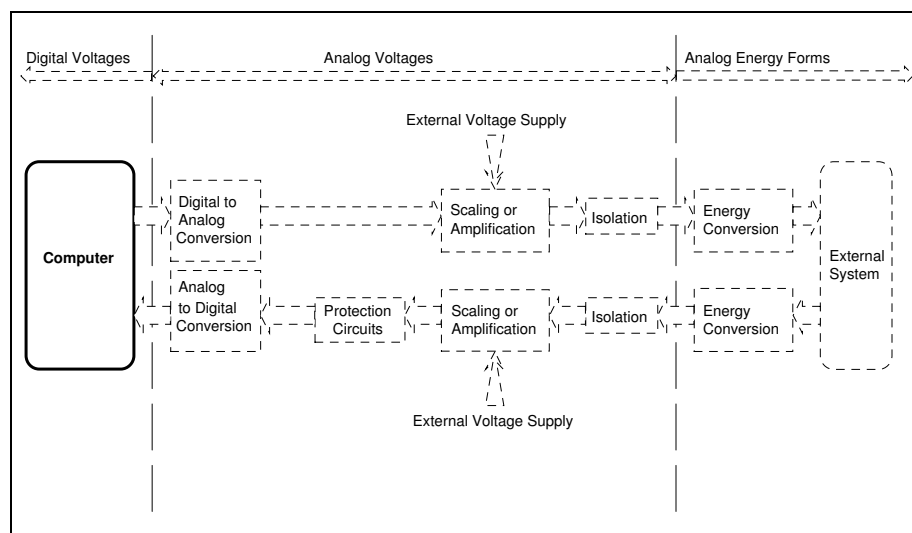

Chapter 4

Fundamentals of Digital Circuits

A Summary...

An introduction to digital circuits and systems and the way they are used to create logical and arithmetic circuits, storage devices (flip-flops and registers), counters and sequential logic. The binary number system and binary arithmetic. Boolean Algebra and digital logic circuit design. Boolean reduction techniques (Karnaugh Mapping and Boolean laws of tautology). Logic gates and types of hardware logic (TTL, MOS, CMOS, etc.).



4.1 A Building Block Approach to Computer Architecture

Many people who use and design systems based upon microprocessors never fully understand the architecture of such processors. There are many reasons for this. Some people have managed to live their professional lives without ever having learnt, whilst others have learnt but have failed to understand.

The difficulty in understanding the architecture of a computer system or microprocessor is that these devices are a combination of many component pieces of knowledge and technology. Many universities and text books explain, in great detail, how individual components work but fail to show how the pieces are brought together into a computer system or microprocessor. Some have endeavoured to explain the basic operation of computers and microprocessors by choosing a realistic example, based upon a particular processor. The problem with this approach is that there are so many side (technical) issues involved in the practical implementation of a computer system (for commercial purposes) that they tend to obscure the basic principles.

In this book, we will be taking a slightly different approach to computer architecture. Firstly, we will introduce, by analogy, the functionality of the microprocessor. We will then overview the basic building blocks that fit together to make up an operational microprocessor and computer system. We will then expand upon these blocks stage by stage until we can piece them together into something resembling a workable system. We will be looking at the design of a system by examining a hypothetical (generic) processor that exhibits the rudimentary traits of most modern processors. If you can come to terms with the basic aspects of the modern processor, then you should be able to approach any technical or commercial description of a particular device and understand where and why it differs from the basic form.

In order to begin our discussions on the architecture of modern digital computers, we examine a relatively basic "mechanical" device. We do so because our objective in this book is to establish, convincingly, that the most sophisticated microprocessors are still very much electronic machines and nothing more. Many people have great difficulty in developing hardware and software because they have come to believe, in a literal sense, the term "intelligent" that has been ascribed to microprocessor based devices. Ironically perhaps, those who use microprocessors as nothing more than machines often achieve far more with them than those who revere their ability. This shouldn't be altogether surprising because maximising the performance of a computer based system really depends upon understanding its intrinsic limitations.

Consider a mechanical control system such as the one found on an older style automatic washing machine. The arrangement first appeared in the 1950s and is shown in Figure 4.1. The system is composed of a dial, driven by a small motor. The dial steps from one position to another at uniform time intervals. At each position, the control system generates a number of output voltages that are used to drive relays and solenoids (which in turn activate pumps, motors, etc.). These output voltages, which we refer to as the "state" of the system controller, are dependent upon the inputs to the controller (from the user buttons, temperature sensors, etc.) and the current position of the dial. The speed at which the system moves from one state to the next is determined by the speed of the motor driving the dial. The motor driven dial can be described as a "state machine".

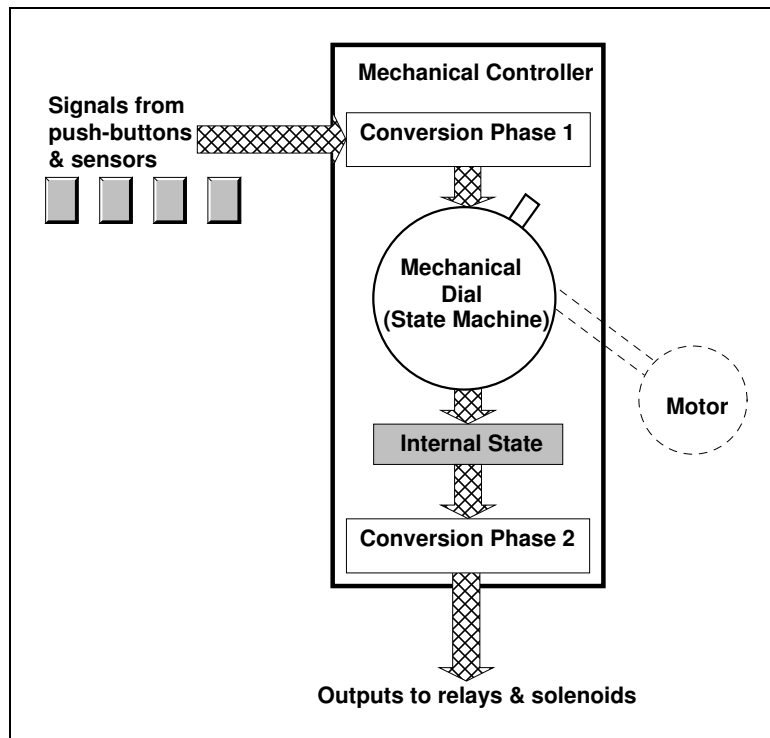


Figure 4.1 - Simple Mechanical Controller

The modern microprocessor is, in principle, little more than an electronic version of such a mechanical controller. The major difference of course is that the microprocessor can be fabricated onto a microscopic piece of semiconductor material and can move from one state to another in micro or nano seconds rather than seconds. In Figure 4.2, the microprocessor is shown in a completely analogous form to the mechanical controller of Figure 4.1.

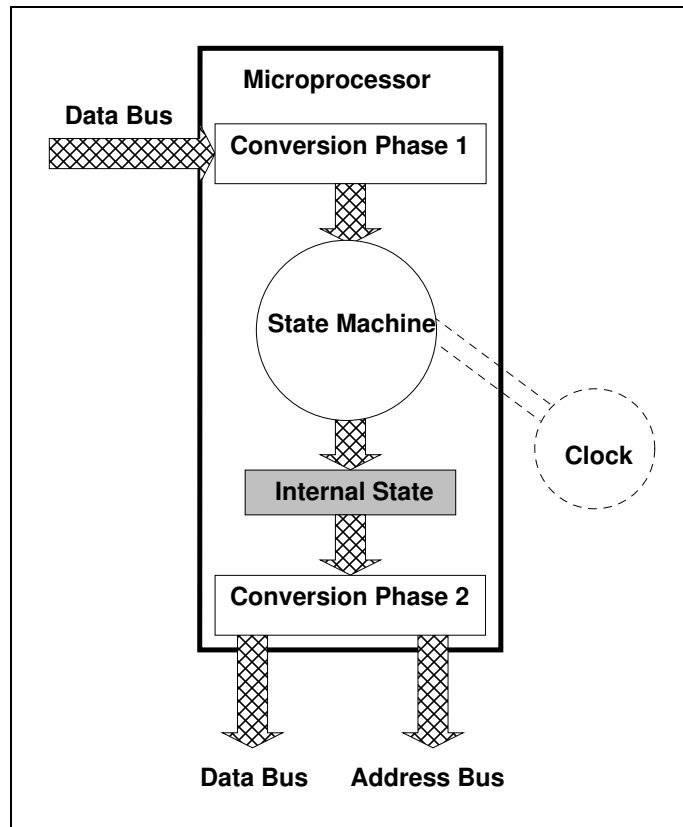


Figure 4.2 - Microprocessor Analogy to Mechanical Controller

In the case of the microprocessor, the heart of the system is an electronic "state machine", which moves the system from one state to another based upon a digital clock input. The faster the clock signal, the faster the machine moves from one state to another. The inputs to the microprocessor system come from the data bus and the outputs from the microprocessor can either be fed back onto the data bus or onto the address bus.

The internal states of the microprocessor are decoded by conversion circuits so that some useful work can be performed before an output is provided. For example, successive inputs from the data bus can be added together or subtracted from one another and the result sent to the output side of the processor (data bus or address bus).

Outsiders to the world of computing are often surprised to find that even the most powerful microprocessors are relatively primitive devices in the sense that they can do little more than add or subtract (most don't even multiply or divide), temporarily store and manipulate inputs and then feed them out again. However, this highlights the very "mechanical" nature of computers and the need for a high level of understanding before they can effectively be utilised.

A number of basic elements are common to all modern processors. These include the following:

- Reasoning circuits (referred to as combinational or Boolean logic)
- Storage circuits (referred to as registers)
- Mathematical circuits (referred to as numerical logic)
- Sequential circuits or "state machines"
- A cluster of conductors for input and output of voltages (called the data bus)
- A cluster of conductors for output of voltages (called the address bus).

The microprocessor, in isolation, does not perform any realistic computation. In order to operate effectively, it must be coupled to a number of other devices, including:

- A collection of registers for temporary storage of data (referred to as memory) that can provide input to the processor or receive output from the processor
- A bulk data storage facility controlled by another state-machine or processor (referred to as a disk-drive)
- A data entry pad for human users (referred to as a keyboard)
- An output device, driven by another state machine or processor, for interaction with human users (referred to as a graphics card).

The complete system is then referred to as a computer. However, the computer cannot perform any useful work unless the processor generates meaningful outputs (to the data and address bus structures). This, in turn, can only occur if the processor receives meaningful inputs (from the data bus). The meaningful inputs are entered by the human user and are stored on disk or in memory until they are fed through the processor. A collection of "meaningful" inputs is referred to as a program or as software. Since the microprocessor is only a machine, and all other devices in a computer system are of lesser or equal intellect, the only intelligence which can be ascribed to computer systems is via software. The computer system is shown schematically in Figure 4.3, with a number of its key elements.

Most of the elements within the computer system share a number of common attributes. Firstly, all the devices work with only two numbers (zero and one) that are represented by two voltages - typically in the order of zero volts and five volts, respectively. Secondly, all the devices use a kind of reasoning or, more appropriately, logic, which we refer to as "Boolean logic". This is named after the mathematician George Boole (1815-1864) and is implemented via a range of different circuits that we refer to as "gates". Boolean logic gates can not only be used to instil human reasoning, but they can also be used to generate numerical circuits that can perform simple arithmetic, such as addition and subtraction. Boolean logic gates are most commonly formed by fabricating a number of transistors into circuits within a piece of semiconductor material, and in this chapter, we shall examine a few of the different technologies that are in use.

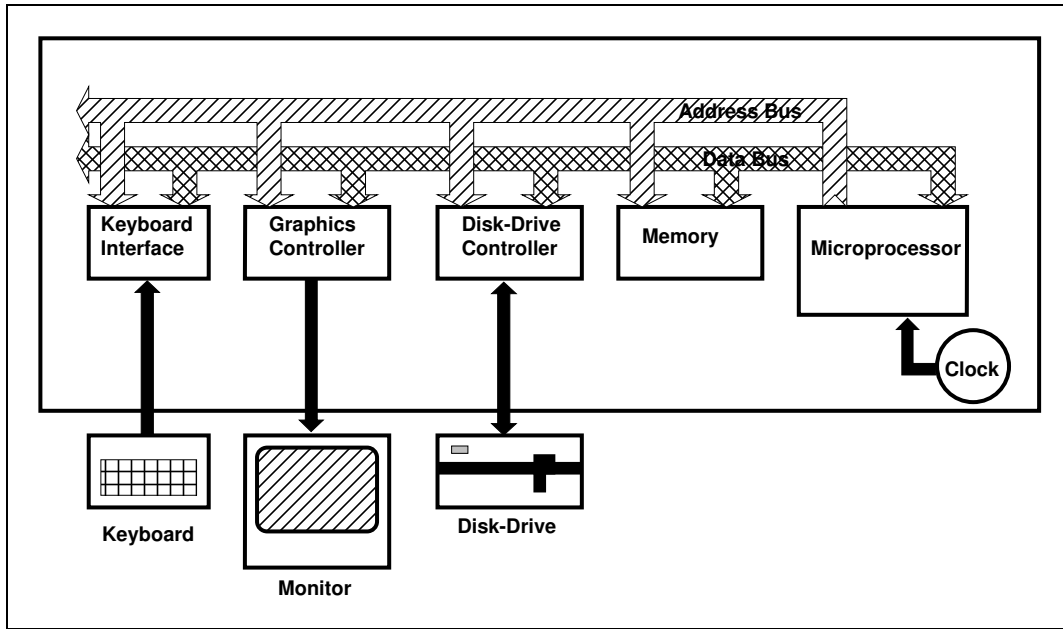


Figure 4.3 - Basic Computer System Elements

Boolean logic gates are fundamental to almost all areas of computing and so we need to find sensible ways of dealing with them. To this end, we shall examine a number of design and simplification techniques, such as the basic laws of Boolean Algebra (established by George Boole) and Karnaugh mapping.

Boolean logic gates can also be used to create memory storage elements that we refer to as "flip-flops", and a collection of flip-flops can be used to create a register. A number of registers fabricated onto a chip creates "memory". Moreover, if we take a collection of flip-flops and interconnect them with other Boolean logic gates, then we can create "counters" that move from one set of outputs (state) to another on each clock cycle. Counters are the most basic form of electronic "state machine" and hence the basis for the "heart" of modern processors.

So far, we have only discussed microprocessors and computers, and many would say that not all computers are based upon microprocessors. This is certainly true, and many larger computer systems do not contain a single-chip microprocessor. However, all modern computers contain some form of Central Processing Unit (CPU). Despite what many manufacturers would argue, there is little fundamental difference between the CPU of a large computer system, which is composed of many individual chips, and the CPU of a microcomputer system, which is composed of a single chip (the microprocessor). Most systems operate on the so-called "Von Neumann" architecture and the major difference between processors is in the way CPU functions are distributed - that is, on a single chip or over a number of discrete chips for improved performance.

The Von Neumann architecture is one in which program instructions and data are all stored in a common area of memory and transferred into and out of the CPU on a common data bus. Not all processors function on the Von Neumann architecture that we will discuss in some detail. A number of specialised processors, referred to as Digital Signal Processors or DSPs function on a slightly different architecture, referred to as the so-called "Harvard" architecture (notably, Von Neumann was a Princeton man!).

In the Harvard architecture, program instructions and data are stored and transferred separately from one another. This has advantages in a number of signal-processing areas because it enables sophisticated control calculations (such as Fast Fourier Transforms, etc.) to execute on a Harvard processor more rapidly. For this reason, devices such as DSPs are widely used where low-cost, high-performance processors are required. The Harvard architecture is however, less suitable for common applications than the Von Neumann architecture and is therefore not as widely used. In the final analysis, if one can come to terms with the Von Neumann architecture then one should be able to understand the Harvard architecture without difficulty.

The building blocks required to construct either the Von Neumann architecture or the Harvard architecture are the same. The difference is in the way the blocks are put together. The basic building blocks for both architectures are shown in Figure 4.4. It is these building blocks that we shall pursue for the remainder of this chapter and in Chapters 5 and 6, where we see how the basic elements fit together.

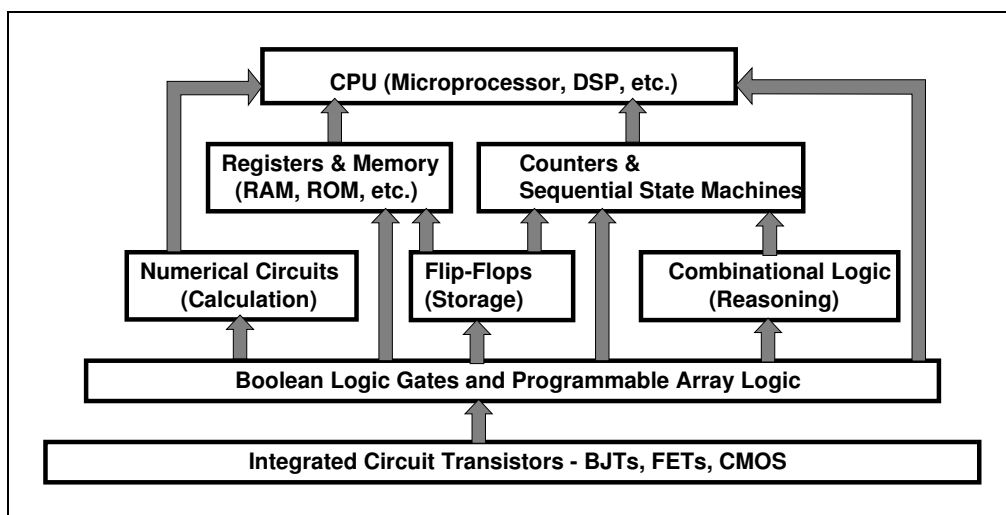


Figure 4.4 - Building Blocks in Computer Architecture

4.2 Number Systems, Conversion and Arithmetic

Digital computer systems have been designed in such a manner as to minimise the range of possible voltage levels that exist within. In fact, only two voltage ranges are normally permitted to represent data. One voltage range (normally somewhere near zero volts) represents the number zero and the other voltage range (normally around five volts) represents the number one.

The fact that we only have two possible voltage ranges means that we do not need to concern ourselves with circuit accuracy but rather with more important issues such as increasing speed and minimising size, cost and power dissipation. However, the restriction of having only two possible numbers (in other words, a binary number system) means that we need to be able to come to terms with a range of number systems other than the decimal one to which most humans are accustomed.

In Section 4.1 we looked at a simplified model of the microprocessor and computer system, as shown in Figures 4.2 and 4.3. Within that system we noted that the microprocessor essentially had one input, called the data bus, and two outputs, these being the data bus and the address bus. The term "bus" really refers to a number of conductors that are used to transfer energy (by current and voltage) or information via voltage levels, as in a computer system. The data bus is therefore not a single input/output line but rather a cluster of lines. A data bus can typically be composed of anywhere between 8 and 64 conductors, depending on the microprocessor system in question. The same applies to the address bus.

Each conductor on the address or data bus can, at any instant in time, be either "high" (around 5 volts) or "low" (around 0 volts) and in digital systems, we generally do not concern ourselves with the values in between. At any instant, therefore, a conductor can contain one *binary digit* of data. This is referred to by the abbreviated term "bit". Figure 4.5 shows the typical sort of voltage waveforms that could be travelling along an 8-bit data bus. At any instant in time, "T", the data lines contain 8-bits of binary data. In many instances the 8 bits are used to represent a number or a character. For example, at time "T", in Figure 4.5, the data bus could be representing the number 10111101 or a character corresponding to that binary number. There is no stage at which the microprocessor (or computer) ever sees numbers or characters in the form in which humans enter them. They are always represented by binary numbers.

The diagram of Figure 4.5 is actually only an approximation of what is actually occurring within a computer system. In fact, although we talk of digital circuits, most circuits only approximate digital behaviour. Figure 4.6 shows a time-scale enlarged version of a digital waveform which actually turns out to be analog in nature. However, for most design purposes the digital approximation is extremely good and it is only in relatively sophisticated "trouble-shooting" situations that we need to consider the analog behaviour of digital circuits.

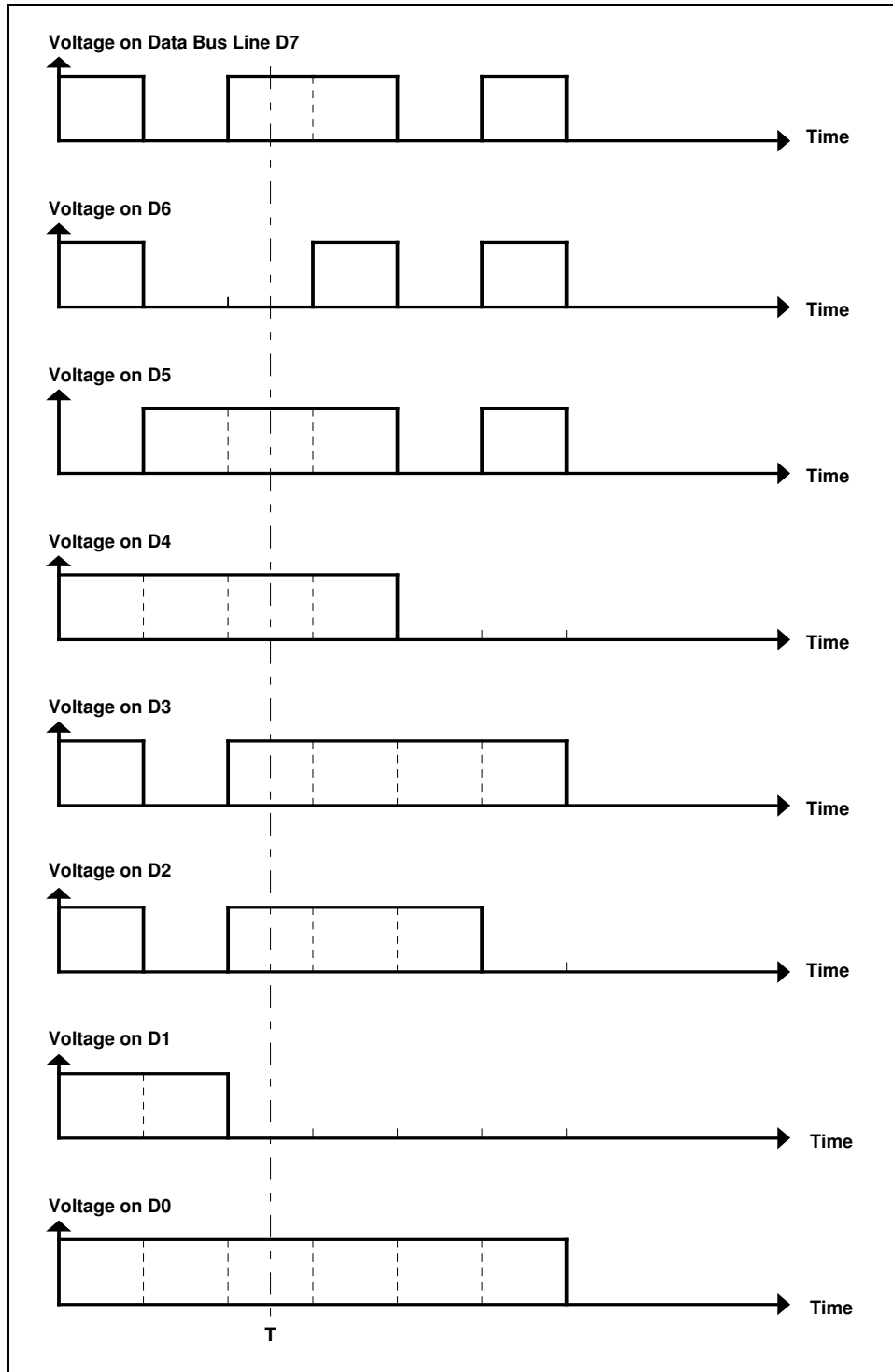


Figure 4.5 - Typical Digital Waveforms on an 8-bit Data Bus

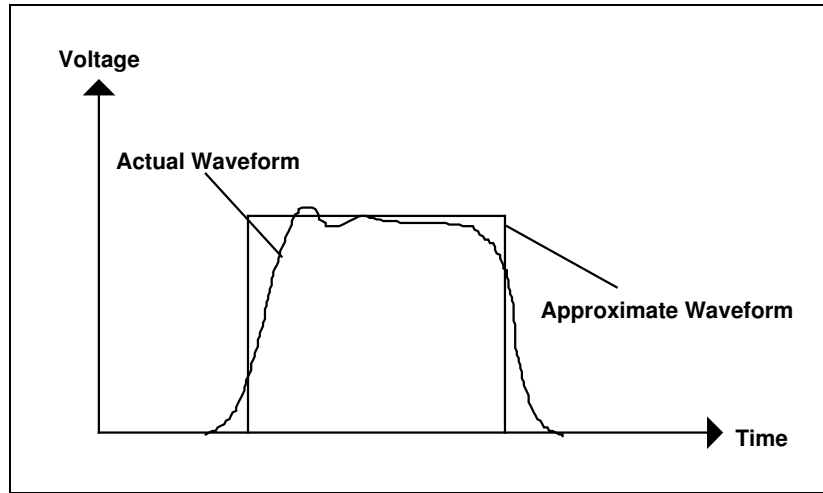


Figure 4.6 - The True Analog Nature of Digital Waveforms

All of the above discussions point to the fact that we need to understand how binary numbers can be used and manipulated and how we can convert from the binary number system to the decimal number system and so on. However, in order to understand how other number systems work, we must first ensure that we understand the decimal (or base 10) number system. To begin with, we note that the following is the natural, base-10, 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:

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

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

As an example, in order to convert the number 10111101_2 to decimal, we use the following procedure:

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

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:

$$\begin{array}{cccccc} 1 & 011 & 011 & 110 & 111 & \\ 1 & 3 & 3 & 6 & 7 & \end{array}$$

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:

$$\begin{array}{cccc} 1 & 0110 & 1111 & 0111 \\ 1 & 6 & F & 7 \end{array}$$

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 has also been used. 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 suitable 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. BCD is not used in sophisticated keyboards such as those found on most personal computers, workstations and main-frames. A more sophisticated representation is used for such keyboards and is discussed in Section 4.3.

To summarise the various number representations, most commonly associated with computers, Table 4.1 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 4.1 - Representation of Decimal Numbers from 0 to 20

Numbers from different bases can be dealt with arithmetically in exactly the same manner 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*

$$\begin{array}{r}
 11111 \text{ (Carry)} \\
 11101 \\
 + 1011 \\
 \hline
 101000 \\
 \hline
 \end{array}$$

(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*

$$\begin{array}{r}
 1011 \overline{) 11101} \\
 \underline{1011} \\
 1011000011
 \end{array}$$

4.3 Representation of Alpha-numerics

It should be clear from the discussions of 4.1 and 4.2 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 4.2, 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 4.2. 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
NULL	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 4.2 - ASCII and EBCDIC Character Representation

Table 4.2 may appear to be somewhat confusing on first glance and so a number of points need to be noted about its contents:

- (i) The first 32 characters in the ASCII character set are special characters that cannot be generated by pressing a single key on a keyboard. In the ASCII character set they are represented by acronyms (abbreviations), but it must be noted that typing the characters in the acronyms will not generate these special characters. A special key-stroke sequence is required to produce these characters. For example, on an IBM or compatible Personal Computer (PC), holding down the "Ctrl" key and then pressing "A" will generate the ASCII character with a value of 1.

The first 32 ASCII characters are sometimes referred to as non-printable characters. However, they do actually generate some symbols on particular computers - for example, on an IBM (or compatible) PC, these characters produce symbols such as ♣, ♦, ♥, ♠, etc. The main purpose of such characters is in data communications and they are also used for special instructions to printers. Table 4.3 lists the values of the first 32 non-printable characters for reference purposes, together with their commonly cited names and abbreviations.

- (ii) 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. Some, older PC software packages take advantage of the spare high-order bit to store additional character information such as bolding, underlining, etc.
- (iii) 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. This means that any numbers typed on a computer keyboard, that are intended to enter the microprocessor as numbers, need to be converted from their binary string (ASCII, EBCDIC, etc.) equivalent to their actual numeric value. For example, if we enter the characters "1" then "6" on the keyboard, then we generate the following ASCII string:

0011 0001 0011 0110

However, the binary number equivalent of 16 is actually 0001000 and so the microprocessor has to make the conversion from:

0011 0001 0011 0110 to 0001000.

This is generally carried out by the program executing on the microprocessor itself.

<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 4.3 - Special Functions of the first 32 ASCII Characters

4.4 Boolean Algebra

There are really only two major attributes that are instilled within the modern computer. One is the ability to "calculate" or to manipulate numbers. The other attribute is the ability to undertake some form of human reasoning. The term "computer" is actually defined as meaning "reckoning machine" and, in a sense, both calculation and reasoning are forms of reckoning.

We shall later observe that calculation and reasoning are somewhat interrelated phenomena in the computing domain because computers only ever calculate with the numbers "1" and "0" and only ever reason with the states "True" and "False". In order to understand how digital circuits are assembled in order to carry out both these functions, one must understand the fundamentals of Boolean Algebra, which is the mathematics of binary numbers and the reasoning (tautology) of systems with only "True" and "False" states.

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. The basic circuits used to provide Boolean logic within computer systems are referred to as "logic gates" and we shall examine these in a little more detail as we progress through this chapter.

Table 4.4 shows the logic gate symbols for the basic Boolean logic functions, together with their equivalent algebraic expressions. The actual standards for the logic symbols vary from country to country, but the ones adopted herein are in widespread use. The logic gate symbols don't necessarily have to describe electronic circuits - they can equally well be symbols for human reasoning or symbols representing mechanical circuits in hydraulic or pneumatic systems. Note that in Boolean Algebra, the symbol "+" signifies a logical "OR" (not addition) and the symbol "." means AND (not multiplication).

Table 4.4 is little more than a formal description of what many may feel to be perfectly obvious - simple reasoning elements. However, the objective of Boolean Algebra is to formalise a minute portion of the human reasoning process. The first step in doing so is to create a "truth table". This is shown beside each logic gate symbol in Table 4.4. A truth table is a listing of outputs corresponding to every possible input and input combination to a system. In a digital system there are only two possible values for every input - zero (False) or one (True). This clearly means that for a system with "n" inputs, there are 2^n possible input combinations or lines in the truth table. For example, the "inverter" gate of Table 4.4 has only one input and hence there are two lines in the truth table. The other gates each have two inputs and hence there are four lines in each of their truth tables.

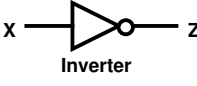
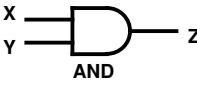
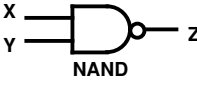
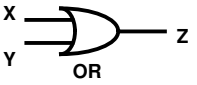
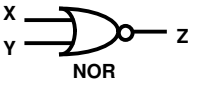
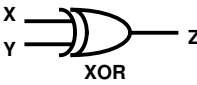
LOGIC GATE	BOOLEAN LOGIC	TRUTH TABLE			FUNCTION
		X	Y	Z	
 <p>Inverter</p>	Z is NOT X	0 1		1 0	$Z = \overline{X}$
 <p>AND</p>	Z is X AND Y	0 0 1 1	0 1 0 1	0 0 0 1	$Z = X.Y$
 <p>NAND</p>	Z is NOT (X AND Y)	0 0 1 1	0 1 0 1	1 1 1 0	$Z = \overline{X.Y}$
 <p>OR</p>	Z is X OR Y (Inclusive OR)	0 0 1 1	0 1 0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

Table 4.4 - Common Boolean Logic Functions and Representation

Consider how we can use Boolean logic to replicate human reasoning. For example, if we say that:

A = It is Hot
 B = It is Cloudy
 C = It is Humid
 D = It is Cold
 E = It is Wet

Then we can make Boolean statements such as the following:

It is cold and it is cloudy = $D \cdot B$ (read D AND B)

We can also instill our own reasoning into a systematic equation form. For example, we can say:

It is always humid when it is hot and cloudy and wet

This can be converted into:

$C = A \cdot B \cdot E$

It doesn't really matter whether our reasoning is valid or not. The issue is how to "automate" our reasoning. Following on from the previous example, we can make an electronic reasoning circuit using the gates shown in Table 4.4. The result is shown in Figure 4.7.

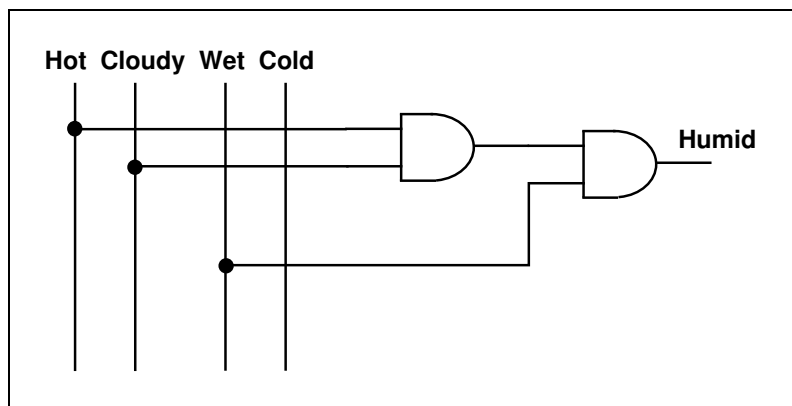


Figure 4.7 - Boolean Logic Circuit to Test for Humidity

Figure 4.7 shows how we can make a very trivial reckoning circuit to test for humidity by replicating our own reasoning with a circuit.

However, in order to understand the ramifications of building human logic into systems via Boolean circuits, we need to tackle a somewhat more substantial design exercise. Consider the following problem:

Design Problem 1:

An incubation chamber needs to be controlled by a simple digital controller. The complete system is shown in Figure 4.8. The chamber is equipped with a fan (F) and a heating element (H). The temperature of the system is fed back to a digital control system in a binary form. The temperature is represented by a 3-bit binary number, T2T1T0, which represents the incubation temperature on a scale from 000 to 111 (ie: 0_{10} to 7_{10}). If the temperature is less than 3, then the controller must switch the heating element on and the fan off. If the temperature is greater than 3 then the controller must switch the fan on and the heating element off. If the temperature is equal to 3 then the controller must switch both the fan and the heater element off. Design the control system using simple Boolean logic gates.

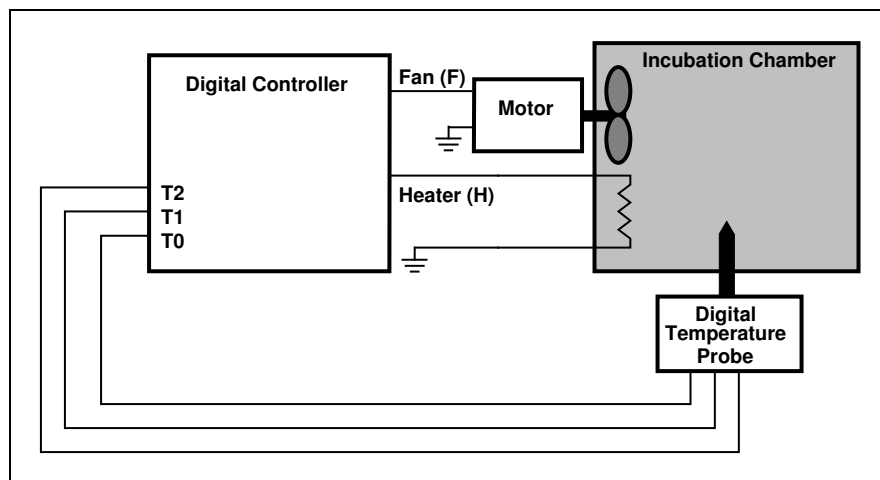


Figure 4.8 - Digital Control System for Incubator

Solution to Design Problem 1:

The first step in solving most digital design problems is to identify the inputs and outputs of the system. Sometimes this isn't as simple a task as it may seem. In this instance however, it is clear that the inputs are the three binary signals being fed back from the temperature probe (T2, T1 and T0). The next step in solving the problem is to construct a truth table for the problem. We will assume that the state "heater on" is represented by "H = 1", similarly, we assume that "fan on" is represented by "F = 1". The truth table is shown in Table 4.5.

System Inputs (2 ³ Combinations)			System Outputs	
T2	T1	T0	H	F
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Table 4.5 - Truth Table for Digital Incubator Controller

The next step in the design process is to determine the digital logic required to fulfil the logic in the truth table. The simplest technique is to use the so-called "sum-of-products" method. In order to do this, we go through each line of the truth table until we come across a line where the output variable is equal to one. We then write out the product of input variables that causes this to happen and then move down the truth table until we have written down a product for each time the output variable has a value of one. The products are then "ORed" together and the result is called the sum-of-products. In the case of the heater output "H" and the fan output "F", we have the following sum-of-products expressions:

$$H = \overline{T2} \cdot \overline{T1} \cdot \overline{T0} + \overline{T2} \cdot \overline{T1} \cdot T0 + \overline{T2} \cdot T1 \cdot \overline{T0}$$

$$F = T2 \cdot \overline{T1} \cdot \overline{T0} + T2 \cdot \overline{T1} \cdot T0 + T2 \cdot T1 \cdot \overline{T0} + T2 \cdot T1 \cdot T0$$

The sum-of-products expressions are really nothing more than common sense and define exactly the sort of logic that will fulfil the truth table. Looking at the sum-of-products expression for H, we can say that the heater is on whenever:

(T2 and T1 and T0 are all low) OR (T2 and T1 are low and T0 is high) OR (T2 is low and T1 is high and T0 is low)

Once we have a sum-of-products expression, we can convert that expression into logic gate symbols so that we have a Boolean logic circuit. This is shown in Figure 4.9 for the heating circuit (H).

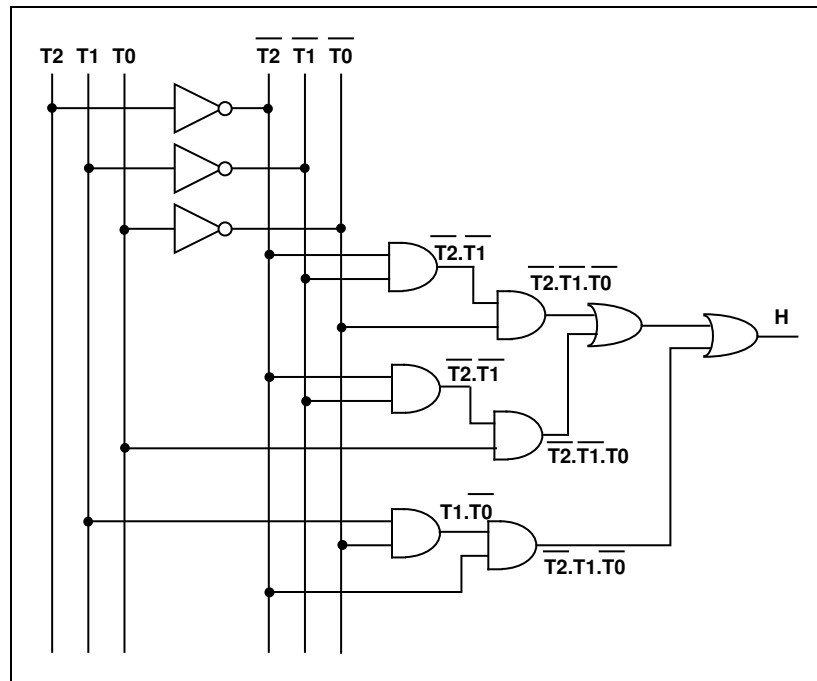


Figure 4.9 - Boolean Logic Circuit for Heater in Incubator Control

A similar circuit can be constructed for the incubator fan. Design this circuit, as an exercise, using the sum-of-products expression, defined above, for F . Combine the two circuits to show the complete control system.

Design problem 1 gives us a good insight into the way in which relatively simple digital controls can be constructed using simple logic building blocks. However, it also illustrates that a large number of components may be required for what is a relatively simple circuit. The sum-of-products technique is the most direct way of designing a Boolean logic circuit, however, in general it does not provide the simplest possible circuit to achieve a given objective. Many different logic circuit combinations may achieve exactly the same truth-table result as the one shown in Table 4.5 but some will use far fewer gates than others.

There are a number of laws and postulates in Boolean algebra that can be used to reduce an expression to its simplest form. These are listed in Table 4.6 and they are the basis of Boolean algebra. Using such laws to determine whether one Boolean algebraic expression is equivalent to another is referred to as "tautology". The final arbiter of tautology is the truth table. If two expressions are equivalent, then the truth table of the left hand side must be identical to the truth table of the right hand side for all possible variable combinations. The Boolean laws in Table 4.6 can all be verified by truth table.

<i>Postulates of Boolean Algebra</i>		
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:		
<i>OR</i>	<i>AND</i>	<i>NOT</i>
$1 + 1 = 1$	$1 \cdot 1 = 1$	$\bar{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\bar{0} = 1$
<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)	
$A \cdot A = A$		
$A + 1 = 1$		
$A \cdot 1 = A$		
$A + 0 = A$		
$A \cdot 0 = 0$		
$A + \bar{A} = 1$		
$A \cdot \bar{A} = 0$		
$A + \bar{A} \cdot B = A + B$		
$\overline{\overline{A}} = A$	(Laws of Double Complementation)	
$\overline{\overline{A + B}} = A + B$		
$\overline{\overline{A \cdot B}} = A \cdot B$		
$\overline{A \cdot B} = \bar{A} + \bar{B}$	(DeMorgan's Theorem)	
$\overline{A + B} = \bar{A} \cdot \bar{B}$	(DeMorgan's Theorem)	

Table 4.6 - Fundamental Principles of Boolean Algebra

The laws of Boolean algebra, as defined in Table 4.6, are normally used in order to make complex expressions simpler. Boolean logic is used for a number of functions including:

- Design of digital circuits from logic gates
- Design of logic circuits for hydraulics, pneumatics, etc.
- Writing conditional expressions in computer programs.

For these applications we always need to establish the simplest form of a Boolean expression before committing ourselves to an implementation phase. This reduces the complexity and cost of circuits and the running time of software.

Design Problem 2:

Using the laws and postulates of Boolean Algebra, simplify the following expression:

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

Solution to Design Problem 2:

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

Design Problem 2 highlights the difficulty in applying the laws of Boolean algebra to simplify a circuit. The main problem is that there is no predefined way of beginning the simplification process - the first step is arbitrary. Secondly, there are multiple paths that can be taken to achieve the same objective and the approach shown in Design Problem 2 is not systematic. Finally, for complicated expressions, we never really know when we have reached the simplest expression.

There are a number of techniques that can systematically simplify a Boolean expression. The most common is called "Karnaugh Mapping" and it is this technique which we shall explore herein.

A Karnaugh Map is really nothing more than a strategically drawn truth-table that plots the output variable in terms of the input variables. Table 4.7 is the conventional truth table for the original expression in Design Problem 2.

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>Z</i>
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Table 4.7 - Truth Table for Original Expression in Design Problem 2

Figure 4.10 shows the Karnaugh Map, equivalent to Table 4.7, for the output variable (*Z*) in the expression. The Karnaugh map is just a truth table plotted in a different way. However, there are two points to note about the Karnaugh Map:

- The count sequence on the map does not follow a normal binary count pattern. The reason for this is to ensure that only one variable changes at a time and is referred to as a "Gray Code" count sequence
- The map needs to be considered as a sheet of paper which can be folded around on itself. In other words, there really aren't any ends on the map. It can be rolled either vertically or horizontally.

		AB			
		00	01	11	10
CD	00	1	1	1	1
	01	1	1	0	1
	11	1	1	1	1
	10	1	1	1	1

Figure 4.10 - Karnaugh Map for Original Expression for "Z" in Design Problem 2

Once we have constructed a Karnaugh Map for an expression, the objective is to look for regions where the output variable is independent of the input variables. How do we do this? Firstly, by circling regions where the output has a value of 1 - however, we can only do this in a certain way:

- (i) In a 4 x 4 map as shown in Figure 4.10, we begin by looking for a region in which there are 16 "ones". If we find such a region then it means that the output is equal to one regardless of the inputs and hence is independent of the inputs. If this is the case then the process is concluded, otherwise we move on to (ii)
- (ii) We then move on to regions where there are 8 "ones" and circle all those. When there are no regions of 8 "ones" we look for regions of 4 "ones" then 2 "ones" and then 1 "one". It doesn't matter if some of the circled regions overlap
- (iii) Ultimately, all "ones" on the map have to be circled
- (iv) If a map has only regions of 1 "one", then there is no possibility of simplifying the expression
- (v) When all the regions have been circled, we look for regions of independence. In other words, we ask ourselves for each circled region, "do any of the inputs change in the circled region where the output remains constant?" If the answer is yes, then the output is independent of these inputs. If inputs do not change within a circled region, then the output is dependent upon those inputs

- (vi) The expression for the output variable is the sum of expressions for all the circled regions.

The Karnaugh Mapping technique can only be fully understood after some practice, so let us begin by simplifying the expression in Design Problem 2. As a first step, we identify regions as shown in Figure 4.11. Note how we only put boundaries around regions in either horizontal or vertical directions.

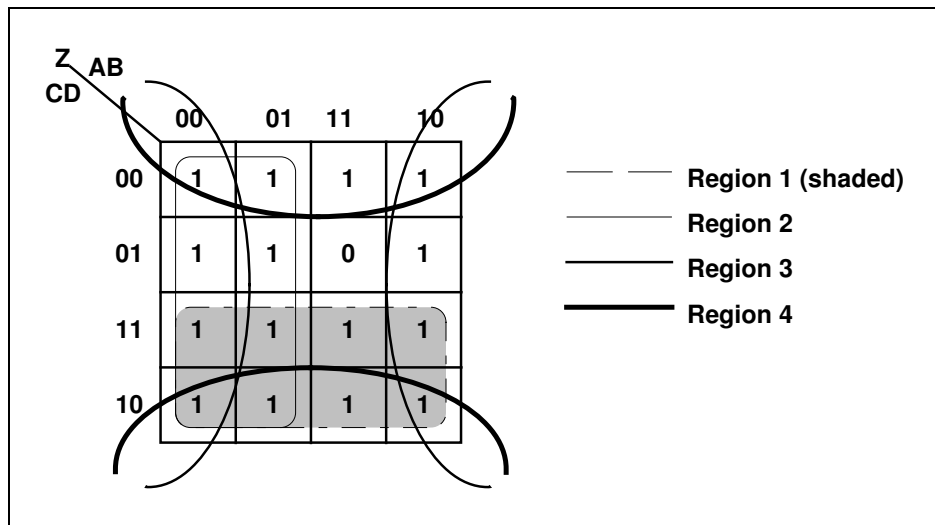


Figure 4.11 - Karnaugh Map for "Z" in Design Problem 2 with Regions Circled

The simplest Boolean expression can be determined by creating the largest possible regions (16, 8, 4, 2 and 1 consecutively) and then deducing how the output is affected in those regions. Let us begin with region 1 in Figure 4.11 (the shaded region). Note that the value of "Z" in this region is always one, despite the fact that input variables A, B and D change within the region. This means that for this region, the value of Z only depends upon C being equal to one. Therefore the simplest expression for Z is:

$$Z = C + ? + ? + ?$$

The second term in the expression for Z can be obtained from region 2 in Figure 4.11. In this region, the value of Z remains one, despite the fact that B, C and D inputs vary between one and zero. Hence in this region, Z is only dependent upon A being equal to zero. Our simplest expression for Z now becomes:

$$Z = C + \bar{A} + ? + ?$$

The third term in the expression for Z can be obtained from region 3 in Figure 4.11, which is the "outside" region (remember that the Karnaugh Map can be rolled around so that the ends meet in both vertical and horizontal directions). In this region, A , C and D all vary between zero and one and only the variable B remains constant with a value of zero. Therefore, Z depends upon B having a value of zero. Our simplest expression for Z now becomes:

$$Z = C + \bar{A} + \bar{B} + ?$$

The final term in our expression for Z is obtained from region 4 in Figure 4.11. In this region, A , B and C all vary between one and zero and have no affect upon Z . However, D must remain constant with a value of zero. Hence, our simplest expression for Z is:

$$Z = C + \bar{A} + \bar{B} + \bar{D}$$

as determined earlier by the unsystematic process of applying algebraic laws. Provided that we always select the largest possible groups, we will always arrive at the simplest possible expressions.

Karnaugh Mapping is always difficult to come to terms with at first, so following are a number of simple design problems to assist your understanding.

Design Problem 3:

Using the Karnaugh Maps shown in Figure 4.12, determine the simplest expressions for Z in each case.

<div style="text-align: left; margin-bottom: 5px;"> Z AB CD </div> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 5px;"></th> <th style="padding: 2px 5px;">00</th> <th style="padding: 2px 5px;">01</th> <th style="padding: 2px 5px;">11</th> <th style="padding: 2px 5px;">10</th> </tr> </thead> <tbody> <tr> <th style="padding: 2px 5px;">00</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <th style="padding: 2px 5px;">01</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> </tr> <tr> <th style="padding: 2px 5px;">11</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> </tr> <tr> <th style="padding: 2px 5px;">10</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> </tbody> </table> <p style="text-align: right; margin-top: 5px;">(i)</p>		00	01	11	10	00	1	0	0	1	01	0	0	0	0	11	0	1	0	0	10	1	0	0	1	<div style="text-align: left; margin-bottom: 5px;"> Z AB CD </div> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 5px;"></th> <th style="padding: 2px 5px;">00</th> <th style="padding: 2px 5px;">01</th> <th style="padding: 2px 5px;">11</th> <th style="padding: 2px 5px;">10</th> </tr> </thead> <tbody> <tr> <th style="padding: 2px 5px;">00</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <th style="padding: 2px 5px;">01</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> </tr> <tr> <th style="padding: 2px 5px;">11</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> </tr> <tr> <th style="padding: 2px 5px;">10</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> </tr> </tbody> </table> <p style="text-align: right; margin-top: 5px;">(ii)</p>		00	01	11	10	00	1	1	1	1	01	0	1	0	0	11	0	1	0	0	10	0	1	1	0
	00	01	11	10																																															
00	1	0	0	1																																															
01	0	0	0	0																																															
11	0	1	0	0																																															
10	1	0	0	1																																															
	00	01	11	10																																															
00	1	1	1	1																																															
01	0	1	0	0																																															
11	0	1	0	0																																															
10	0	1	1	0																																															
<div style="text-align: left; margin-bottom: 5px;"> Z AB C </div> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 5px;"></th> <th style="padding: 2px 5px;">00</th> <th style="padding: 2px 5px;">01</th> <th style="padding: 2px 5px;">11</th> <th style="padding: 2px 5px;">10</th> </tr> </thead> <tbody> <tr> <th style="padding: 2px 5px;">0</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <th style="padding: 2px 5px;">1</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> </tbody> </table> <p style="text-align: right; margin-top: 5px;">(iii)</p>		00	01	11	10	0	1	0	0	1	1	1	1	0	1	<div style="text-align: left; margin-bottom: 5px;"> Z AB C </div> <table border="1" style="margin-left: 20px; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 5px;"></th> <th style="padding: 2px 5px;">00</th> <th style="padding: 2px 5px;">01</th> <th style="padding: 2px 5px;">11</th> <th style="padding: 2px 5px;">10</th> </tr> </thead> <tbody> <tr> <th style="padding: 2px 5px;">0</th> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> <tr> <th style="padding: 2px 5px;">1</th> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> </tr> </tbody> </table> <p style="text-align: right; margin-top: 5px;">(iv)</p>		00	01	11	10	0	0	1	0	1	1	1	0	1	0																				
	00	01	11	10																																															
0	1	0	0	1																																															
1	1	1	0	1																																															
	00	01	11	10																																															
0	0	1	0	1																																															
1	1	0	1	0																																															

Figure 4.12 - Sample Karnaugh Map Problems

Solution to Design Problem 3:

The largest possible regions (containing binary ones) for each of the Karnaugh map are identified and marked as in Figure 4.13. Regions can be octets, quads, pairs or singles.

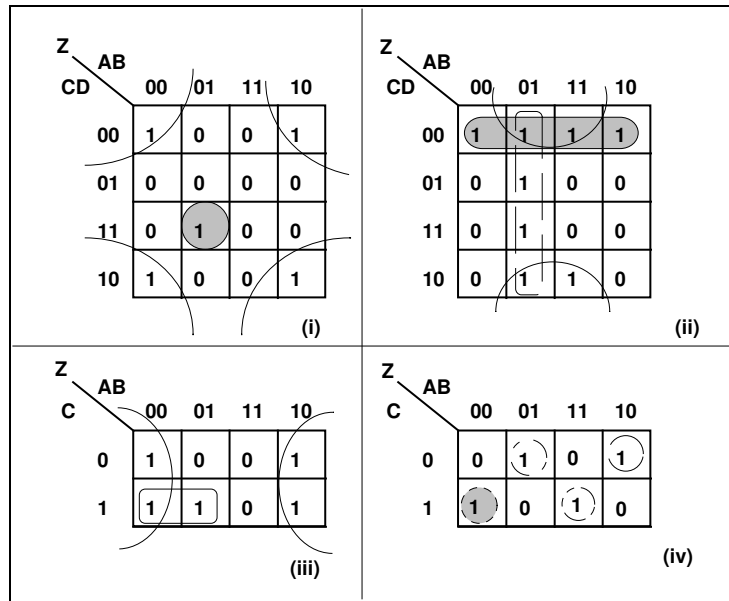


Figure 4.13 - Karnaugh Maps Marked out to Maximise Regions of "Ones"

We begin by considering Karnaugh Map (i) in Figure 4.13. Note how we have been able to group the four corners together into a quad group. This is because we can "roll" the edges of a Karnaugh map so that they join one another in either the vertical or horizontal directions. Rolling is permissible provided that we adhere to the basic rule that no more than one variable can change from any one position in the map to any other position. This binary number sequence, in which only one bit changes at a time, is referred to as "Gray code". From Figure 4.13 (i), we find that there are only two regions to be considered. The simplest expression for Z in (i) is:

$$Z = \bar{B} \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot D$$

In Figure 4.13 (ii), we have three "quad" regions. Two of the quads are in a line, but one arises from rolling the horizontal edges of the map together. The simplest expression for Z in (ii) is:

$$Z = \bar{C} \cdot \bar{D} + \bar{A} \cdot B + B \cdot \bar{D}$$

The Karnaugh Map of Figure 4.13 (iii) has only three variables and hence eight possible combinations. Karnaugh Maps can also be constructed for expressions with only two input variables - these are 2 x 2 Maps. In Figure 4.13 (iii), the largest possible region is a quad, obtained by rolling the vertical (left and right) edges of the Map. Note also that we have overlapping regions in the Map. If we had not joined the lone "one" with another one to form a pair, then the expression we derived would not be in its simplest form. The simplest expression for Z in (iii) is:

$$Z = \bar{B} + C \cdot \bar{A}$$

Figure 4.13 (iv) is an unusual Map and has deliberately been included because it is one instance where the Karnaugh Mapping technique doesn't provide the simplest Boolean expression at first sight. From the Map of Figure 4.13 (iv) it is evident that no octets, quads or pairs can be formed and that:

$$Z = \bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C$$

However, it turns out that this expression, is identical to

$$Z = A \oplus B \oplus C$$

which is a much simpler expression, formed from "Exclusive-OR" operators. In order to find Exclusive-OR operators in a Karnaugh Map one needs to look for specific patterns such as the one in Figure 4.13 (iv). As an exercise, construct truth tables and Karnaugh Maps for 2 input and 4 input Exclusive-OR systems and note the patterns that arise.

The examples in Design Problem 3 really don't highlight the radical simplification that can occur in expressions as a result of Karnaugh Maps. In order to observe this phenomenon, we really need an example that highlights "before" and "after" cases.

Design Problem 4:

Using the Karnaugh Mapping technique, redesign the incubation controller developed in Design Problem 1.

Solution to Design Problem 4:

The truth table derived for the incubator design problem was shown in Table 4.5. The Karnaugh Maps for the heater, "H", and the cooling fan, "F", are derived from that truth table and are shown in Figures 4.14 (i) and (ii) respectively.

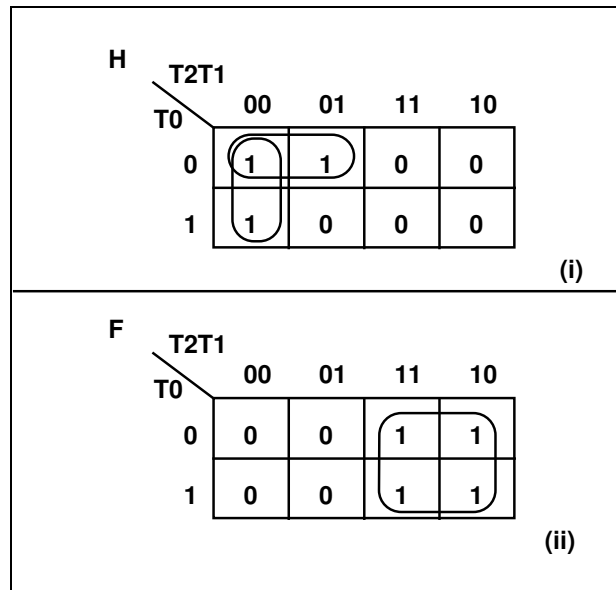


Figure 4.14 - Karnaugh Maps for Incubator Control
 (i) Heater; (ii) Fan

From these Karnaugh Maps, we can determine new expressions for H and F as follows:

$$\begin{aligned}
 H &= \overline{T2} \cdot \overline{T1} + \overline{T2} \cdot \overline{T0} \\
 &= \overline{T2} \cdot (\overline{T1} + \overline{T0})
 \end{aligned}$$

$$F = T2$$

These are clearly much simpler than the previously derived expressions and lead to the new (simplified) controller circuit of Figure 4.15, which performs precisely the same function as the original, but with far fewer components.

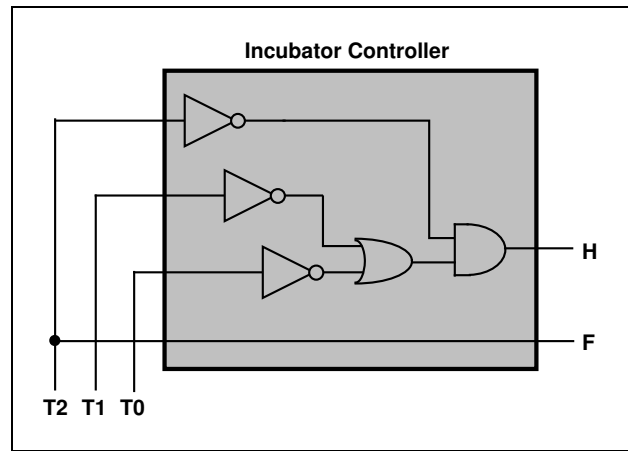


Figure 4.15 - Simplified Incubator Control System

4.5 Digital Logic Circuits

4.5.1 Introduction

In section 4.4 we examined a range of different digital logic circuits that could be used to exert some form of human reasoning (control) over a system. In that section, we dealt only with the symbol associated with each digital logic gate and assumed that the actual device could be fabricated from some form of electronic circuit. We now need to gain some understanding of how digital logic devices are actually fabricated so that we can understand their applications and limitations.

As a starting point, it should be noted that all of the digital gates described in 4.4 are available in an integrated circuit (IC) fabricated within a semiconductor chip. Normally, digital logic gates are implemented in a low-density semiconductor fabrication referred to as SSI, which is an acronym for Small-Scale-Integration. Even with SSI, one chip generally contains more than one logic gate. For example, Figure 4.16 schematically shows the contents of a "7400 quad 2-Input NAND gate" device.

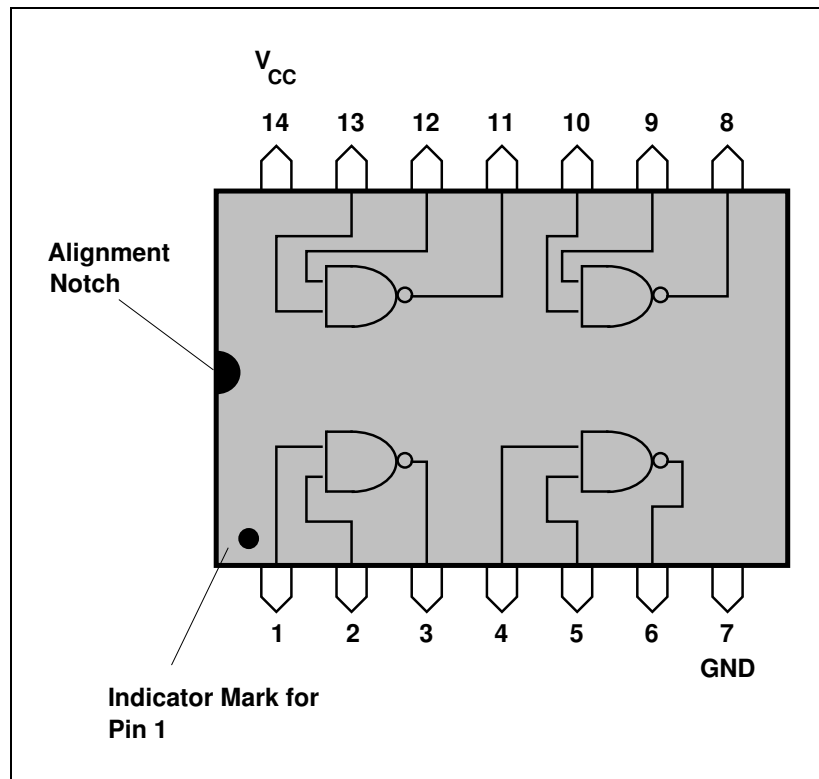


Figure 4.16 - 7400 Quad 2-Input NAND-Gate Dual-In-Line Package Chip

A number of features need to be noted about commercially available chips such as the one shown in Figure 4.16:

- (i) The semiconductor material upon which the digital circuits are fabricated, is less than a few square millimetres in area. The user generally doesn't see the semiconductor material on low cost devices such as the one in Figure 4.16.
- (ii) The semiconductor is encased in plastic or ceramic material that is referred to as the "package". This provides a practical casing that simplifies manual handling of the device and allows a larger area for external connections to pins on the outside of the package.
- (iii) Extremely fine wires connect various points in the semiconductor to the physical conducting pins protruding from the package.
- (iv) The number of pins and their functionality is referred to as the "pin-out" of the package.
- (v) The pin numbers are generally not marked onto the package of the chip. Most packages therefore have identifying features (notches, circular indentations, etc.) that enable users to determine the pin numbering and orientation of the device.
- (vi) The functionality of each pin in a particular package can only be determined by reference to a data sheet from the manufacturer and should never be "guessed" by looking at patterns for common chips
- (vii) Each chip has two power supply pins (normally referred to as V_{CC} and GND). Unless a power supply is connected to these pins then the chip will not generate the required digital logic.

From the above points it is evident that the package is generally much larger than the semiconductor material itself. In the 1960s, when this technology originated, few would have imagined the complexity of the circuits developed today and the major objective was to make packages to which users could easily connect other devices by manual techniques. However, one of the modern problems of developing complex circuits using packages such as the one shown in Figure 4.16 is that the size of the circuit largely represents packaging and not functional devices.

In automated production environments, manual handling of devices can be eliminated and the bulk of the packaging removed. This makes circuits far more compact. The most common automated technique for using integrated circuits (ICs) without packaging is called "surface-mount" technology.

A surface-mount machine is similar to a 3-axis (XYZ) CNC machine but its purpose is to pick and place components onto circuit boards positioned in the machine bed. The specially designed ICs are normally purchased in bulk in a "bubble-pack" roll. The surface mount machine removes each IC from its bubble (by suction) and places it onto the circuit board. A light adhesive holds the IC in place temporarily and the entire board is then heated in an oven in order to create conducting joints at appropriate locations between the board and the IC. A range of ICs and other components (capacitors, resistors, etc.) are available in bubble-pack rolls for integration onto surface mount boards and the technology has been in widespread use since the early 1980s. It is one of the most efficient techniques for mass production of both analog and digital circuits.

Most small-scale designers and prototype builders will have little use for surface-mount technology, since the machinery involved is quite substantial, even when purchased in a manual "pick and place" form. The two simplest approaches for building circuit boards with digital circuits involve:

- Hot soldering (the traditional method of connecting electrical and electronic circuit elements) onto printed circuit boards
- Wire-Wrapping or cold soldering (a technique which involves wrapping wires around the pins of various IC sockets into which are plugged the ICs themselves).

Neither of these approaches are suitable for prototyping because it is difficult to "undo" mistakes or rectify design faults. A common short-term approach is to build digital circuits on prototyping boards that are specially designed so that wires, ICs, resistors, etc. can all be inserted into spring-loaded, conducting tracks so that temporary connections can be made for test purposes. These boards are sometimes referred to as "bread boards".

As long as one operates digital circuits at moderate speeds, connects ICs together with short lengths of wire and doesn't load the output of one device with too many inputs from other devices, most physical circuits will function precisely as predicted in theory. However, the problems that arise from breaking such rules can only be understood when one understands the circuits used to fabricate digital circuit chips.

4.5.2 Transistor to Transistor Logic (TTL)

Transistor to Transistor Logic was one of the earliest and most widespread forms of digital logic and is still prominent today because most modern digital circuits still comply with the voltage and current standards established for that logic. The actual circuit for a TTL inverter gate is shown in Figure 4.17.

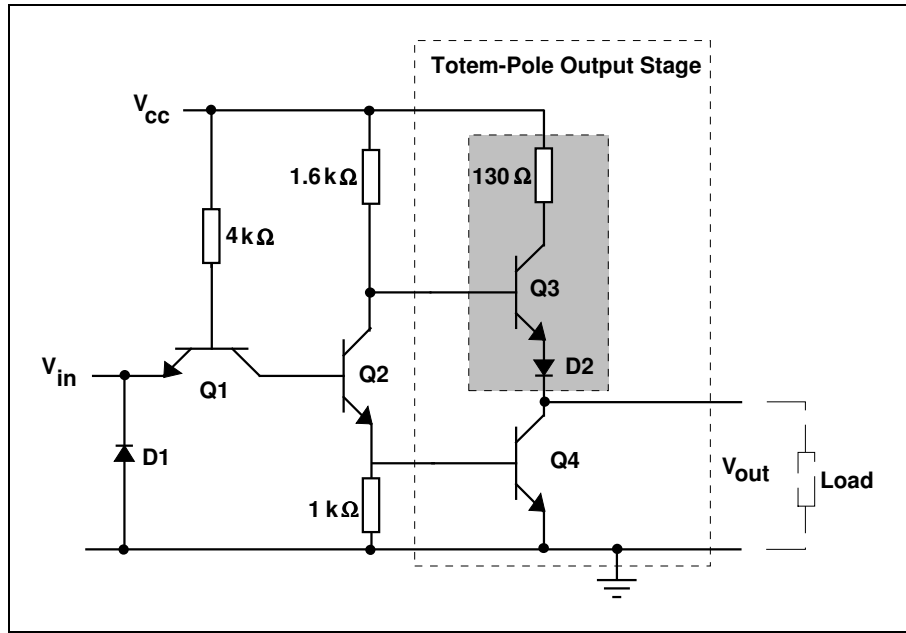


Figure 4.17 - TTL Inverter Gate

The actual operation of this circuit was discussed briefly in Chapter 3 and the input/output voltage levels associated with TTL type circuits (originally shown in Figure 1.2 (b)) are reproduced in Figure 4.18.

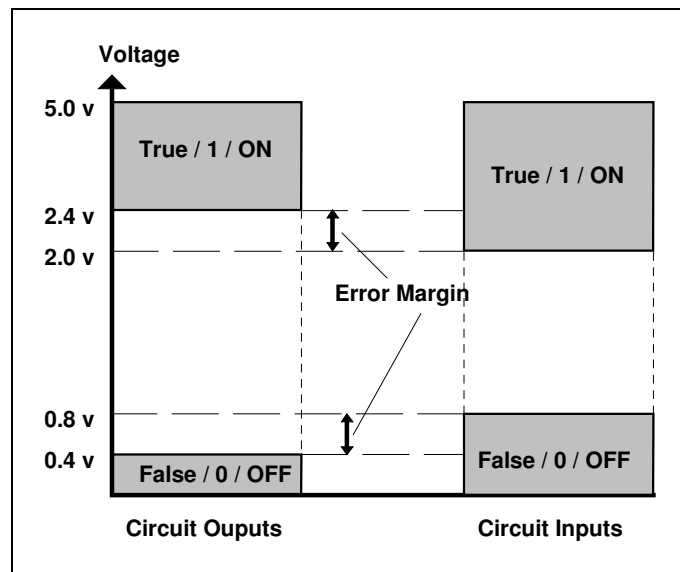


Figure 4.18 - Input and Output Voltage Levels Associated with TTL

The acceptable input and output voltage levels for TTL have been designed on the assumption that no logic gate is ideal and that its deviation from the ideal is restricted by good design. A primary consideration in gate design is the loading. Only a limited number of gates can be connected to the output of a TTL gate before its performance suffers. When TTL gates output a logical "High" they act as a source of current and when they output a logical "Low" they act as a sink for current.

A gate's ability to source or sink current determines the maximum number of gates which can be connected to its output - that is, its "fan-out". If too much current is drawn from the output when a gate is in the high state, the current will eventually drag the gate down to a logical low, which is clearly unacceptable. The typical maximum output current from a TTL gate is in the order of a few milli-Amps and permissible fan-outs are normally in the order of 10.

Fan-out not only affects output voltage levels but also gate performance. Figure 4.19 is a time-scale enlarged diagram showing the output of an inverter gate (in response to a changing input) when the output is loaded with one gate and then with ten gates.

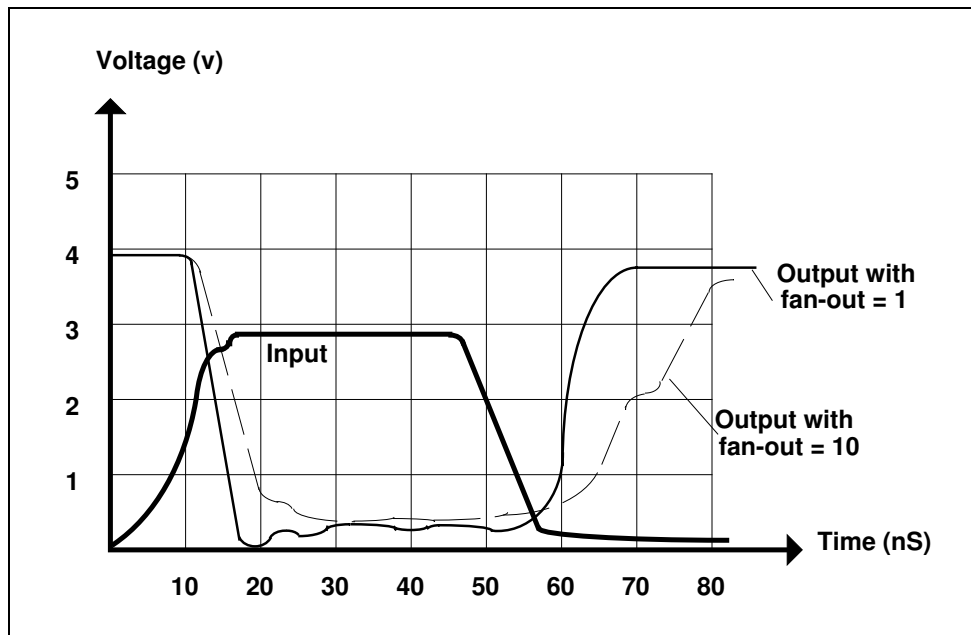


Figure 4.19 - Effect of Fan-Out on TTL Gate Performance

Notice in Figure 4.19 how the performance of a TTL gate suffers as a result of extra loading. Switching times are increased. In the case of most TTL gates, such as the inverter shown in Figure 4.17, the most notable effect of loading is that the transition from low to high is affected. This is due to the fact that the totem-pole transistor Q4 is heavily saturated when it is sinking a large load current. The time taken to move this transistor from saturation back to cut-off is affected by the level of transistor saturation. When only one gate is connected to the output of Q4, the transistor is only just saturated and can recover more quickly.

Most semiconductor data books do not show detailed diagrams of gate performance as in Figure 4.19. Rather, a simplified approach is taken towards displaying the time performance of logic gates. This is shown in Figure 4.20 for the inverter gate.

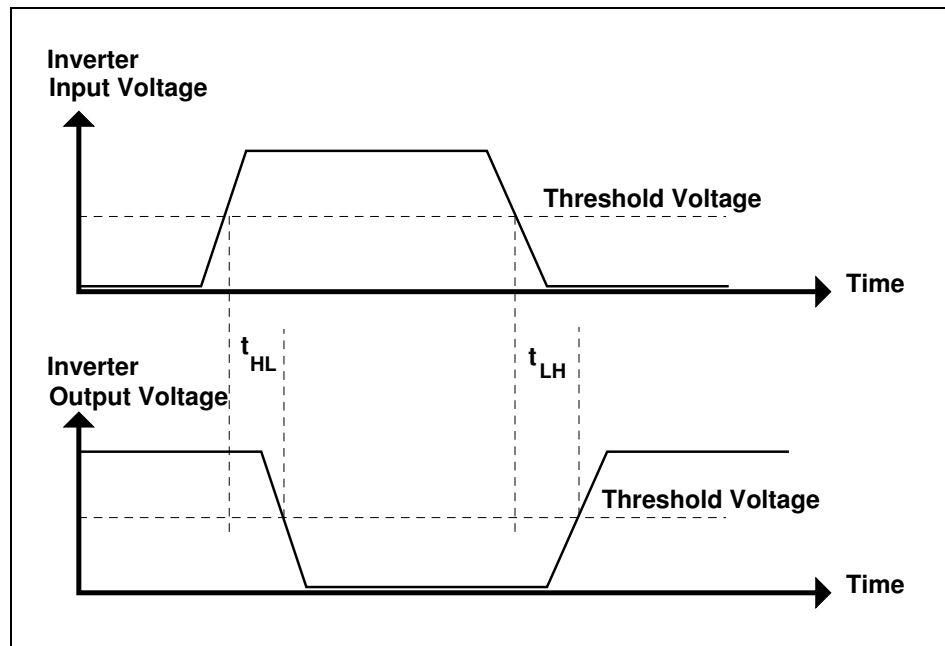
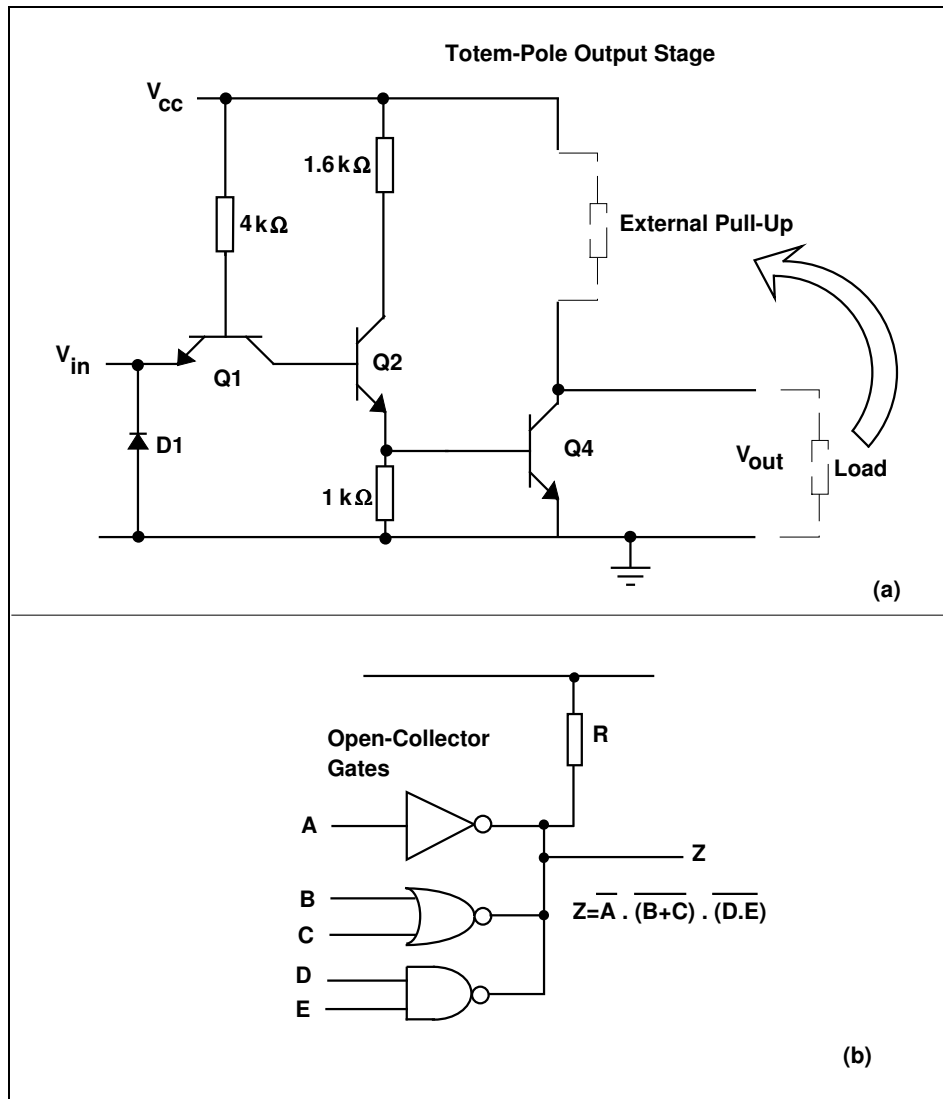


Figure 4.20 - Typical Data-Book Timing Diagrams Illustrating Propagation Delays from Gate Input to Gate Output

In Figure 4.20, we see a time-scale enlarged diagram approximating the behaviour of gate inputs and outputs in the case of an inverter. The transition from high to low (HL) or from low to high (LH) is defined to occur when the input or output voltage reaches some predefined threshold level (typically around 1.5 volts in the case of TTL).

The totem-pole output stage of the inverter gate shown in Figure 4.17 has what is referred to as an "active" pull-up section composed of Q3, the 130Ω resistor and a diode. A modified version of TTL removes this active pull-up stage and is referred to as open-collector TTL. The resistor can be supplied externally or can even be the "load" (such as a light-emitting diode, etc.) which is activated whenever the normal output (V_{out}) is low. An open-collector version of the inverter gate is shown in Figure 4.21 (a). Figure 4.21 (b) shows how a number of these gates can be interconnected to create a wired "AND" function, thus sparing one additional gate.



**Figure 4.21 - (a) Open-Collector TTL Inverter
(b) Combining Open-Collector Gates to Create an "AND" Function**

TTL gates are normally available in IC packages that contain several of the same devices. TTL based ICs cover all the common Boolean logic functions including:

- NOT (Invert)
- AND
- NAND
- OR
- NOR
- XOR.

However, it is interesting to note that any Boolean combinational logic can be realised using only NAND gates because all the other gates can be created from NAND gates. The reason for the other functions is really to minimise the number of chips required to create a circuit - that is, minimise the "chip-count".

Minimising the chip-count in a circuit is much more important than minimising the number of gates because extra chips add to the size of a board, the complexity of the wiring and the energy requirements of the board. When we examined Karnaugh Mapping, our objective was to minimise Boolean logic expressions. However, we have to follow this up with an analysis of how many gates are available on each chip and how many chips we need to make a given logic.

Another point that needs to be noted with TTL gates is that although we have only looked at 2-input gates, functions such as NAND, NOR, etc. are normally available with a range of inputs in order to simplify circuits. For example, it is possible to purchase 2, 3, 4 and 8 input NAND gates so that we can adjust designs to minimise the logic circuitry.

4.5.3 Schottky TTL

Digital circuits, have the same trade-off problems as most other modern devices. One of the most prevalent is the speed versus power consumption issue. High-speed logic generally uses more power than low-speed logic. Standard TTL is relatively fast compared to circuits built from MOSFETs but it dissipates significantly more energy. Energy consumption is an important issue because many digital logic devices are designed for battery operation and so energy usage has to be minimised in order to provide acceptable battery life levels.

Standard TTL can be modified in a number of ways in order to vary the speed versus power consumption trade-off. Increasing resistance values in TTL gates decreases speed but also reduces power consumption. Decreasing resistance values increases both speed and power consumption. Both High-Speed TTL and Low Power TTL have been implemented by semiconductor companies.

Schottky TTL is another interesting variation on standard TTL. The term "Schottky" refers to a special type of diode which is also known as a barrier diode or hot-carrier diode. The Schottky diode can be used in conjunction with a Bipolar Junction Transistor (BJT) to prevent that transistor from completely saturating. This is achieved by connecting the anode of a Schottky diode to the base of a BJT and the cathode to the collector of the BJT.

In Section 4.5.2, we noted that one cause of propagation delays in gates (ie: performance degradation) is the level of saturation in transistors. If transistors can be kept from complete saturation then the performance of a logic gate can be increased and hence the development of Schottky TTL. In Schottky TTL, the diodes are actually fabricated as part of the transistors, rather than as separate elements.

Logic circuits developed using Schottky TTL are much faster (typically three to five times faster) than those using normal TTL. This translates into another advantage in that it is possible to fabricate low power consumption gates with Schottky TTL (using higher resistance values) that still perform as fast as standard TTL.

4.5.4 Emitter Coupled Logic (ECL)

So far we have looked at a number of different gates that have all been based upon the Bipolar Junction Transistor (BJT). As a general rule, digital circuits based upon BJTs are significantly faster than those based upon Field Effect Transistors (MOSFETs, CMOS, etc.). However, we also noted that standard TTL can be varied to minimise power consumption or maximise speed. This can be achieved by variation of resistance values in the gate circuits. Performance can also be improved through the addition of Schottky diodes that prevent critical transistors from heavily saturating. All of these techniques have been used to produce commercially available integrated circuits.

For some time-critical applications, none of the simple variations on TTL, cited above, can produce a sufficiently high switching speed. In these instances it is necessary to use a special type of high performance logic that is still based upon BJTs but uses a more complex circuitry. This high performance logic is referred to as Emitter Coupled Logic or ECL.

The core of all ECL gates, as the name suggests, is a pair of transistors whose emitters are joined (coupled) together. The speed of ECL digital circuits is achieved through the use of high speed transistors and the use of a coupled-pair where transistors mutually prevent one another from saturating heavily.

The problem with ECL is that it is much more difficult to work with than other forms of logic. Primarily, this is because the circuits tend to be operated at very high switching speeds, thereby emphasising parasitic effects in wires and so on. Most circuits designed using ECL devices have to be built using the so-called "ground-plane" construction technique. In this form of construction, one side of the circuit board is covered with conducting material (normally copper sheet) and this sheet is the ground point for the circuit. The objective of this technique is to avoid using wires to connect devices to ground. The inductance associated with ground "wires" is sufficient (at high switching speeds) to create an additional circuit between two wired ground points and this is referred to as an "earth-loop". This leads to unwanted noise in ECL circuits.

Ground wires are not the only wires that cause problems in ECL circuit design. All wires need to be kept as short as possible within a circuit board. If ECL devices need to be interconnected across two circuit boards then a transmission line (such as a twisted-pair line or coaxial cable) link needs to be implemented thereby increasing the cost and complexity of the system.

As a result of the practical difficulties of working with ECL it is normally reserved for applications that cannot be served by the more traditional logic circuits.

4.5.5 CMOS Logic

All the BJT based circuits that we have looked at have been primarily designed for high-speed operation. However, there are a number of problems with BJT based circuits. Firstly, BJTs are physically much larger than Field Effect Transistor based devices such as MOSFETs. This means that we cannot create high-density circuits using BJT technology. The second problem is that the gate circuits built from BJT devices are heavy consumers of power when compared to MOSFET based devices. This means that they are less desirable for battery operated systems, where power consumption can be a serious issue.

The more common, modern technology for general-purpose applications that do not require high-speed switching is therefore based upon MOSFETs, and in particular, circuits containing one p-channel and one n-channel pair. This is referred to as Complementary MOSFET logic or CMOS. This form of digital logic is extensively used in memory circuits, microprocessors and covers the spectrum of devices commonly found in Small, Medium and Large Scale Integrated (SSI, MSI and LSI) circuits.

The circuit for an inverter gate is shown in simplified form in Figure 4.22. The diagram omits a number of diodes which are integrated across each of the MOSFET devices for protection purposes.

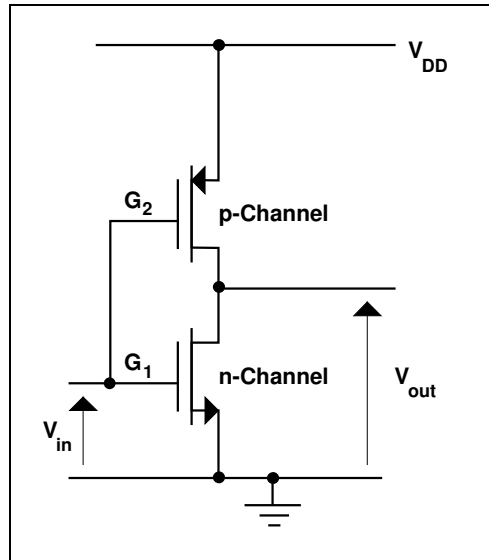


Figure 4.22 - CMOS Inverter Including "p" and "n" Channel Devices

The operation of CMOS logic is conceptually very simple to understand. A high input to the CMOS pair turns one MOSFET on and the other off. A low input to the CMOS pair causes the reverse switching to occur. In the case of the inverter, a high input causes the n-channel device to short-circuit the output to ground. A low input causes the p-channel device to short-circuit the output to the supply rail. The result is an output voltage which is the Boolean NOT of the input. As with most other forms of logic, CMOS provides a broad spectrum of gates, including AND, OR, etc. which can provide the same functionality as TTL. In many cases, voltage levels are kept compatible with TTL.

There are two distinct disadvantages with CMOS devices. The first, as we have already noted, is that they are slower than TTL type devices. The second is that CMOS circuits are quite susceptible to damage from voltage spikes that can break down the insulating oxide layer in the MOSFETs. Despite the diode protection that is built into CMOS circuits, even static charges from humans handling CMOS chips can be sufficient to cause irreparable damage. Static charges, although very low in energy, are still associated with high voltages capable of destroying the delicate oxide layer in the MOSFETs. CMOS devices therefore need to be handled with care, particularly where synthetic floor-coverings can be responsible for the build-up of static. Moreover, when using CMOS devices in circuits, all inputs must be terminated in a manner that allows for a current discharge path to the circuit ground.

Despite the disadvantages associated with CMOS, devices formed from this integrated-circuit technology are amongst the most prolific digital circuits currently available.

4.6 Digital Circuits for Arithmetic Functions

Section 4.5 provided an important break from the primary objective of this chapter, which has been written to show how digital circuits can be designed in order to carry out the two forms of human reckoning that we wish to instil into computers, namely:

- Simple human reasoning (Boolean logic)
- Arithmetic manipulation.

The reason for the departure of Section 4.5 is because one needs to understand the different forms of technology available to produce the various digital circuits that were described earlier (in design problems) and those which are to follow.

In Section 4.4, we examined some simple circuits that could be used to provide some degree of human reasoning in low level control applications. In this section, we return to the design of digital circuits in order to pursue the development of systems that can be used to manipulate numbers. We already know that the sort of numbers, with which humans must work, can be represented in a binary form as strings of digits. At the very least, we would expect computers to be able to carry out functions such as the addition and subtraction of numbers that had been converted into a binary form. To begin with, we can examine the simplest of these circuits - that is, one that can add two binary digits. In order to design such a circuit, the starting point, as with all other digital designs, is a truth table. A truth table is shown in Table 4.8 for two binary digits, A and B that are added to produce a sum, S.

<i>A</i>	<i>B</i>	<i>S</i>
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.8 - Truth Table for Simple Addition Circuit

The truth table shown in Table 4.8, doesn't really represent a "full" adder circuit. When we add the binary numbers "1" and "1" we should end up with binary "10" - or more specifically, zero carry one. However, for a simple addition circuit, there is no carry. This is referred to as a "half-adder circuit". We could create a Karnaugh Map for Table 4.8 or a sum-of-products expression in order to generate the circuit. However, it is obvious from looking at the table that S is the exclusive-OR of A and B. That is:

$$S = A \oplus B$$

The half-adder circuit is obviously not very useful for any sensible addition of binary digits and so we need to design a full-adder circuit. In a full-adder circuit, such as the one shown in Figure 4.23, we have two binary inputs and one carry input that are added to produce both a sum and a carry output. The truth table is shown in Table 4.9.

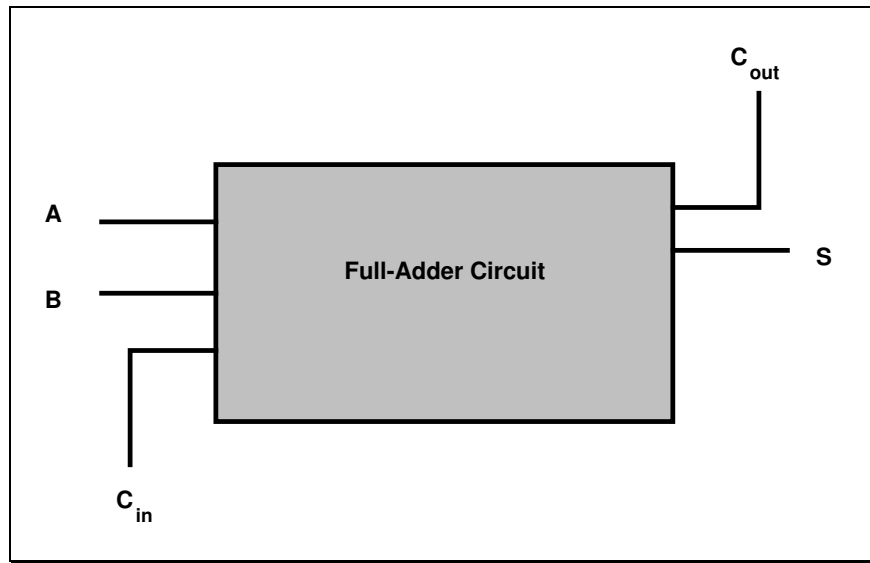


Figure 4.23 - Block Diagram for Full-Adder Circuit

<i>A</i>	<i>B</i>	<i>C_{in}</i>	<i>S</i>	<i>C_{out}</i>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 4.9 - Truth Table for Full-Adder

In order to derive the simplest expressions for the outputs in the full-adder circuit, we really need to go directly to the Karnaugh Mapping technique. The Karnaugh Maps for the outputs *S* and *C_{out}* are shown in Figure 4.24.

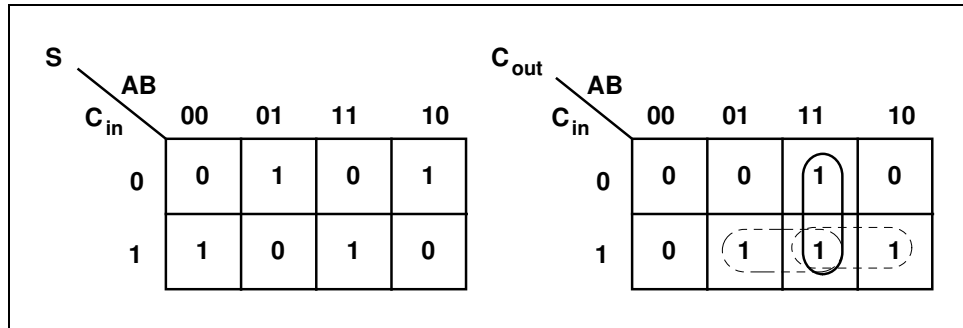


Figure 4.24 - Karnaugh Maps for Full-Adder Circuit

The Karnaugh Map for the output sum, S, cannot be simplified by normal techniques since there are no groupings. However, as discussed in Design Problem 3, earlier in this chapter, the pattern in the map corresponds to a known pattern for the exclusive-OR gate. The expression for S is:

$$S = A \oplus B \oplus C_{in}$$

The Karnaugh Map in Figure 4.24 for the output carry variable has three paired regions which lead to the following, simplified expression:

$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

The circuit diagram for the full-adder circuit can be derived directly from the above expressions.

A number of full-adder circuits can be cascaded so that two digital numbers composed of multiple bits can be added together. For example, say that we wished to design a full-adder circuit that could add two 8-bit numbers as follows:

$$\begin{array}{r}
 A_7A_6A_5A_4A_3A_2A_1A_0 \\
 + \\
 B_7B_6B_5B_4B_3B_2B_1B_0 \\
 \hline
 C_{out7} + S_7S_6S_5S_4S_3S_2S_1S_0
 \end{array}$$

The 8-bit full-adder circuit is shown schematically in Figure 4.25. Note that if the high-order output carry (ie: the output from the eighth adder, C_{out7}) is high, then an overflow error has occurred because the result is nine bits long.

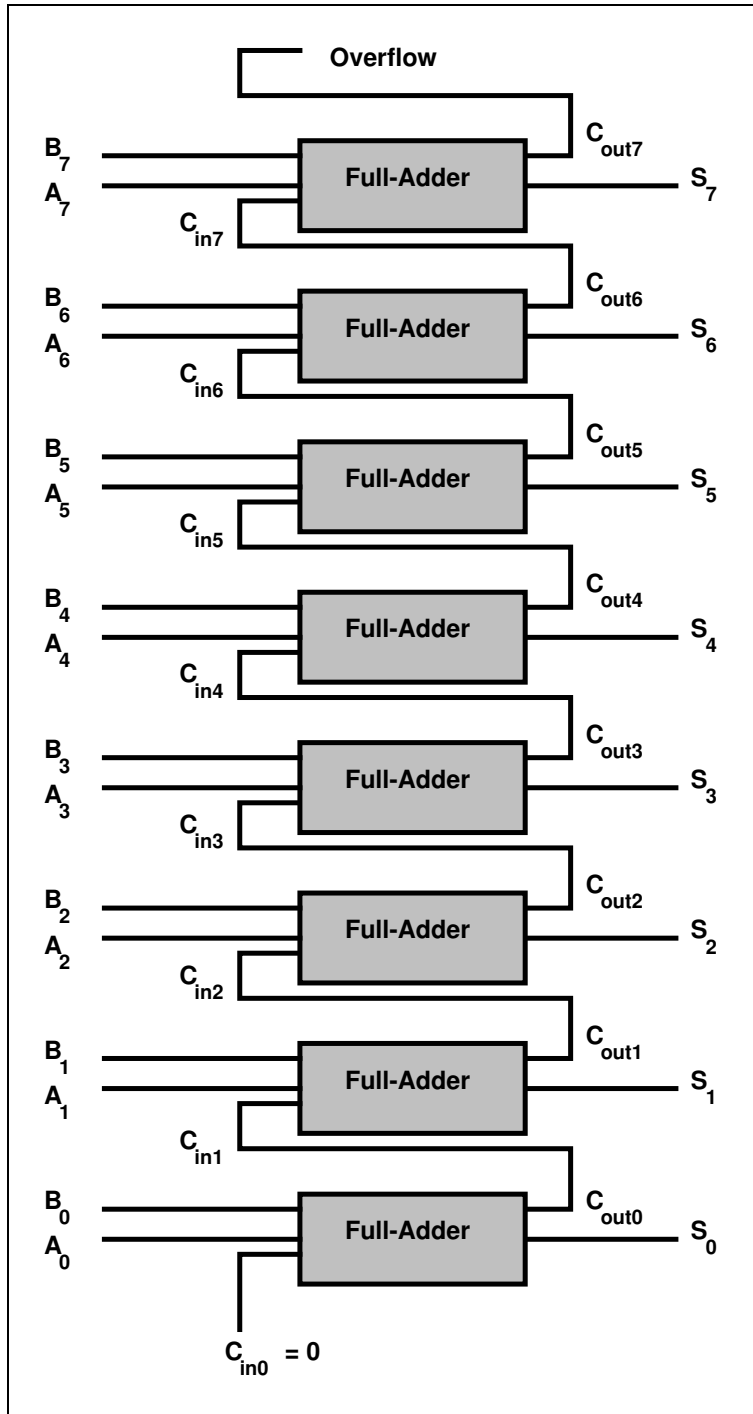


Figure 4.25 - A Circuit to Add two 8-Bit Numbers "A" and "B"

As an exercise, you should attempt to design a circuit that will subtract one 8-bit number, B (the subtrahend), from another 8-bit number, A (the minuend), assuming that A is greater than or equal to B:

$$\begin{array}{r}
 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 \quad (\text{Minuend}) \\
 - \quad B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 \quad (\text{Subtrahend}) \\
 \hline
 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
 \end{array}$$

The problem with designing subtracter circuits is that when the subtrahend is larger than the minuend, we end up with a negative number. We clearly need methods of representing negative numbers. If we can find such techniques, then we don't really need special subtracter circuits because we can simply use addition circuits and add a negative number instead of a positive number. There are three techniques by which we can perform binary subtraction. These are all dependent upon the way in which we represent negative numbers and include:

- (i) One's Complement
- (ii) Two's Complement
- (iii) Signed Numbers.

These methods are discussed below:

(i) ***One's Complement Arithmetic***

The one's complement of an n-bit number is obtained by taking the Boolean NOT of every bit - that is, inverting each bit. In order to subtract an n-bit number, B, from an n-bit number, A, using one's complement arithmetic, the following procedure is followed:

- Take the one's complement of B
- Add the one's complement of B to the number A
- If the overflow bit is one, then the result is positive and the overflow bit is added to the result (this is referred to as an "end around carry")
- If the overflow bit is zero, then the result is negative and its magnitude is obtained by taking the one's complement of the result (excluding the carry bit).

As an example, consider the following subtraction:

$$\begin{array}{r}
 \begin{array}{r}
 \text{_} 11100011 \\
 00100010 \\
 \hline
 11000001
 \end{array}
 \quad \longrightarrow \quad
 \begin{array}{r}
 \begin{array}{r}
 + \text{_} 11100011 \\
 \text{_} 11011101 \\
 \hline
 1 \text{_} 11000000 \\
 \longleftarrow + 1 \\
 \hline
 11000001
 \end{array}
 \end{array}
 \end{array}$$

In this instance the results are verified by a simple examination of the original expression.

(ii) *Two's Complement Arithmetic*

The two's complement of an n-bit number is obtained by adding binary one to the one's complement of that number. In order to subtract an n-bit number, B, from an n-bit number, A, using one's complement arithmetic, the following procedure is followed:

- Take the two's complement of B
- Add the two's complement of B to the number A
- If the overflow bit is one, then the result is positive and the overflow bit is ignored
- If the overflow bit is zero, then the result is negative and its magnitude is obtained by taking the two's complement of the result (excluding the carry bit).

As an example, consider the following subtraction:

$$\begin{array}{r}
 \begin{array}{r}
 \text{_} 11100011 \\
 00100010 \\
 \hline
 11000001
 \end{array}
 \quad \longrightarrow \quad
 \begin{array}{r}
 \begin{array}{r}
 + \text{_} 11100011 \\
 \text{_} 11011101 \\
 \hline
 \text{_} 1 \\
 \hline
 \text{_} 11000001
 \end{array}
 \end{array}
 \end{array}$$

Again, the results can be verified by a visual examination of the original binary subtraction.

(iii) Signed Numbers

In a signed number system, one bit out of each n-bit number (the high-order bit) is used to represent the sign of that number. Zero represents positive and one represents negative. For example, in an 8-bit system, the number +5 is represented as follows:

00000101

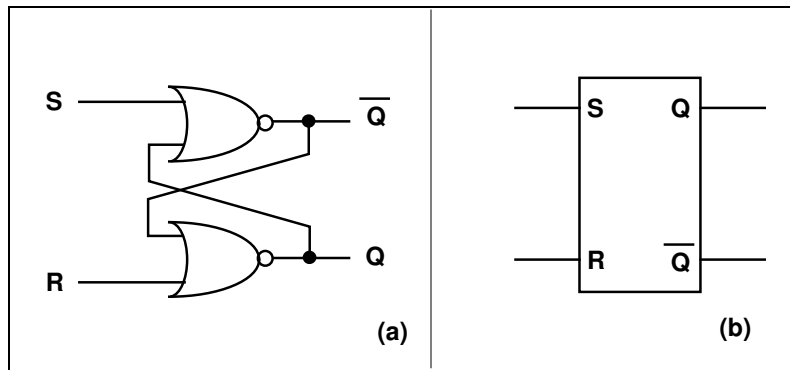
The number -5 is represented as follows:

10000101

The high-order bit is referred to as the "sign bit". The problem with signed number representation is that it fails to take advantage of special purpose hardware available on most processors that performs one's or two's complement subtraction as shown in (i) and (ii).

4.7 Flip-Flops and Registers

In the previous sections of this chapter, we examined a range of circuits that were designed to perform some instantaneous logic or arithmetic function - for example to produce a control output based upon inputs or to instantaneously add numbers together. However, in computer systems we need to do far more than add numbers together, we also need to be able to store the results. The most basic storage element in digital systems is called the flip-flop. The simplest of all flip-flops is the S-R flip-flop which is constructed from two NOR gates as shown in Figure 4.26.



*Figure 4.26 - (a) Schematic of R-S Flip-Flop Construction
(b) Block Diagram Representation of R-S Flip-Flop*

The R-S flip-flop looks deceptively simple but it is not a combinational circuit. The difference is that in a flip-flop, the outputs are fed back into the inputs. This means that the output "Q" is influenced by both the current inputs (R and S) and the current value of Q. For this reason, we can refer to the current value of Q as Q_n and the next value of Q as Q_{n+1} . Moreover, since the flip-flop output is dependent upon its own current value, combined with inputs, we refer to Q as being the "state" of the flip-flop, rather than just the output. Notice also, that the flip-flop has two outputs which, from symmetry are complementary (ie: one is the NOT of the other). The truth table for the R-S flip-flop is shown in Table 4.10.

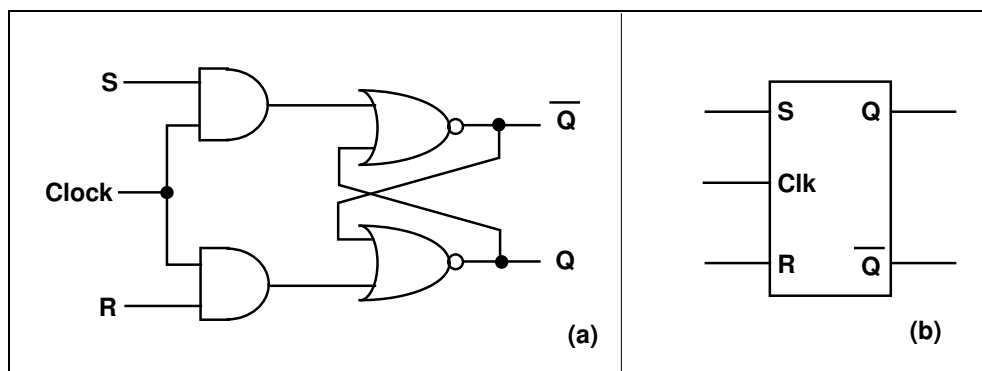
The truth table is easy to determine when only one or other of the inputs is high. However, when both inputs are high, then the output cannot be determined from simple Boolean logic and really only depends upon the switching speed of the gates involved. The condition where both gates are set to high is referred to as a "race condition" and is a "not allowed" state because it is not defined by digital logic.

S	R	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	Not Defined
1	1	1	Not Defined

Table 4.10 - Truth Table for R-S Flip-Flop

Looking at the truth table of Table 4.10, it can be seen that the effect of "S" is to change the output state, Q, from 0 to 1. The effect of "R" is to change the output state from 1 to 0. For this reason, S and R are referred to as "Set" and "Reset", respectively. A further inspection of Table 4.10 reveals that once the output state of an R-S flip-flop is 1, it remains equal to 1 regardless of the value of S and until the value of R is set to 1. Similarly, the output state of a flip-flop remains equal to 0, regardless of R and until the value of S is set to 1. In other words, the flip-flop acts like a "latch" and it is often referred to by this alternate name.

Despite the problem of the race condition, R-S flip-flops are still used in a number of circuits. In particular, the R-S flip-flop circuit can be modified so that the inputs are only allowed to influence the outputs when additional requirements are fulfilled. The so-called "clocked" R-S flip-flop is one such circuit and is shown in Figure 4.27.



**Figure 4.27 - (a) Clocked R-S Flip-Flop Schematic
(b) Clocked R-S Flip-Flop Block Diagram**

In "clocked" digital devices, the inputs are gated through an additional binary variable, abbreviated "Clk", and can only influence outputs when this variable is high. A clock waveform can be generated by a number of mechanisms, including a crystal oscillator or a special digital IC known as a "555 Timer". Although a clock can be just another binary signal, it is more typically a repetitive binary waveform, as its name implies. A typical set of waveforms for a clocked R-S flip-flop are shown in Figure 4.28.

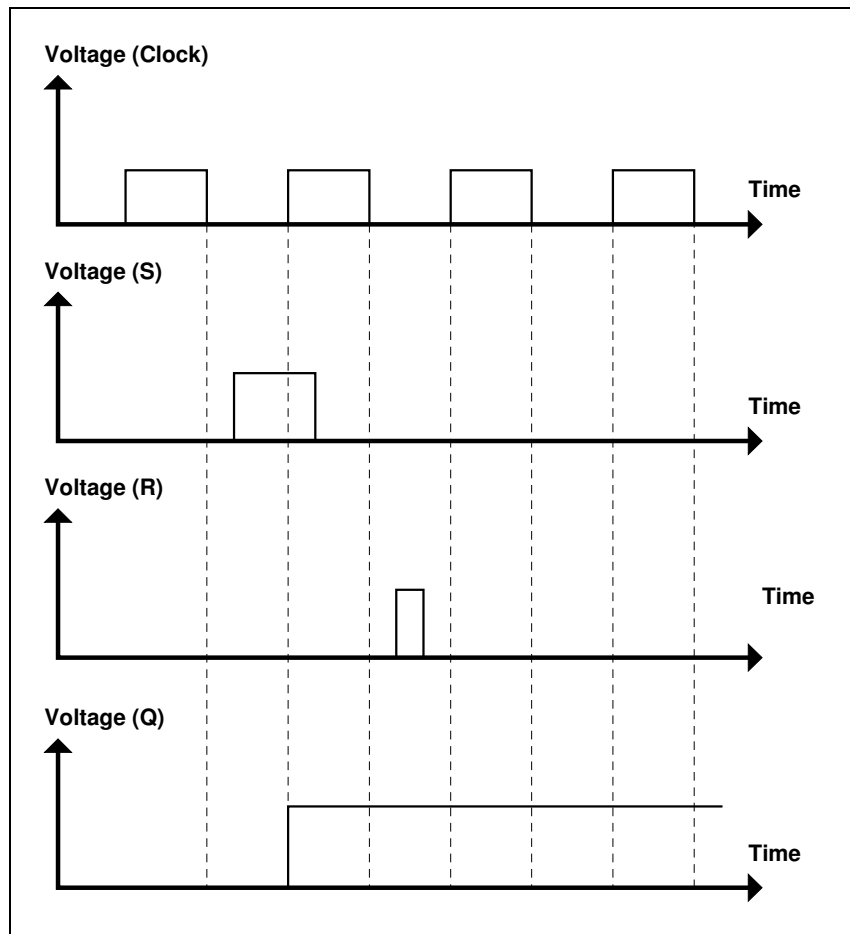
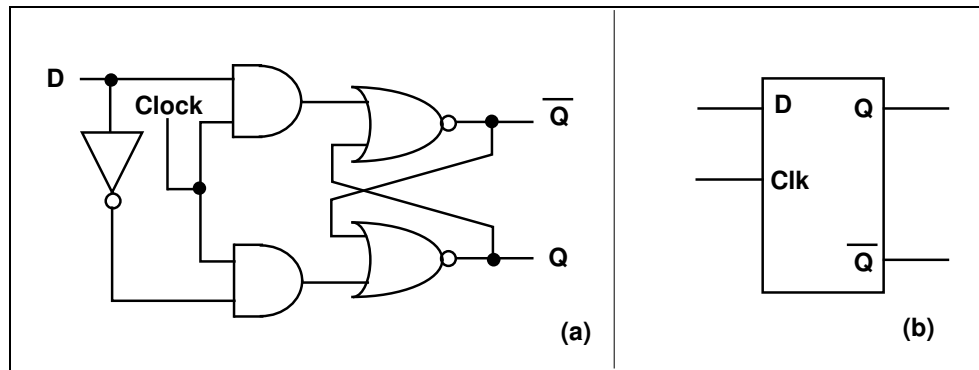


Figure 4.28 - Typical Timing Diagrams for a Clocked R-S Flip-Flop

The clocked R-S flip-flop arrangement does not eliminate the "race" condition, since it is still possible that both inputs will be high when the clock is high. One simple technique for avoiding the race condition is to ensure that S and R are never simultaneously high. This is achieved by connecting an inverter between S and R as shown in Figure 4.29. The result is a single-input flip-flop. The single-input is called "D" and the flip-flop is called a "D flip-flop".



**Figure 4.29 - (a) Clocked D Flip-Flop Schematic
(b) Clocked D Flip-Flop Block Representation**

A more sophisticated flip-flop that is very commonly used in digital circuits is the so-called JK flip-flop and J and K inputs are analogous to S and R, respectively. The JK flip-flop behaves identically to the R-S flip-flop whenever one or other of the inputs are high - however, the JK flip-flop does not suffer from the race condition and when both inputs are high, the outputs toggle (invert) from their previous value. The truth table for a JK flip-flop is shown in Table 4.11. Note that if Q_n is 0 and both J and K are set to 1 then Q_{n+1} is 1. If Q_n is 1 and J and K are both set to 1, then Q_{n+1} becomes 0. JK flip-flops, like R-S flip-flops are available in a clocked form.

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1 (Toggle)
1	1	1	0 (Toggle)

Table 4.11 - Truth Table for JK Flip-Flop

The problem with all clocked flip-flops is that it is very difficult to produce clock pulses such as those shown in Figure 4.28 - that is, uniform. Flip-flop circuits that depend upon uniform timing from clocks are subject to faulty state changes. For this reason, it is much more practical to design flip-flops that only change state on the positive or negative going edges of a clock pulse, thereby making them independent of the clock pulse width. These flip-flops are referred to as "edge-triggered" devices. A flip-flop which changes state as the clock pulse goes from high to low is referred to as "negative-edge-triggered" and one that changes state as the clock goes from low to high is referred to as "positive-edge-triggered".

The problem with edge-triggered flip-flops is that they can miss small input pulses that do not coincide with the positive or negative edge of the clock waveform. For this reason, the so-called JK-Master-Slave flip-flop has been developed. This is shown in Figure 4.30.

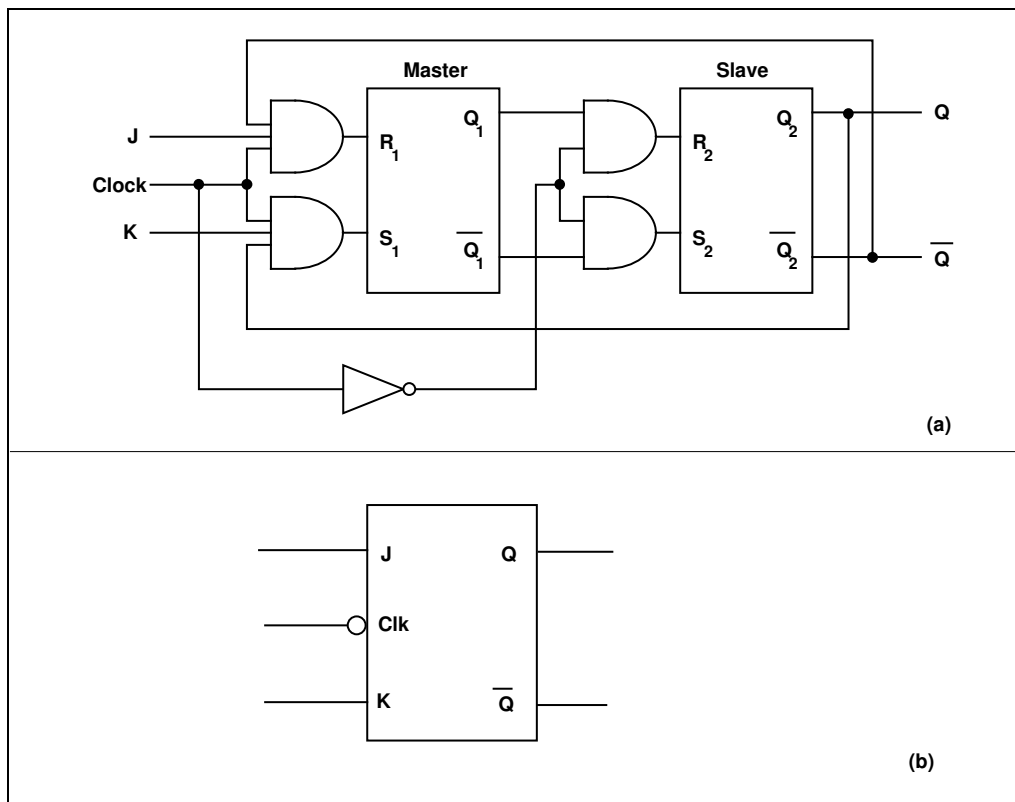


Figure 4.30 - Negative-Edge-Triggered JK Master-Slave Flip-Flop
(a) Schematic (b) Block Diagram Form

Note from Figure 4.30 (b) how a small circle is placed in front of the Clk input terminal on the flip-flop to indicate that the flip-flop is triggered on the negative edge. Figure 4.31 shows the timing diagram for the master-slave flip-flop of Figure 4.30 and compares it with the standard, negative edge triggered device that "misses" small input pulses. In the master-slave device, the master section is active whenever the clock is high, so the input values (whenever the clock is high) are registered and they can only change the outputs when the slave becomes active (on the negative edge).

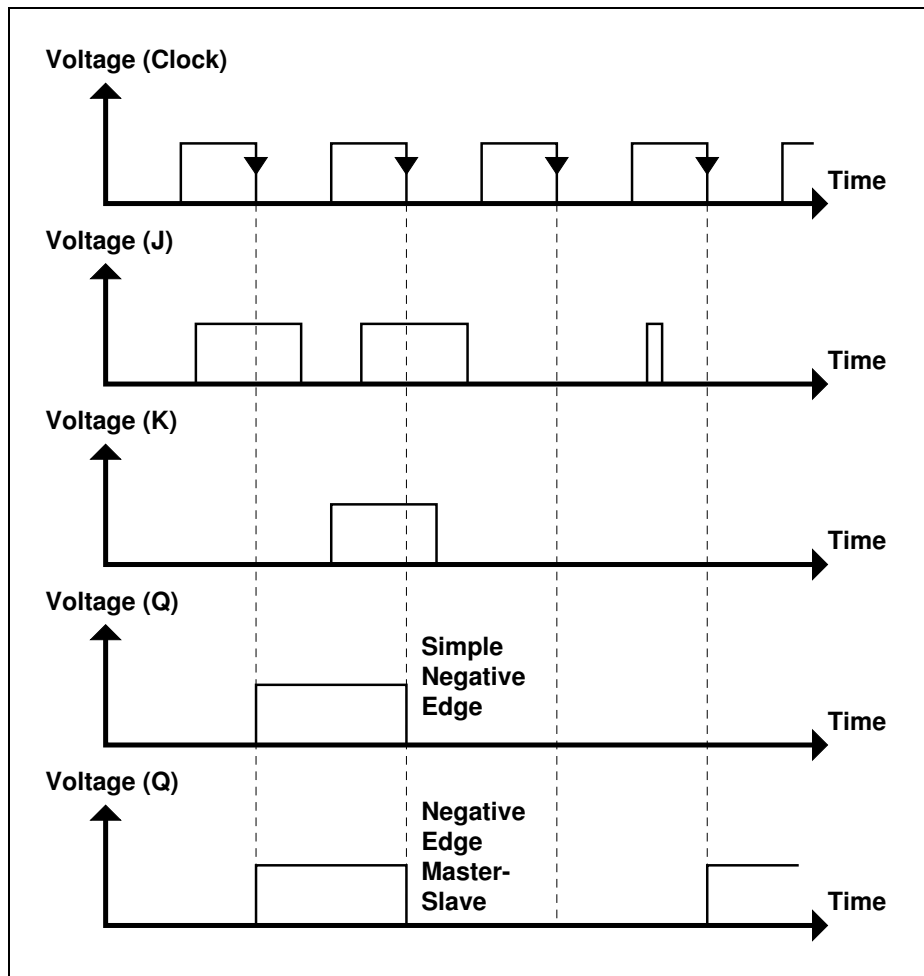


Figure 4.31 - Timing Diagrams Highlighting the Operation of Simple Negative Edge Triggered Flip-Flop and Master-Slave Flip-Flop

In addition to the normal inputs, such as J, K, S, R and Clk, most flip-flops have two additional (asynchronous) inputs, normally referred to as PRESET and CLEAR. Whenever a PRESET terminal is enabled, the flip-flop output is immediately set to 1, regardless of other inputs and clock state. Whenever CLEAR is enabled, the flip-flop output is immediately set to 0, regardless of other inputs and clock state.

A collection of flip-flops connected together for some common purpose is referred to as a register. There are many different kinds of registers to perform a range of different functions. The most basic function of a register is to act as a data storage device, with each flip-flop storing 1 bit of data. This is shown in Figure 4.32. The "state" of the register at any point in time is defined by the outputs (Q_n to Q_0) of the flip-flops in the register.

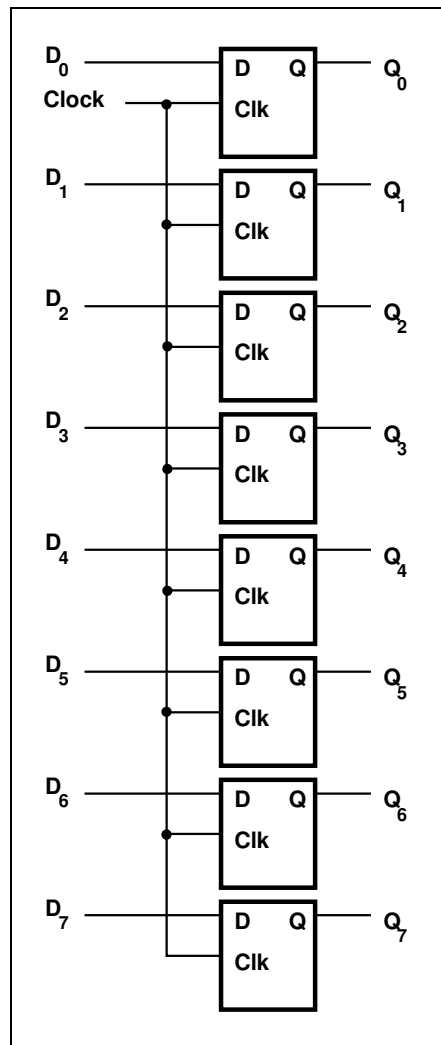


Figure 4.32 - Eight Bit Storage Register

The device in Figure 4.32 is simply an 8-bit storage location. Data is clocked in (in parallel) and outputs are held in flip-flops until required. Figure 4.33 shows a slightly more sophisticated form of register, called a "shift register". The purpose of a shift register is to take an incoming serial bit stream at the input and shift the data along from flip-flop to flip-flop until all are loaded with one bit of data. In the case of the shift register in Figure 4.33, if the first bit in the serial stream is the least significant bit (LSB) and the last is the most significant bit (MSB), then after eight clock pulses, Q_0 contains the LSB and Q_7 contains the MSB. The shift register has effectively converted serial data into parallel form.

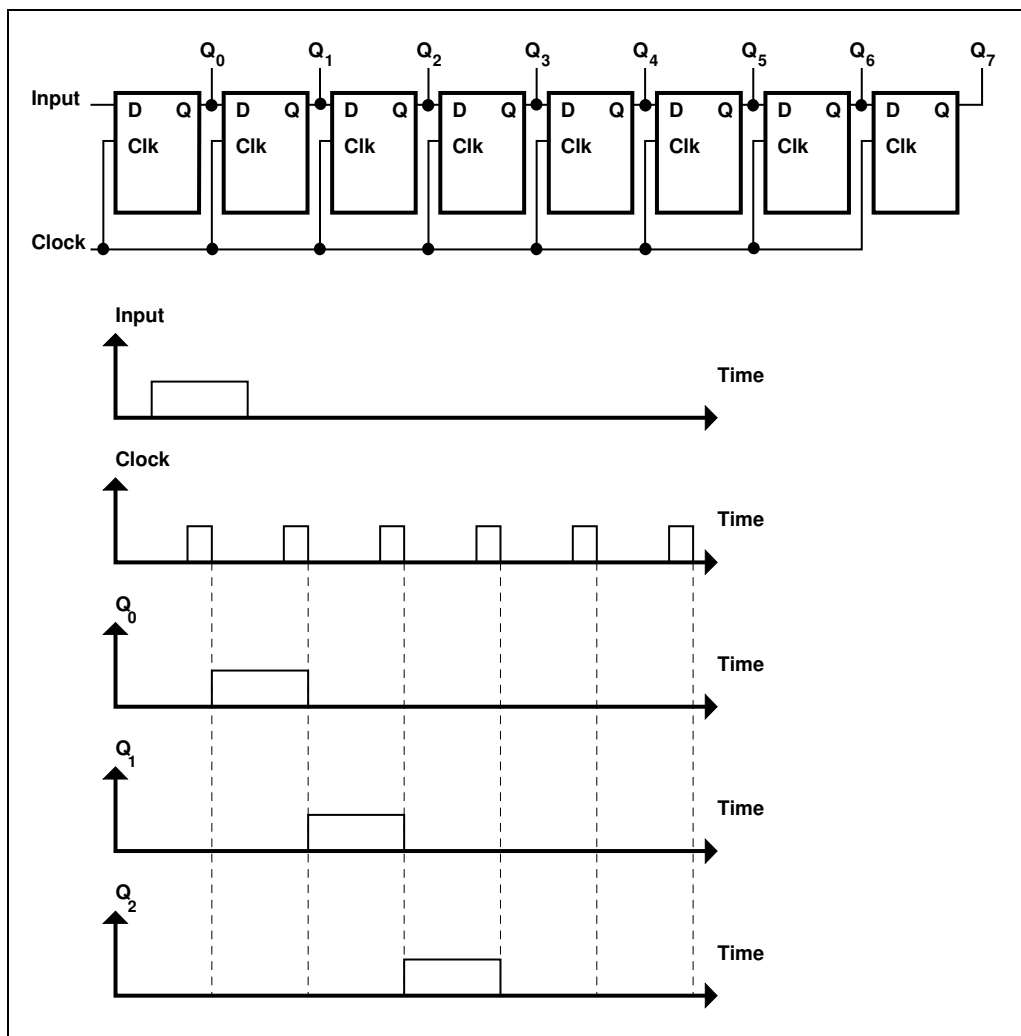


Figure 4.33 - Eight Bit Shift Register and Timing Diagrams

4.8 Counters - Electronic Machines

At the beginning of this chapter, we examined the possibility of generating an electronic machine that could actually move from one state to another. If each state can then be decoded using both Boolean logic and numerical circuits, then we can effectively produce a reckoning machine or computer. However, nearly all the Boolean logic and flip-flop circuits that we have examined in this chapter have been rather static in nature, except for the shift register of Figure 4.32. In the shift register, the state changed with every cycle of the clock. In this section we shall examine a few basic circuits that change from one state to another on each clock cycle and act as "state machines".

The simplest state machines are actually counters and we shall see why they are so named after examining Figure 4.34. In this diagram, four JK flip-flops are interconnected to produce an "asynchronous up-counter". Each flip-flop has its J and K inputs tied to binary one and hence each is in toggle mode. Assuming that each flip-flop is negative-edge-triggered, then each output will invert on the negative edge of each clock cycle. The timing diagram in Figure 4.34 shows that the output Q_0 then effectively runs at half the clock frequency. Since Q_0 is connected to the clock input of the second flip-flop, Q_1 then runs at one quarter of the original clock frequency and so on. Table 4.12 lists the outputs Q_3 to Q_0 on each successive negative edge of the clock pulse.

<i>Pulse</i>	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
16	0	0	0	0

Table 4.12 - Flip-Flop States for Asynchronous "Up-Counter"

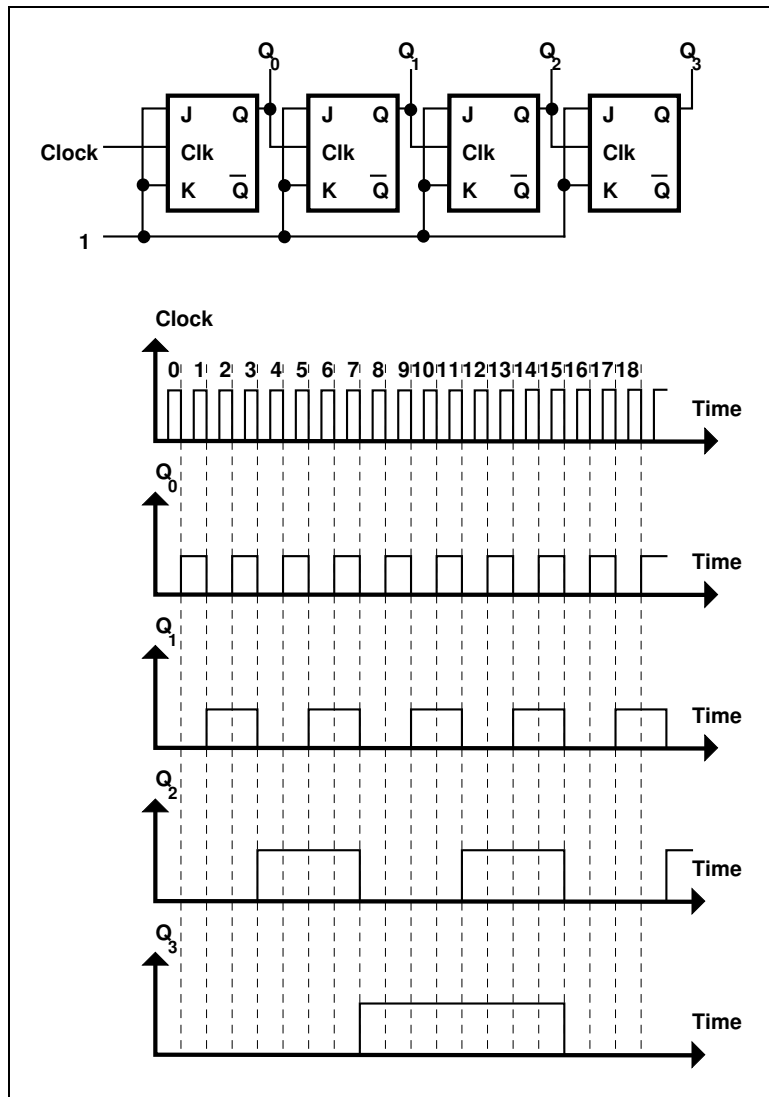


Figure 4.34 - Asynchronous Up-Counter

The asynchronous up-counter of Figure 4.34 is composed of four flip-flops and hence can generate a maximum of sixteen different states (ie: 2^4), before we return to the original state. The sixteen different states produced by the up-counter follow a standard binary count sequence (from zero to fifteen) and hence the name of the circuit. In order to make a device that counts down from fifteen to zero, that is, an asynchronous down-counter, we take the inverted output from each flip-flop and connect it to the clock of the subsequent flip-flop (the output states are still taken from the non-inverting outputs).

From the above discussions and diagrams it becomes evident that counters are analogous to the electronic machines that we were seeking at the beginning of this chapter. We will look at state machines in a little more detail in Chapter 6 in order to show how we can use basic digital logic to create an electronic machine that is capable of carrying out some form of reckoning. At this stage however, it is more important to consider what we can do with state machines such as counters.

The self-evident application for counters is to count numbers so that they can provide a digital output proportional to time (ie: clock pulses). Another use for counters is to carry out some form of sequential control. As an illustration of the use of counters in control functions, and as a conclusion to this chapter, consider the following problem.

Design Problem 5:

Referring back to the incubator control system of Design Problem 1, modify the original design so that the incubator temperature is only read once every ten seconds. After reading the temperature, the fan or heater should be turned on as before, but only for a period of three seconds and switched off thereafter until the next reading.

Solution to Design Problem 5:

In order to solve the problem, we need add a counter circuit to the original system and two flip-flops to store the required state of the fan/heater at the time the reading is taken. Since we would like to have a new cycle every 10 seconds, the BCD up-counter of Figure 4.35 (b) would be appropriate. If we drive the counter with a clock having a frequency of one cycle per second, then the counter increments one count per second and resets after ten seconds. The clock signal can be generated via a 555 Timer IC.

The outputs from the counter ($Q_3Q_2Q_1Q_0$) become additional inputs to the digital controller previously designed. The new circuit is shown in Figure 4.36. When all counter outputs are zero, then the digital controller initialises the system by resetting both flip-flops (this is achieved by setting outputs r_f and r_h high) and setting "enable" outputs e_f and e_h to zero.

When the counter has a binary output of 1, then the control calculation takes place as before. However, instead of turning on the fan or heater directly, the outputs f and h are stored in two S-R flip-flops by connecting to the set pin of each corresponding flip-flop. The controller outputs e_f and e_h are ANDed with the flip-flop outputs to prevent a high signal reaching the fan or heater until the appropriate time.

When the counter has binary outputs equivalent to 2,3 or 4 the outputs of the appropriate flip-flops are gated to the fan or heater. When the counter has binary outputs equivalent to 5,6,7,8 or 9, the reset pins on the relevant flip-flops are enabled. The combinational logic required to perform this task can be designed using previously discussed techniques.

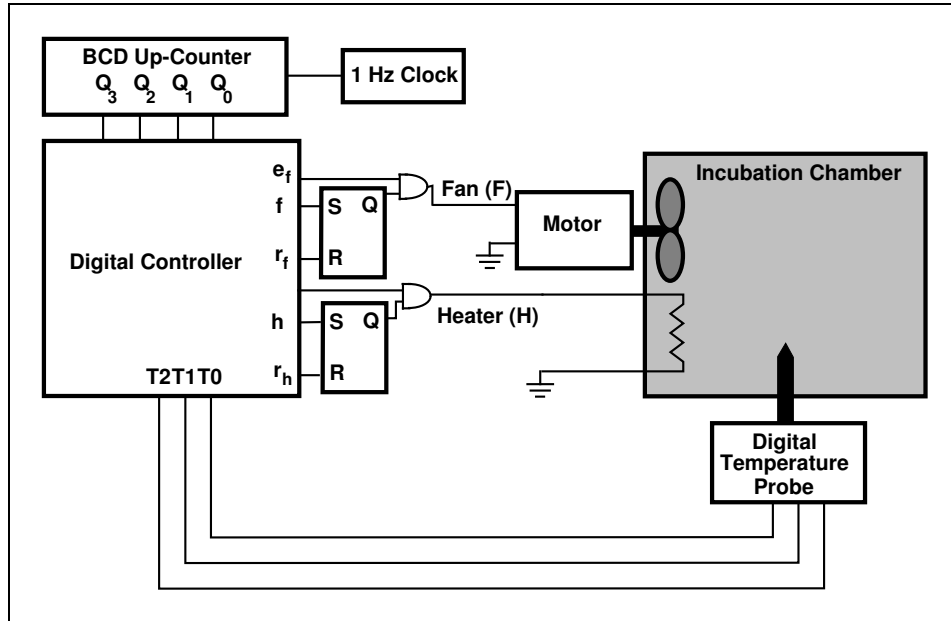


Figure 4.36 - Modified Incubator Control System

As an exercise, design the complete circuit for the incubator controller, beginning with the previously derived outputs for F and H.