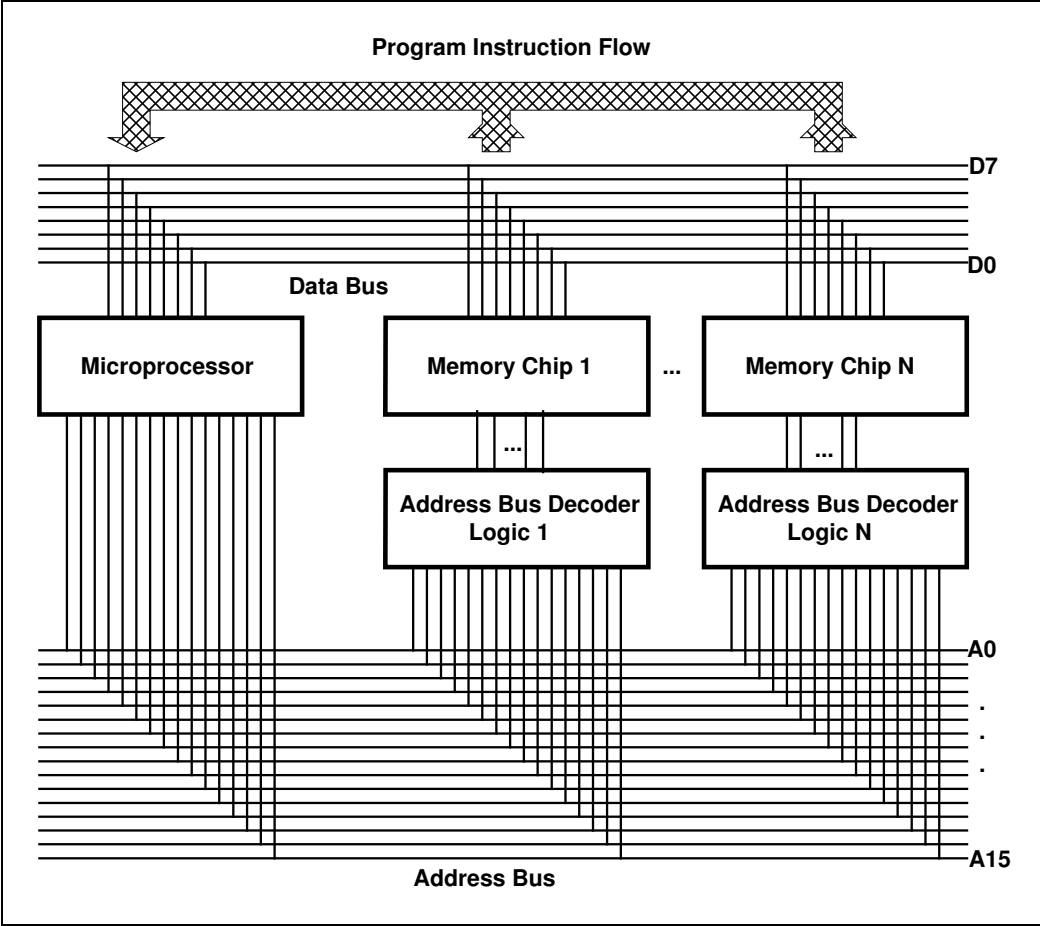


Figure 1.6 - Microprocessor System Data and Address Bus

To help us understand the role of the address bus, we look more closely at the memory chips themselves. Figure 1.7 schematically shows a hypothetical memory chip, which has storage for 16 bytes of information. Actual memory chips typically store many hundreds of kilobytes and a correspondingly larger number of memory access pins. You should observe that in our hypothetical chip of Figure 1.7, the bit patterns we have shown inside are the same as our addition program from earlier discussions.

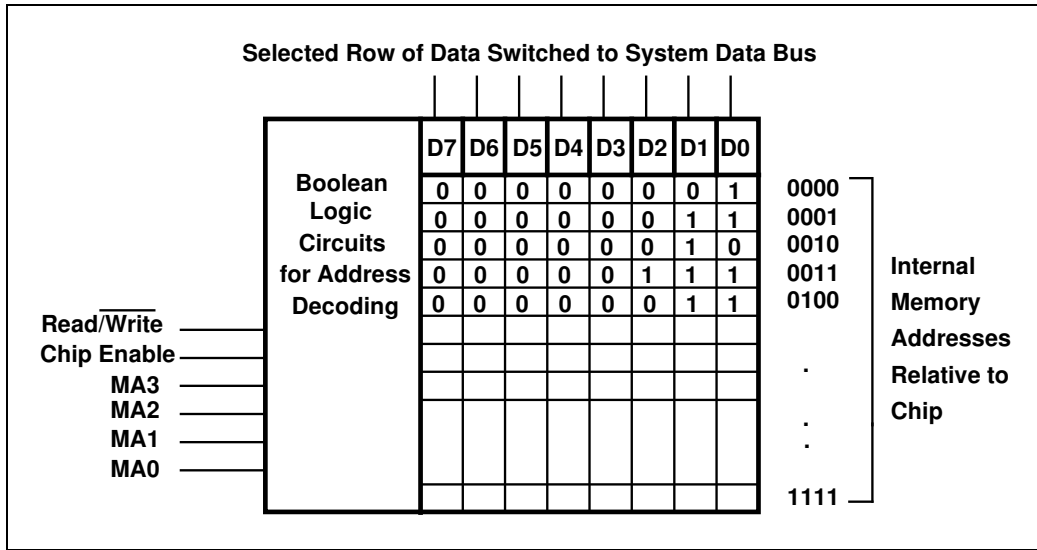


Figure 1.7 - Schematics of Memory Chip Operation

The memory chip is composed of two functional sections - the Boolean decoding logic section and the actual storage section.

The decoding section is responsible for controlling the transfer of data to/from the computer data bus from/to a storage address in the chip. This access control is based upon the status of the memory address pins (MA3..MA0), the **Read/Write** pin and the **Chip Enable** pin.

The **Chip Enable** pin can be thought of as a locking device for the chip. In order to use the chip, the **Chip Enable** pin must be set either high or low (depending on the manufacturer's design). The **Read/Write** pin is used on RAM chips to define whether data should flow into or out of the chip. Depending upon the high or low state of this pin, data will be written to or read from the appropriate address location in the chip. As an example, if the microprocessor in our hypothetical system is to force the memory chip to place its third row of storage onto the data bus, then it would have to set:

- MA3 = 0
- MA2 = 0
- MA1 = 1
- MA0 = 1
- Read/Write = 1
- Chip Enable = 1

thus accessing the third row in the memory chip.

The microprocessor controls the memory chip address pins and chip enable pin/s, by setting appropriate lines on the address bus high or low. In other words, a selection of address lines from the microprocessor is connected to pins on the memory chips. These lines are selectively set or reset by the microprocessor in order to make the memory chips respond in the desired manner. The **Read/Write** line of the memory chip is tied to a corresponding driver line on the microprocessor.

The dilemma that immediately arises, is that if there are many identical memory chips, connected to the address bus of a microprocessor system in an identical manner, then all the chips will respond simultaneously to each request from the microprocessor. This is clearly ridiculous, since it would imply that no matter how many memory chips we have, we would only effectively have the storage capacity of one chip. Since it would be equally ridiculous to have scores of specially designed memory chips, the problem is overcome through the use of "memory mapping" techniques.

Each memory chip in a microprocessor system must have a unique connection to the microprocessor address bus, otherwise a conflict occurs. This unique addressing is achieved through address bus decoding logic, as shown in Figure 1.6. This logic can be implemented through Boolean logic gates, as discussed earlier.

Let us now assume that two of the memory chips in the system shown in Figure 1.6 are like the hypothetical one shown in Figure 1.7. The following Boolean logic could be used to decode the address lines for chip 1:

$$\begin{aligned}MA0 &= A0 \\MA1 &= A1 \\MA2 &= A2 \\MA3 &= A3 \\Chip\ Enable &= X1\end{aligned}$$

where:

$$X1 = \overline{A4 + A5 + A6 + A7 + A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15}$$

Memory chip number 2 in the system could have the following Boolean logic gates for address bus decoding:

$$\begin{aligned}
 MA0 &= A0 \\
 MA1 &= A1 \\
 MA2 &= A2 \\
 MA3 &= A3 \\
 \text{Chip Enable} &= A4 \cdot X2
 \end{aligned}$$

where:

$$X2 = \overline{A5 + A6 + A7 + A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15}$$

This arrangement implies that memory chip 1 can only be accessed (enabled) when all microprocessor address lines A4 through to A15 are low. Otherwise chip 1 remains locked. Memory chip 2 can only be accessed when address line A4 is high and lines A5 to A15 are low. Otherwise chip 2 remains locked. Table 1.7 shows the effect that this has from the microprocessor's point of view. A number of such chips could be selectively "mapped" so that to the microprocessor, all address locations from 0000 0000 0000 0000 to 1111 1111 1111 1111 can be accessed, with no two chips responding to the same address.

<i>Address Bus Value</i>	<i>Memory Chip Accessed</i>	<i>Memory Chip Internal Address</i>
0000 0000 0000	1	0000
0000 0000 0001	1	0001
0000 0000 0010	1	0010
0000 0000 0011	1	0011
.	.	.
.	.	.
0000 0000 1111	1	1111
0000 0001 0000	2	0000
0000 0001 0001	2	0001
0000 0001 0010	2	0010
0000 0001 0011	2	0011
.	.	.
.	.	.
0000 0001 1111	2	1111

Table 1.7 - Memory Map for Example System of Figure 1.6

The address bus differs from the data bus in the sense that data flow is essentially uni-directional. Generally, the microprocessor is the master which sets or resets selective lines on the address bus, and the other connected devices respond accordingly as slaves. This is an important point. Also note that memory chips are not the only devices that are "mapped" in microprocessor based systems. A number of different chips and devices are controlled through the same mapping mechanism and appear, to the microprocessor, as though they were nothing more than memory locations.

In some microprocessor based systems, the same conductors are used for the address bus and the data bus. The microprocessor multiplexes (switches) the lines from one application to another. This saves considerable space on the system board. Most realistic systems also have quasi-intelligent chips known as "bus controllers" that are responsible for switching and resolving contentions on the bus system.

Now that we have seen how data is stored and transferred within a microprocessor based system, we can examine what happens when our addition program, from above, executes on such a system.

In order to run a program, a microprocessor must be primed with the starting address of the first instruction. Let us assume that our program is stored in chip number 2 of the hypothetical system, and that the first instruction is stored in the first memory slot within this chip. A register within the microprocessor is used to store the address of the current instruction to be executed. This is updated (incremented) as each instruction is executed. The first instruction, according to our memory map, will be at address 0000 0001 0000.

The microprocessor uses the address bus, and asserts the READ mode on memory chips to force them to place their contents on the data bus. Both instructions and data flow through the data bus into the microprocessor, where they are processed. This is all "time-governed" by the microprocessor system clock input. Figure 1.8 shows the voltage waveforms on the data bus (which enter into the microprocessor's data port) that make up our simple addition program.

The following is the logical sequence for the execution of the addition program cited as an earlier example:

- (i) Microprocessor sets address bus to 0000 0000 0001 0000 and asserts READ line
- (ii) Memory chip 2 sets data bus to 0000 0001(LDA)
- (iii) Microprocessor sets address bus to 0000 0000 0001 0001

- (iv) Memory chip 2 sets data bus to 0000 0011 (3)
- (v) Microprocessor sets address bus to 0000 0000 0001 0010
- (vi) Memory chip 2 sets data bus to 0000 0010(LDB)
- (vii) Microprocessor sets address bus to 0000 0000 0001 0011
- (viii) Memory chip 2 sets data bus to 0000 0111 (7)
- (ix) Microprocessor sets address bus to 0000 0000 0001 0100
- (x) Memory chip 2 sets data bus to 0000 0011(ADD)

The timing diagram of Figure 1.8 shows that information transfer along the data bus is essentially parallel, with all 8 bits issued and arriving simultaneously at their destination. Addressing information is also transferred in this parallel manner. The actual mechanism for data flow in realistic systems is complicated by the fact that the design needs to account for differences in chip speeds. This is generally achieved through the insertion of "idle" or "wait" states.

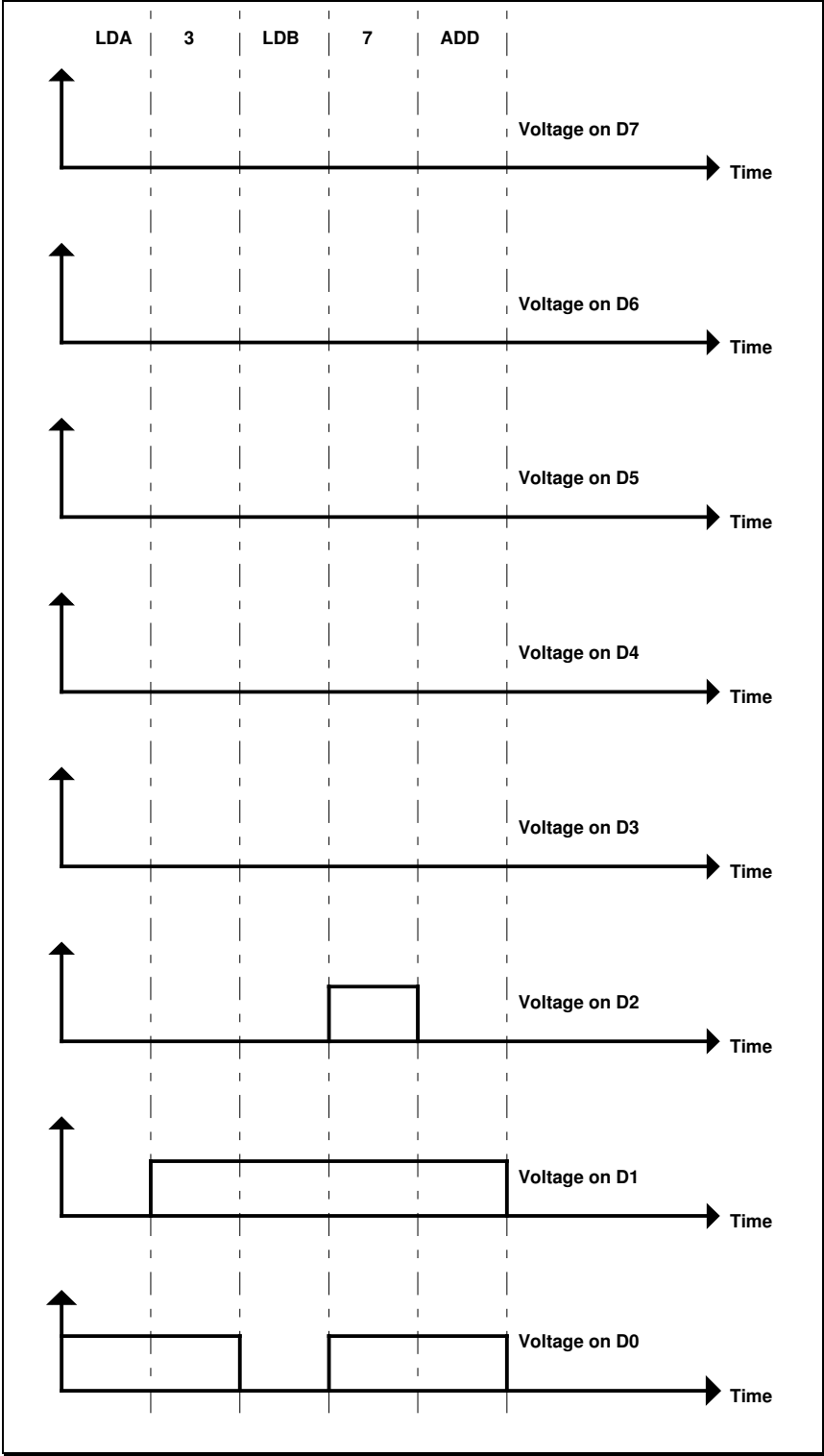


Figure 1.8 - Data Bus Voltages as Addition Program Executes

1.6 Master-Slave Relationships and Interrupts

We have now seen how data is both represented and transferred, in parallel, within microprocessor based systems, through the data bus. We have seen how the microprocessor exerts control over other chips within the system, by selectively enabling and disabling "Chip Enable" pins, through the system address bus. Up to this point, we have assumed that all devices connected to such a system are passive, and incapable of performing complex functions akin to those of the microprocessor. This is in fact not the case.

Many devices that are mapped into the microprocessor system are semi-intelligent in their own right. The chip devices that control the serial communications port are a good example. These are called Universal Asynchronous Receiver Transmitter (UART) chips and are used to transmit and receive data from equipment that is external to the microprocessor system. Such devices require a degree of autonomy in order to unload work from the microprocessor.

For example, the microprocessor may require data entering the UART from an external system. There may be long and varying time intervals between the arrival of data. If the microprocessor has to spend all its time interrogating (polling) such a device to determine whether data has arrived, then there is little scope for the processor to carry out normal program execution. This problem is overcome through the use of "interrupts" and "interrupt programming". This is shown schematically in Figure 1.9.

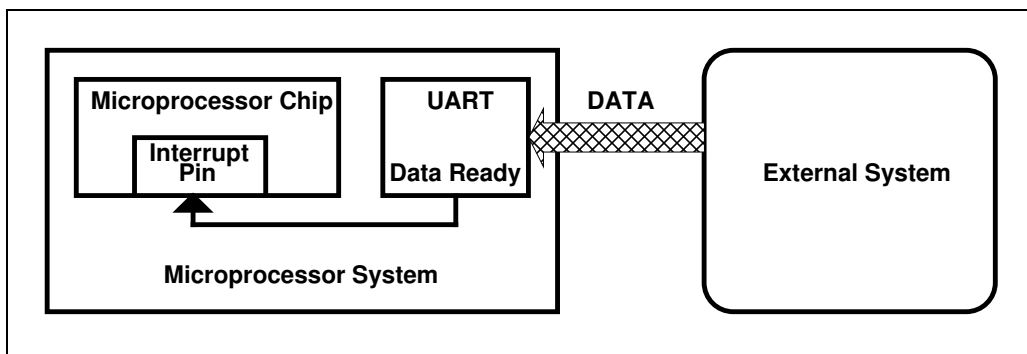


Figure 1.9 - Interrupt Methods for UART Control

Microprocessors generally have one or more interrupt pins that enable them to provide other devices within the system a degree of autonomy. The microprocessor can issue a task to another intelligent device and then continue with normal program execution. When the intelligent device has completed the set task, it sets the interrupt pin on the microprocessor either high or low (depending upon the processor).

When an interrupt occurs, the microprocessor can temporarily halt normal program execution and save all relevant parameters, such as the address of the current instruction, etc. The microprocessor then jumps to a new location in memory to execute a special program (routine) which has been written to handle the situation that has led to the interruption. Once the processor completes the execution of the interrupt routine, it restores the parameters from its original program and reverts to normal program execution as though nothing had happened.

This form of interrupt programming enables other intelligent devices, within the microprocessor based system, to perform tasks without the constant supervision of the processor.

It is important to note however that despite the additional degree of freedom granted to devices, through interrupt programming, the "closed" computer system is still very stable and free of conflicts. The microprocessor essentially co-ordinates, enables and disables all the other devices in the system through its master to slave relationship.

Under normal circumstances there is no possibility of a slave chip attempting to place data onto the data bus at the same time as the microprocessor. Similarly, conflicts on the address bus are not possible, since the microprocessor uses this as the controlling mechanism for the slave chips and not vice versa.

This well controlled, conflict free, master-slave situation only exists because individual systems are generally designed by a single, co-ordinating manufacturer. However as we shall see in later chapters, when dealing with more than one system and manufacturer, the problems of resolving conflicts during communication are greatly increased.

