Contents

List of Figures

List of Tables

List of Programs

CHAPTER 1

Digital representation

To a computer or microprocessor, the world is seen in terms of patterns of digits. The **decimal** (or denary) system represents quantities in terms of the ten digits 0...9. Together with the judicious use of the symbols $+$, $-$ and . any quantity in the range $\pm\infty$ can be depicted. Indeed non-numeric concepts can be encoded using numeric digits. For example the American Standard Code for Information Interchange (ASCII) defines the alphabetic (alpha) characters A as 65, B = 66...Z = 90 and a = 97, b = 98...z = 122 etc. Thus the string "Microprocessor" could be encoded as "77, 105, 99, 114, 111, 112, 114, 111, 99, 101, 115, 115, 111, 114". Provided you know the context, that is what is a pure quantity and what is text, then just about any symbol can be coded as numeric digits.[1]

Electronic circuits are not very good at storing and processing a multitude of different symbols. It is true that the first American digital computer, the ENIAC (Electronic Numerical Integrator And Calculator) in 1946 did its arithmetic in decimal[2] but all computers since handle data in **binary** (base 2) form. The decimal (base 10) system is really only convenient for humans, in that we have ten fingers.[3] Thus in this chapter we will look at the properties of binary digits, their groupings and processing. After reading it you will:

- *Understand why a binary data representation is the preferred base for digital circuitry.*

---

[1] Of course there are lots of encoding standards, for example the 6-dot Braille code for the visually impaired.
[2] As did Babbage's mechanical computer of a century earlier.
[3] And ten toes, but base-20 systems are rare.

- *Know how a quantity can be depicted in natural binary, hexadecimal and binary coded decimal.*

- *Be able to apply the rules of addition and subtraction for natural binary quantities.*

- *Know how to multiply by shifting left.*

- *Know how to divide by shifting right and propagating the sign bit.*

- *Understand the Boolean operations of NOT, AND, OR and EOR.*

The information technology revolution is based on the manipulation, computation and transmission of digitized information. This information is virtually universally represented as aggregrates of *bi*nary digi*ts* (**bits**).[4] Most of this processing is effected using microprocessors, and it is sobering to reflect that there is more computing power in a singing birthday card than existed on the entire planet in 1950!

Binary is the universal choice for data representation, as an electronic switch is just about the easiest device that can be implemented using a transistor. Such 2-state switches are very small; they change state very quickly and consume little power. Furthermore, as there are only two states to distinguish between, a binary depiction is likely to be resistant to the effects of noise. The upshot of this is that both the packing density on a silicon chip and switching rate can be very high. Although a switch on its own does not represent much computing power; five million switches changing at 100 million times a second, manage to present at least a facade of intelligence!

The two states of a bit are conventionally designated **logic 0** and **logic 1** or just 0 & 1. A bit may be represented by two states of any number of physical quantities; for example electric current or voltage, light, pneumatic pressure. Most microprocessors use 0 V (or ground) for state 0 and 3 – 5 V for state 1, but this is not universal. For instance, the RS232 serial

---

[4]The binary base is not a new fangled idea invented for digital computer; many cultures have used base 2 numeration in the past. The Harappān civilisation existed more than 4000 years ago in the Indus river basin. Found in the ruins of the Harappān city of Mohenjo-Daro, in the beadmakers' quarter, was a set of stone pebble weights. These were in ratios that doubled in the pattern, 1,1,2,4,8,16…, with the base weight of around 25g ($\approx$ 1oz). Thus bead weights were expressed by digits which represented powers of 2; that is in binary.

port on your computer uses nominally $+12\,$V for state 0 and $-12\,$V for state 1.

A single bit on its own can only represent two states. By dealing with groups of bits, rather more complex entities can be coded. For example the standard alphanumeric characters can be coded using 7-bit groups of digits, as shown in Table. 1.1. Thus the ASCII code for "Microprocessor" becomes:

1001101 1101001 1100011 1110010 1101111 1110000 1110010 1101111
1100011 1100100 1110011 1110011 1101111 1110010

Unicode is an extension of ASCII and with its 16-bit code groups is able represent characters from many languages and mathematical symbols.

The ASCII code is **unweighted**, as the individual bits do not signify a particular quantity; only the overall pattern has any significance. Other examples are the die code of Fig. **??** on page **??** and the 7-segment code of Fig. 8.7 on page 186. Here we will deal with **natural binary** weighted codes, where the position of a bit within the number field determines its value or weight. In an integer binary number the rightmost digit is worth $2^0 = 1$, the next left column $2^1 = 2$ and so on to the nth column which is worth 2^{n-1}. For example the decimal number one thousand nine hundred and ninety eight is represented as $1 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 8 \times 10^0$ or 1998. In **natural binary** the same quantity is $1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, or $11111001101b$. Fractional numbers may equally well be represented by columns to the right of the binary point using negative powers of 2. Thus $1101.11b$ is equivalent to 14.75. As can be seen from this example, binary numbers are rather longer than their decimal equivalent; on average a little over three times. Nevertheless, 2-way switches are considerably simpler than 10-way devices, so the binary representation is preferable.

An n-digit binary number can represent up to 2^n patterns. Most computers store and process groups of bits. For example the first microprocessor, the Intel 4004, handled its data four bits (a **nybble**) at a time. Many current processors cope with blocks of 8 bits (a **byte**), 16 bits (a **word**), or 32 bits (a **long-word**). 64-bit (a **quad-word**) devices are on the horizon. These groupings are shown in Table 1.2. The names illustrated are somewhat de-facto, and variations are sometimes encountered.

MS nybble — LS nybble	0h 000b	1h 001b	2h 010b	3h 011b	4h 100b	5h 101b	6h 110b	7h 111b
0h 0000b	NUL	DLE	SP	0	@	P	'	p
1h 0001b	SOH	DC1	!	1	A	Q	a	q
2h 0010b	STX	DC2	"	2	B	R	b	r
3h 0011b	ETX	DC3	#	3	C	S	c	s
4h 0100b	EOT	DC4	$	4	D	T	d	t
5h 0101b	ENQ	NAK	%	5	E	U	e	u
6h 0110b	ACK	SYN	&	6	F	V	f	v
7h 0111b	BEL	ETB	'	7	G	W	g	w
8h 1000b	BS	CAN	(	8	H	X	h	x
9h 1001b	HT	EM	)	9	I	Y	i	y
Ah 1010b	LF	SUB	*	:	J	Z	j	z
Bh 1011b	VT	ESC	+	;	K	[	k	{
Ch 1100b	FF	FS	,	<	L	\	l	|
Dh 1101b	CR	GS	-	=	M	}	m	}
Eh 1110b	SO	RS	.	>	N	^	n	~
Fh 1111b	SI	US	/	?	O	_	o	DEL

7-bit American Standard Code for Information Interchange (ASCII)

Table 1.1 *Table of 7-bit ASCII characters.*

As in the decimal number system, large binary numbers are often expressed using the prefixes k (kilo), M (mega) and G (giga). A binary kilo is $2^{10} = 1024$; for example 64 kbyte of memory. In an analogous way, a binary mega is $2^{20} = 1,048,576$; thus a 1.44 Mbyte floppy disk. Similarly a 2 Gbyte hard disk has a storage capacity of $2 \times 2^{30} = 2,147,483,648$ bytes. The former representation is certainly preferable.

Bit (0–1)	(1 bit)	0–1
Nybble (0000–1111)	(4 bits)	0–15
Byte (0000 0000–11111 1111)	(8 bits)	0–255
Word (0000 0000 0000 0000–1111 1111 1111 1111)	(16 bits)	0–65,535
Long-word (0000 0000 0000 0000 0000 0000 0000 0000–1111 1111 1111 1111 1111 1111 1111 1111)	(32 bits)	0–4,294,967,295

Table 1.2 *Some common bit groupings.*

Long binary numbers are not very human friendly. In Table 1.2, binary numbers were zoned into fields of four digits to improve readability. Thus the address of a data unit stored in memory might be 1000 1100 0001 0100 0000 1010b. If each group of four can be given its own symbol, 0...9 and A...F, as shown in Table 1.3, then the address becomes 8C140Ah; a rather more manageable characterization. This code is called **hexadecimal**, as there are 16 symbols. Hexadecimal (base-16) numbers are a viable number base in their own right, rather than just being a convenient binary representation. Each column is worth $16^0, 16^1, 16^2 \ldots 16^n$ in the normal way.[5]

Binary Coded Decimal is a hybrid binary/decimal code extensively used at the input/output ports of a digital system (see Chapter **??**). Here each decimal digit is individually replaced by its 4-bit binary equivalent. Thus 1998 is coded as $(0001\ 1001\ 1001\ 1000)_{BCD}$. This is very different from the equivalent natural binary code; even if it is represented by 0s and

---

[5]Many scientific calculators, including that in the Accessories group under Windows 95, can do hexadecimal arithmetic.

Decimal	Natural binary	Hexadecimal	Binary
00	00000	00	0000 0000
01	00001	01	0000 0001
02	00010	02	0000 0010
03	00011	03	0000 0011
04	00100	04	0000 0100
05	00101	05	0000 0101
06	00110	06	0000 0110
07	00111	07	0000 0111
08	01000	08	0000 1000
09	01001	09	0000 1001
10	01010	0A	0001 0000
11	01011	0B	0001 0001
12	01100	0C	0001 0010
13	01101	0D	0001 0011
14	01110	0E	0001 0100
15	01111	0F	0001 0101
16	10000	10	0001 0110
17	10001	11	0001 0111
18	10010	12	0001 1000
19	10011	13	0001 1001
20	10100	14	0010 0000

Table 1.3 *Different ways of representing the quantities decimal 0…20.*

1s. As might be expected, arithmetic in such a hybrid system is difficult, and BCD is normally converted to natural binary at the system input and processing is done in natural binary before being converted back (see Program **??** on page **??**).

The rules of arithmetic are the same in natural binary[6] as they are in the more familiar base 10 system, indeed any base-n radix scheme. The simplest of these is **addition**, which is a shorthand way of totalling quantities, as compared to the more primitive counting or incrementation process. Thus $2 + 4 = 6$ is rather more efficient than $2 + 1 = 3; 3 + 1 = 4; 4 + 1 = 5; 5 + 1 = 6$. However, it does involve memorizing the rules

---

[6]Sometimes called 8-4-2-1 code after the weightings of the first four lowest columns.

of addition.[7] In decimal this involves 45 rules, assuming that order is irrelevant; from $0 + 0 = 0$ to $9 + 9 = 18$. Binary addition is much simpler as it is covered by only three rules:

$$0 + 0 \quad = \quad 0$$
$$\left.\begin{array}{l} 0 + 1 \\ 1 + 0 \end{array}\right\} = \ 1$$
$$1 + 1 \quad = 10 \quad (0 \text{ carry } 1)$$

Based on these rules, the least significant bit (LSB) is totalized first, passing a **carry** if necessary to the next left column. The process ends with the most significant bit (MSB) column, its carry being the new MSD of the sum. For example:

```
 1                        1
 0 1                      2 6 3  1
 0 0 1                    8 4 2 6 8 4 2 1
   96  Augend               1100000  Augend
 + 37  Addend             + 0100101  Addend
  1 1   Carries            1 1        Carries
  133  Sum                 10000101  Sum
```

(a) Decimal   *(b) Binary*

Just as addition implements an up count, **subtraction** corresponds to a down count, where units are removed from the total. Thus $8 - 5 = 3$ is the equivalent of $8 - 1 = 7; 7 - 1 = 6; 6 - 1 = 5; 5 - 1 = 4; 4 - 1 = 3$.

The technique of decimal subtraction you are familiar with applies the subtraction rules commencing from LSB and working to the MSB. In any given column were a larger quantity is to be taken away from a smaller quantity, a unit digit is **borrowed** from the next higher column and given back after the subtraction is completed. Based on this borrow principle, the subtraction rules are given by:

$$0 - 0 = 0$$
$$^{1}0 - 1 = 1 \quad \text{Borrowing 1 from the higher column}$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$

For example:

---

[7]Which you had to do way back in the mists of time in primary/elementary school!

```
 1                         6 3 1
 0 1                       4 2 6 8 4 2 1
 96  Minuend              1100000  Minuend
-37  Subrahend          - 0100101  Subrahend
   1   Borrows            1 1 1 1 1 1   Borrows
 59  Difference           0111011   Difference
```

(a) Decimal   *(b) Binary*

Although this familiar method works well, there are several problems implementing it in digital circuitry.

- How can we deal with situations where the minuend is larger than the subtrahend?
- How can we distinguish between positive and negative quantities?
- Can a digital system's adder circuits be coerced into subtracting?

To illustrate these points, consider the following example:

```
 37  Minuend             0100111  Minuend
-96  Subrahend         - 1100000  Subrahend
  1                      1
 41  Difference (-59)    1000111   Difference (-0111001)
```

(a) Decimal   *(b) Binary*

Normally when we know that the when Minuend is greater than the Subtrahend, the two operands are interchanged and a minus sign is appended to the outcome; that is $-$(Subtrahend $-$ Minuend). If we do not swap, as in (a) above, then the outcome appears to be incorrect. In fact 41 is correct, in that this is the difference between 59 (the correct outcome) and 100. 41 is described as the **10's complement** of 59. Furthermore, the fact that a borrow digit was generated from the MSD indicates that the difference is negative, and therefore appears in this 10's complement form. Converting from 10's complement decimal numbers to the 'normal' magnitude form is simply a matter of inverting each digit and then adding one to the outcome. A decimal digit is inverted by computing its difference from 9. Thus the 10's complement of 3941 is -6059:

$$\overline{3941} \implies 6058; +1 = -6059$$

However, there is no reason why negative numbers should not remain in this complement form — just because we are not familiar with this type of notation.

The complement method of negative quantity representation of course applies to binary numbers. Here the ease of inversion $(0 \to 1; 1 \to 0)$ makes this technique particularly attractive. Thus in our example above:

$$\overline{1000111} \Longrightarrow 0111000; +1 = -0111001$$

Again, negative numbers should remain in a **2's complement** form. This complement process is reversible. Thus:

$$\text{complement} \Longleftrightarrow \text{normal}$$

Signed decimal numeration has the luxury of using the symbols $+$ and $-$ to denote positive and negative quantities. A 2-state system is stuck with 1s and 0s. However, looking at the last example gives us a clue on how to proceed. A negative outcome gives a borrow back out to the highest column. Thus we can use this MSD as a **sign bit**, with 0 for $+$ and 1 for $-$. This gives $1,1000111b$ for -59 and $0,01110011b$ for $+59$. Although for clarity the sign bit has been highlighted above using a comma delimiter, the advantage of this system is that it can be treated in all arithmetic processes in the same way as any other ordinary bit. Doing this, the outcome will give the correct sign:

0,1100000	(+96)	0,0100101	(+37)
1,1011011	(−37)	1,0100000	(−96)
1		1	
0,0111011	(+59)	1,1000101	(−59)

(a) Minuend less than subtrahend   *(b) Minuend greater than subtrahend*

From this example we see that if negative numbers are in a signed 2's complement form, then we no longer have the requirement to implement hardware subtractors, as adding a negative number is equivalent to subtracting a positive number. Thus $A - B = A + (-B)$. Furthermore, once numbers are in this form, the outcome of any subsequent processing will always remain 2's complement signed throughout.

There are two difficulties associated with signed 2's complement arithmetic. The first of these is **overflow**. It is possible that adding two positive or two negative numbers will cause overflow into the sign bit; for instance:

0,1000 (+8)   1,1000 (−8)
0,1011 (+11)   1,0101 (−11)

1,0011 (−13!!!)   0,1101 (+3!!!)

(a) Sum of two +ve numbers gives −ve   *(b) Sum of two −ve numbers gives +ve*

In (a) the outcome of $(+8) + (+11)$ is -13! The 2^4 numerical digit has overflowed into the sign position (actually, $10011b = 19$ is the correct outcome). Example (b) shows a similar problem for the addition of two signed negative numbers. Overflow can only happen if both operands have the *same* sign bits. Detection is then a matter of determining this situation with an outcome that differs. See Fig. 1.5 for a logic circuit to implement this overflow condition.

The final problem concerns arithmetic on signed operands with different sized fields. For instance:

0,0011001 (+25)   0,0011001 (+25)
     0,011 (+03)   1,101 (−03)

????   ????

0,0011001 (+25)   0,0011001 (+25)
0,0000011 (+03)   1,1111101 (−03)

0,0011100 (+28)   0,0010110 (+22)

(a) Extending a positive number   *(b) Extending a negative number*

Both the examples involve adding an 8-bit to a 16-bit operand. Where the former is positive, the data may be increased to 16 bits by padding with 0s. The situation is slightly less intuitive where negative data requires extension. Here the prescription is to extend the data by padding out with 1s. In the general case the rule is simply to pad out data by propagating the sign bit left. This technique is known as **sign extension**.

Multiplication by the nth power of two is simply implemented by shifting the data left n places. Thus $00101(5) << 01010(10) << 10100(20)$ multiplies 5 by 2^2, where the << operator is used to denote shifting left. The process works for signed numbers as well:

```
0,00000011  ( 3)      1,11111101  (−3)      0,00000110  (3 x 2)
    <<                    <<              +  0,00011000  (3 x 8)
0,00000110  ( 6)      1,11111010  (−6)        ─────────────────
    <<                    <<                 0,00011110  (3 x 10 = 30)
0,00001100  (12)      1,11110100  (−12)
    <<                    <<
0,00011000  (24)      1,11101000  (−24)
```

(a) +3 × 8 = +24   *(b) −3 × 8 = −24*   *(c) +3 × 10 = 30*

Should the sign bit change polarity, then a magnitude bit has overflowed. Some computers/microprocessors have a Arithmetic Shift Left process that signals this situation, as opposed to the standard Logic Shift Left used in unsigned number shifts.

Multiplication by non-powers of 2 can be implemented by a combination of shifting and adding. Thus as shown in (c) above, 3×10 is implemented as $(3 \times 8) + (3 \times 2) = (3 \times 10)$ or $(3 << 3) + (3 << 1)$ (see also Example ??.2.

In a similar fashion, division by powers of 2 is implemented by shifting right n places. Thus $1100(12) >> 0110(6) >> 0011(3) >> 0001.1(1.5)$. This process also works for signed numbers:

```
0,1111.000  (+15)     1,0001.000  (−15)          0001.1
    >>                    >>               1010 │ 1111.0
0,0111.100  (+7.5)    1,1000.100  (−7.5)         −1010
    >>                    >>                     ─────
0,0011.110  (+3.75)   1,1100.010  (−3.75)         0101
    >>                    >>                      −101.0
0,0001.111  (+1.875)  1,11110.001 (−1.875)       ──────
                                                  000.0
```

(a) +15/8 = 1.875   *(b) −15/8 = −1.875*   *(c) 15/10 = 1.5*

Notice that rather than always shifting in 0s, the sign bit should be propagated in from the left. Thus positive numbers shift in 0s and negative

numbers shift in 1s. This is known as **Arithmetic Shift Right** as opposed to **Logic Shift Right** which always shifts in 0s.

Division by non powers of 2 is illustrated in (c) above. This shows the familiar long division process used in decimal division. This is an analagous process to the shift and add technique for multiplication, using a combination of shifting and subtracting.

Arithmetic is not the only way to manipulate binary patterns. George Boole[8] in the mid-19th century developed an algebra dealing with symbolic processing of logic propositions. This **Boolean algebra** deals with variables which can be true or false. In the 1930s it was realised that this mathematical system could equally well be used to analyze switching networks and thus binary logic systems. Here we will confine ourselves to looking at the fundamental logic operations of this switching algebra.

A	f
0	1
1	0

$A \rightarrow f = \overline{A}$

$A \rightarrow \boxed{1} \rightarrow f = \overline{A}$

(a) Truth table   *(b) Alternative logic symbols*
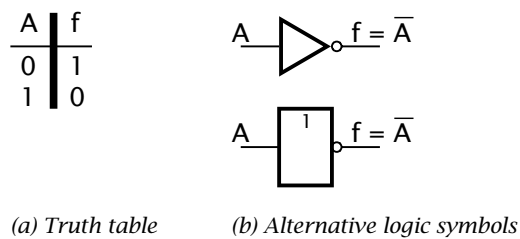
Figure 1.1 *The NOT operation.*

The inversion or **NOT** operation is represented by overscoring. Thus $f = \overline{A}$ states that the variable f is the inverse of A; that is if $A = 0$ then $f = 1$ and if $A = 1$ then $f = 0$. In Fig. 1.1(a) this transfer characteristic is presented in the form of a **truth table**. By definition, inverting twice returns a variable to its original state; thus $\overline{\overline{f}} = f$.[9]

---

[8]The first professor of mathematics at Queen's College, Cork.

[9]In days of yore when logic circuits were built out of discrete devices, such as diodes, resistors and transistors, problems due to sneak current paths were rife. In one such laboratory experiment the output lamp was rather dim, and the lecturer in charge suggested that two NOTs in series in a suspect line would not disturb the logic but would block off the unwanted current leak. On returning sometime later, the students complained that the remedy had had no effect. On investigation the lecturer discovered two knots in the offending wire — obviously not tied tightly enough!

Logic function implementations are normally represented in an abstract manner rather than as a detailed circuit diagram. The **NOT gate** is symbolized as shown in Fig. 1.1(b). The circle *always* represents inversion in a logic diagram, and is often used in conjunction with other logic elements, such as in Fig. 1.2(c).

B A	f
0 0	0
0 1	0
1 0	0
1 1	1

B A	f
0 0	1
0 1	1
1 0	1
1 1	0

$f = B \cdot A$

$f = B \cdot A$

$f = \overline{B \cdot A}$

$f = \overline{B \cdot A}$

(a) Truth table   *(b) Alternative logic symbols*   *(c) NAND*
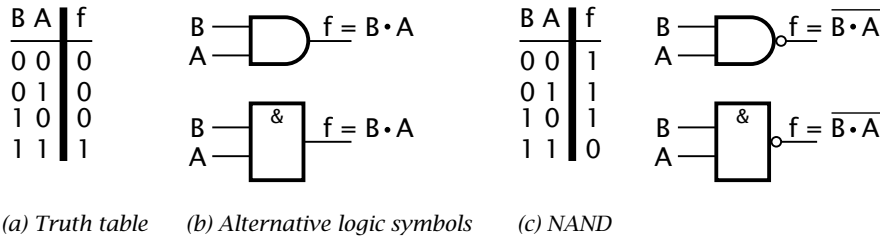
Figure 1.2 *The AND function.*

The **AND operator** gives an *all or nothing* function. The outcome will only be true when *every* one of the n inputs are true. In Fig. 1.2 two input variables are shown, and the output is symbolized as $f = B \cdot A$, where $\cdot$ is the Boolean AND operator. The number of inputs is not limited to two, and in general $f = A(0) \cdot A(1) \cdot A(2) \cdots A(n)$. The AND operator is sometimes called a logic product, as ANDing (cf. multiplying) any bit with logic 0 always yields a 0 output.

If we consider B as a control input and A as a stream of data, then consideration of the truth table shows that the output follows the data stream when $B = 1$ and is always 0 when $B = 0$. Thus the circuit can be considered to be acting as a valve, gating the data through on command. The term **gate** is generally applied to any logic circuit implementing a fundamental Boolean operator.

Most practical AND gate implementations have an inverting output. The logic of such implementations is NOT AND, or NAND for short, and is symbolized as shown in Fig. 1.2(c).

The **OR operator** gives an *anything* function. Here the outcome is true when *any* input or inputs are true (hence the ≥ 1 label in the logic symbol). In Fig. 1.3 two inputs are shown, but any number of variables may be ORed together. ORing is sometimes referred to as a logic sum, and the + used as the mathematical operator; thus $f = B + A$. In an analogous

B A	f
0 0	0
0 1	1
1 0	1
1 1	1

$f = B + A$

$f = B + A$

B A	f
0 0	1
0 1	0
1 0	0
1 1	0

$f = \overline{B + A}$

$f = \overline{B + A}$

(a) Truth table   *(b) Alternative logic symbols*   *(c) NOR*
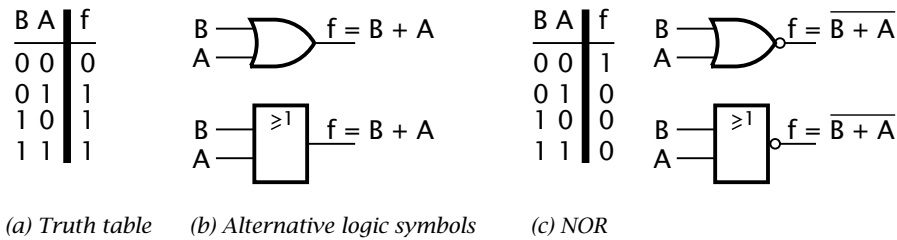
Figure 1.3 *The OR operation.*

manner to the AND gate detecting all ones, the OR gate can be used to detect all zeroes. This is illustrated in Fig. 2.16 on page 34 where an 8-bit zero outcome brings the output of the NOR gate to 1.

Considering B as a control input and A as data (or vice versa), then from Fig. 1.3(a) we see that the data is gated through when B is 0 and inhibited (always 1) when B is 1. This is a little like the inverse of the AND function. In fact the OR function can be expressed in terms of AND using the duality relationship $\overline{A + B} = \overline{B} \cdot \overline{A}$. This states that the NOR function can be implemented by inverting all inputs into an AND gate.

AND, OR and NOT are the three fundamental Boolean operators. There is one more operation commonly available as an electronic gate; the **Exclusive-OR operator** (**EOR**). The EOR function is true if *only one* input is true (hence the =1 label in the logic symbol). Unlike the inclusive-OR, the situation where both inputs are true gives a false outcome.

B A	f
0 0	0
0 1	1
1 0	1
1 1	0

$f = B \oplus A$

$f = B \oplus A$

B A	f
0 0	1
0 1	0
1 0	0
1 1	1

$f = \overline{B \oplus A}$

$f = \overline{B \oplus A}$

(a) Truth table   *(b) Alternative logic symbols*   *(c) ENOR*
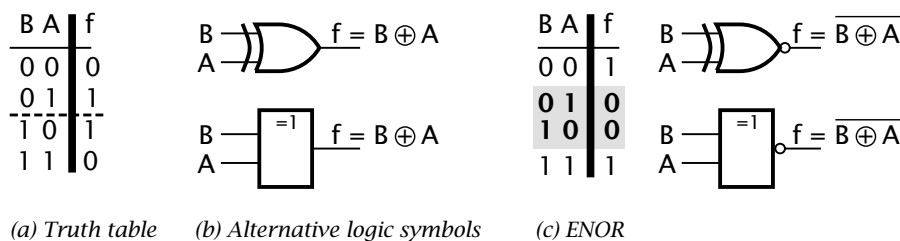
Figure 1.4 *The EOR operation.*

If we consider B is a control input and A as data (they are fully inter-

changeable) then:

- When $B = 0$ then $f = A$; that is the output follows the data input.

- When $B = 1$ then $f = \overline{A}$; that is the output is the inverse of the data input.

Thus an EOR gate can be used as a programmable inverter.

Another useful property considers the EOR function as a logic differentiator. The EOR truth table shows that the gate gives a true output if the two inputs differ. Alternatively, the ENOR truth table of Fig. 1.4(c) shows a true output when the two inputs are the same. Thus an ENOR gate can be considered to be a 1-bit equality detector. The equality of two n-bit words can be tested by ANDing an array of ENOR gates (see Fig. 2.6 on page 23), each generating the function $\overline{B_k \oplus A_k}$; that is:

$$f_{B=A} = \sum_{k=0}^{n-1} \overline{B_k \oplus A_k}$$

As a simple example of the use of the EOR/ENOR gates, consider the problem of detecting sign overflow (see page 10). This occurs if both the sign bits of word B and word A are the same $(\overline{S_B \oplus S_A})$ AND the sign bit of the outcome word C is not the same as either of these sign bits, say $S_B \oplus S_C$. The logic diagram for this detector is shown in Fig. 1.5 and implements the Boolean function:

$$(\overline{S_B \oplus S_A}) \cdot (S_B \oplus S_C)$$
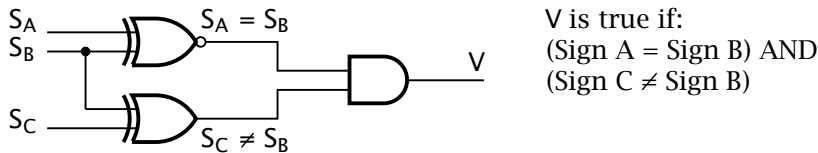
Figure 1.5 *Detecting sign overflow.*

Finally, the EOR function can be considered as detecting when the number of true inputs are odd. By cascading $n + 1$ EOR gates, the overall parity

function is true if the n-bit word has an odd number of ones. Some measure of error protection can be obtained by adding an additional bit to each word, so that overall the number of bits is odd. This oddness can be checked at the receiver and any deviation indicates corruption (see page ??).

CHAPTER 2

Logic circuitry

We have noted that digital processing is all about transmission, manipulation and storage of binary word patterns. Here we will extend the concepts introduced in the last chapter as a lead into the architecture of the computer and microprocessor. We will look at some relevant logic functions, their commercial implementations and some practical considerations.

After reading this chapter you will:

- *Understand the properties and use of active pull-up, open-collector and 3-state output structures.*
- *Appreciate the logic structure and function of the natural decoder.*
- *See how a MSI implementation of an array of ENOR gates can compare two words for equality.*
- *Understand how a 1-bit adder can be constructed from gates, and can be extended to deal with the addition of two n-bit words.*
- *Appreciate how the function of an ALU is so important to a programmable system.*
- *Be aware of the structure and utility of a read-only memory (ROM).*
- *Understand how two cross-coupled gates can implement a R S latch.*
- *Appreciate the difference between a D latch and D flip flop.*
- *Understand how an array of D flip flops or latches can implement a register.*
- *See how a serial connection of D flip flops can perform a shifting function.*
- *Understand how a D flip flop can act as a frequency divide by two, and how a cascade of these can implement a binary count.*

17

- *See how an ALU/PIPO register can implement an accumulator processor unit.*

- *Appreciate the function of a RAM.*

The first integrated circuits, available at the end of the 1960s, were mainly NAND, NOR and NOT gates. The most popular family of logic functions was, and still is, the 74 series transistor transistor logic (TTL); introduced by Texas Instruments and soon copied by all the major major semiconductor manufacturers.

| (a) DIL package | (b) ANSI/IEC logic symbol |
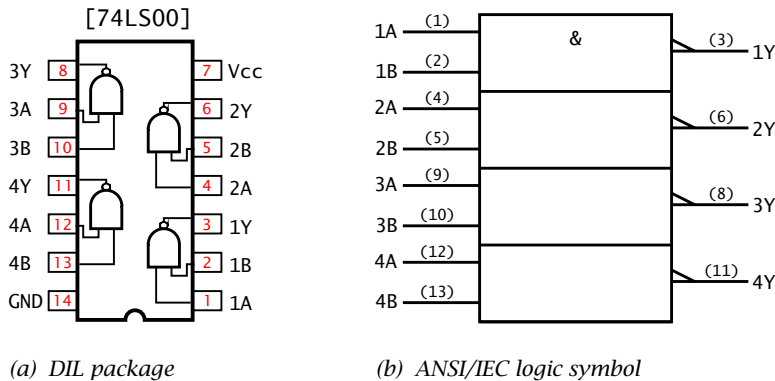
Figure 2.1 *The 74LS00 quad 2-I/P NAND package.*

The 74LS00[1] comprises four 2-input NAND gates in a 14-pin package. The integrated circuit (IC) is powered with a 5 ± 0.25 V supply between V_{CC}[2] (usually about 5 V) and GND. The logic outputs are 2.4 – 5 V High and 0 – 0.4 V for Low. The logic outputs are 2.4 – 5 V High and 0 – 0.4 V for Low.

---

[1] The LS stands for Low-power Schottky transistor. There are very many other versions, such as ALS (Advanced LS), AS (Advanced Schottky) and HC (High-speed Complementary metal-oxide transistor — CMOS). These family variants differ in speed and power consumption, but for a given number designation have the same logic function and pinout.

[2] For historical reasons the positive supply on logic ICs are usually designated as V_{CC}; the C referring to a bipolar's transistor Collector supply. Similarly field-effect circuitry sometimes use the designation V_{DD} for Drain voltage. The zero reference pin is normally designated as the ground point (GND), but sometimes the V_{EE} (for emitter) or V_{SS} (for Drain) label is employed.

Most IC logic families require a 5 V supply, but 3 V versions are becoming available, and some CMOS implementations can operate with a range of supplies between 3 V and 15 V.

The 74LS00 IC is shown in Fig. 2.1(a) in its Dual In-Line (DIL) package. Strictly it should be described as a positive-logic quad 2-I/P NAND, as the electrical equivalent for the two logic levels 0 and 1 are Low (L is around ground potential) and High (H is around Vcc, usually about 5 V). If the relationship $0 \rightarrow H$; $1 \rightarrow L$ is used (negative logic) then the 74LS00 is actually a quad 2-I/P NOR gate. The ANSI/IEC[3] logic symbol of Fig. 2.1(b) denotes a Low electrical potential by using the polarity ◣ symbol. The ANSI/IEC NAND symbol shown is thus based on the *real* electrical operation of the circuit. In this case the logic coincides with a positive-logic NAND function. The & operator shown in the top block is assumed applicable to the three lower gates.

The output structure of a 74LS00 NAND gate is **active pull-up**. Here both the High and Low states are generated by connection via a low-resistance switch to Vcc or GND respectively. In Fig. 2.2(a) these switches are shown for simplicity as metallic contacts, but they are of course transistor derived.

(a) Push/pull (Totem-pole)   (b) Open-collector (open-drain)   (c) Three-state
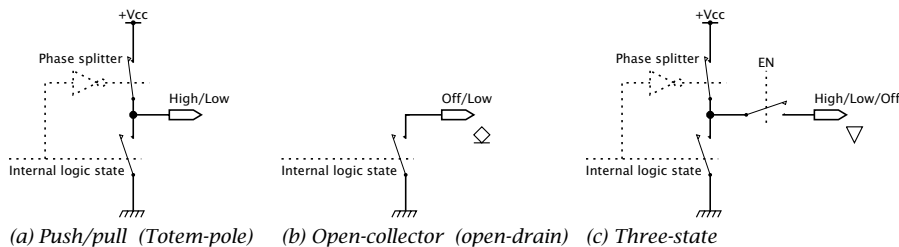
Figure 2.2 *Output structures.*

Logic circuits, such as the 74LS00, change output state in around 10 nanoseconds.[4] To be able to do this, the capacitance of any interconnecting conductors and other logic circuits' inputs must be rapidly discharged. Mainly for this reason, active pull-up (sometimes called totem-pole) outputs are used by most logic circuits. There are certain circumstances where alternative output structures have some advantages.

---

[3]American National Standards Institution/International Electrotechnical Commission.

[4]A nanosecond is 10^{-9} s, so 100,000,000 transitions each second is possible.

The **open-collector** (or open-drain) configuration of Fig. 2.2(b) provides a 'hard' Low state, but the High state is in fact an open-circuit. The High-state voltage can be generated by connecting an external resistor to either Vcc or indeed to a different power rail. Non-orthodox devices, such as relays, lamps or light-emitting diodes, can replace this pull-up resistor. The output transistor is often rated with a higher than usual current and/or voltage rating for such purposes.
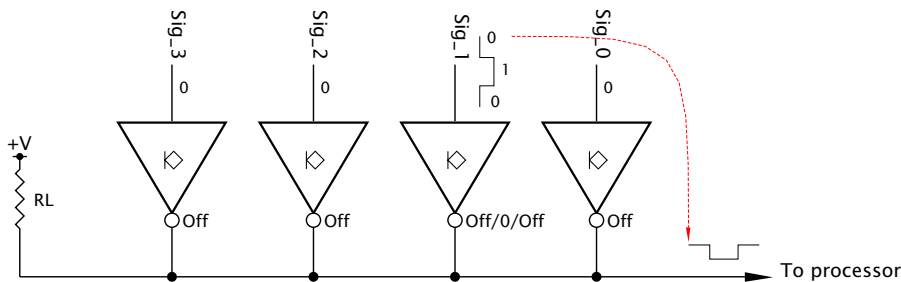
Figure 2.3 *Open-collector buffers driving a party line.*

The application of most interest to us here is illustrated in Fig. 2.3. Here four open-collector gates share a *single* pull-up resistor. Note the use of the ◇ symbol to denote an open-collector output. Assume that there are four peripheral devices, any of which may wish to attract the attention of the processor (eg. computer or microprocessor). If this processor has only one Attention pin, then the four Signal lines must be **wire-ORed** together as shown. With all Signals inactive (logic 0) the outputs of all buffer NOT gates are off (state H), and the party line is pulled up to +V by RL. If *any* Signal line is activated (logic 1), as in Sig_1, then the output of the corresponding buffer gate goes hard Low. This pulls the party line Low, irrespective of the state of the other signal lines, and thus interrupts the processor.

As an example of the use of this structure, consider the situation depicted in Fig. 2.4. Here a master controller wishes to read one of several devices, all connected to this master over a set of party lines. As this data highway or **Data bus** is a common resource, so only the selected device can be allowed access to the bus at any one time. The access has to be

From master controller
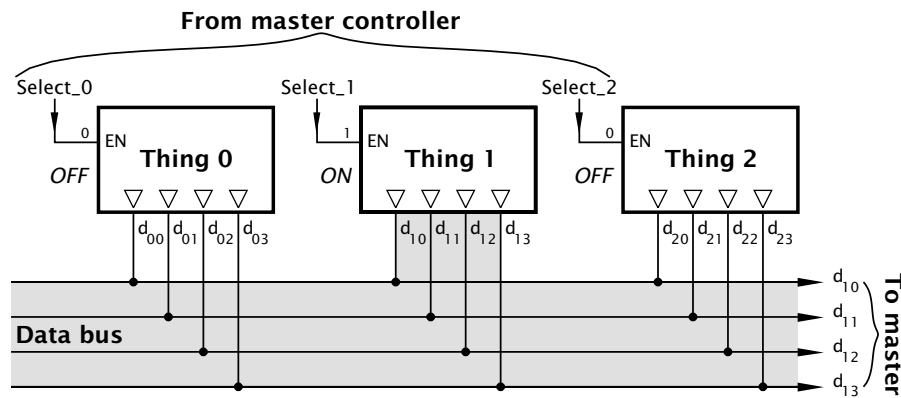
Figure 2.4 *Sharing a bus.*

withdrawn immediately the data has been read, so that another device can use the resource. As shown in the diagram, each Thing connected to the bus outputs, designated by the $\triangledown$ symbol. When selected, the active logic levels will drive the bus lines. The 74LS244 octal ($\times 8$) 3-state buffer shown in Fig. **??**(a) on page **??** has high-current outputs (designated by the $\triangleright$ symbol) specifically designed to charge/discharge the capacitance associated with long bus lines.

Integrated circuits with a complexity of up to 12 gates are categorised as Small-Scale Integration (SSI). Gate counts upwards to 100 on a single IC are Medium-Scale Integration (MSI), up to 1000 are known as Large-Scale Integration (LSI) and over this, Very Large-Scale Integration (VLSI). Memory chips and microprocessors are examples of this latter category.

The NAND gate networks shown in Fig. 2.5 are typical MSI-complexity ICs. Remembering that the output of a NAND gate is logic 0 only when *all* its inputs are logic 1 (see Fig. 1.2(c) on page 13) then we see that for any combination of the *Select* inputs B A ($2^1 \, 2^0$) in Fig. 2.5(a) only *one* gate will go to logic 0. Thus output $\overline{Y_2}$ will be activated when B A = 10. The associated truth table shows the circuit *decodes* the binary address B A so that address n selects output $\overline{Y_n}$. The 74LS139 is described as a dual 2 to 4-line **natural decoder**. Dual because there are two such circuits in the one chip. The symbol X/Y denotes converting code X (natural binary) to
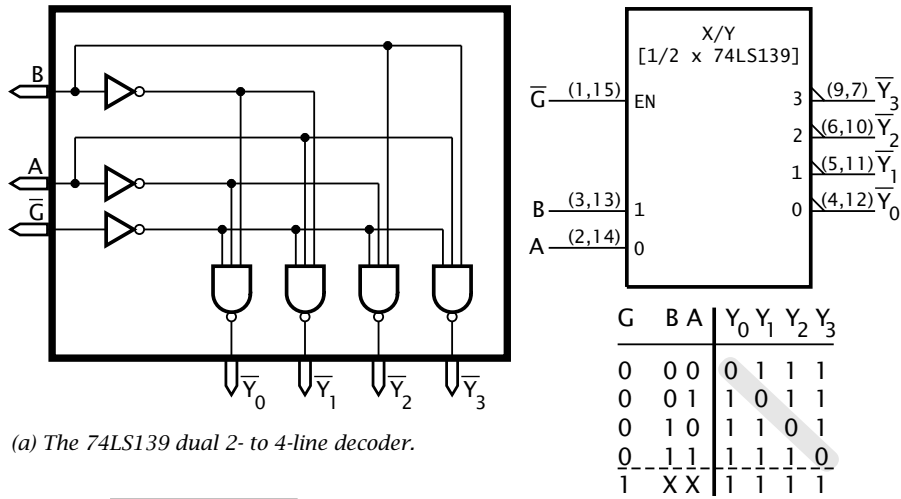
(a) The 74LS139 dual 2- to 4-line decoder.

G	B A	Y_0 Y_1 Y_2 Y_3
0	0 0	0 1 1 1
0	0 1	1 0 1 1
0	1 0	1 1 0 1
0	1 1	1 1 1 0
1	X X	1 1 1 1
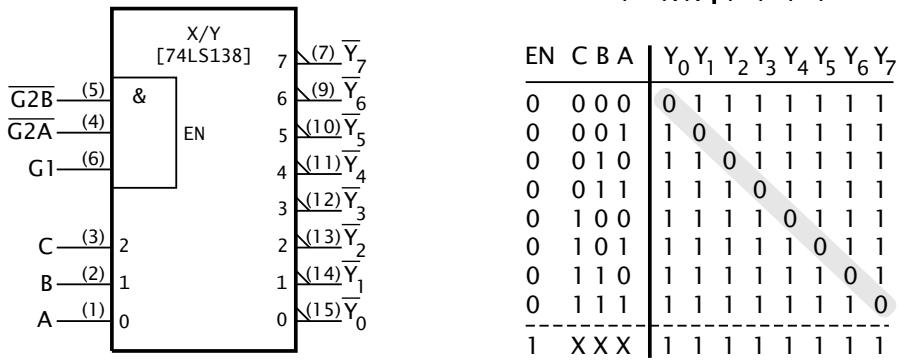
(b) The 74LS138 3- to 8-line decoder

EN	C B A	Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7
0	0 0 0	0 1 1 1 1 1 1 1
0	0 0 1	1 0 1 1 1 1 1 1
0	0 1 0	1 1 0 1 1 1 1 1
0	0 1 1	1 1 1 0 1 1 1 1
0	1 0 0	1 1 1 1 0 1 1 1
0	1 0 1	1 1 1 1 1 0 1 1
0	1 1 0	1 1 1 1 1 1 0 1
0	1 1 1	1 1 1 1 1 1 1 0
1	X X X	1 1 1 1 1 1 1 1

Figure 2.5 *The 74LS138 and '139 MSI natural decoders.*

code Y (unary — one of n). The Enable input $\overline{G}$ is connected to all gates in parallel. Thus the decoder function only operates if $\overline{G}$ is Low (logic 0). If $\overline{G}$ is High, then irrespective of the state of B A (the X entries in the truth table denote a 'don't care' situation) all outputs remain deselected — logic 1. An example of the use of the 74LS139 is given in Fig. 2.20.

The 74LS138 of Fig. 2.5(b) is similar, but implements a 3 to 8-line decoder function. The state of the three address lines CBA ($2^2 \, 2^1 \, 2^0$) n selects one only of the eight outputs $\overline{Y_n}$. The 74LS138 has three Gate inputs which generate an internal Enable signal $\overline{G2B} \cdot \overline{G2A} \cdot G1$. Only if both $\overline{G2A}$ and $\overline{G2B}$ are Low and $G1$ is High will the device be enabled. The 74LS138 is used several times in Chapter ?? to decode microprocessor Address lines, for example Fig. ?? on page ??.
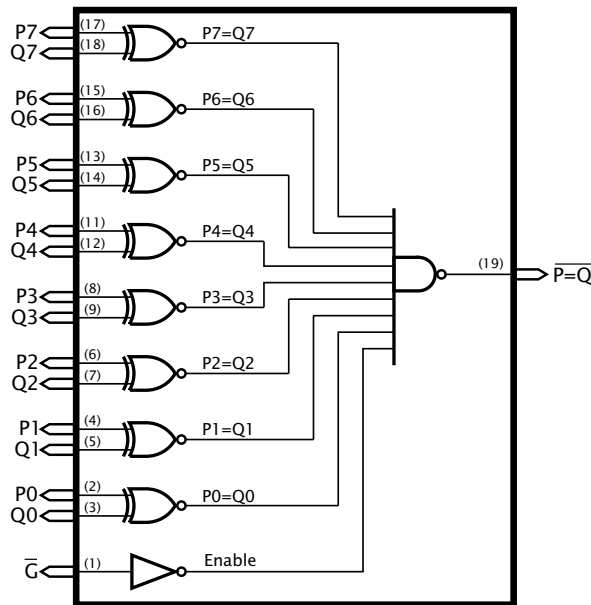
Figure 2.6 *The 74LS688 octal equality detector.*

A large class of ICs implement arithmetic operations. The gate array illustrated in Fig. 2.6 detects when the 8-bit byte P7...P0 is identical to the byte Q7...Q0. Eight ENOR gates each give a logic 1 when its two input bits Pn, Qn are identical, as described on page 15. Only if *all* eight bit pairs are the same, will the output NAND gate go Low. The 74LS688 **Equality comparator** also has a direct input $\overline{G}$ into this NAND gate, acting as an overall Enable signal.

The ANSI/IEC logic symbol, shown in Fig. **??** uses the COMP label to denote the arithmetic comparator function. The output is prefixed with the numeral 1, indicating that its operation P=Q is dependent on any input qualifying the same numeral; that is G1. Thus the active-Low Enable input G1 gates the active-Low output, 1P=Q.

One of the first functions beyond simple gates to be integrated into a single IC was that of addition. The truth table of Fig. 2.7(a) shows the Sum (S) and Carry-Out (C_1) resulting from the addition of the two bits A and B and any Carry-In (C_0). For instance row 6 states that adding two 1s with a Carry-In of 0 gives a Sum of 0 and a Carry-Out of 1 ($1 + 1 + 0 =^1 0$). To

(a) One-bit addition

(b) The 74LS283 four-bit adder
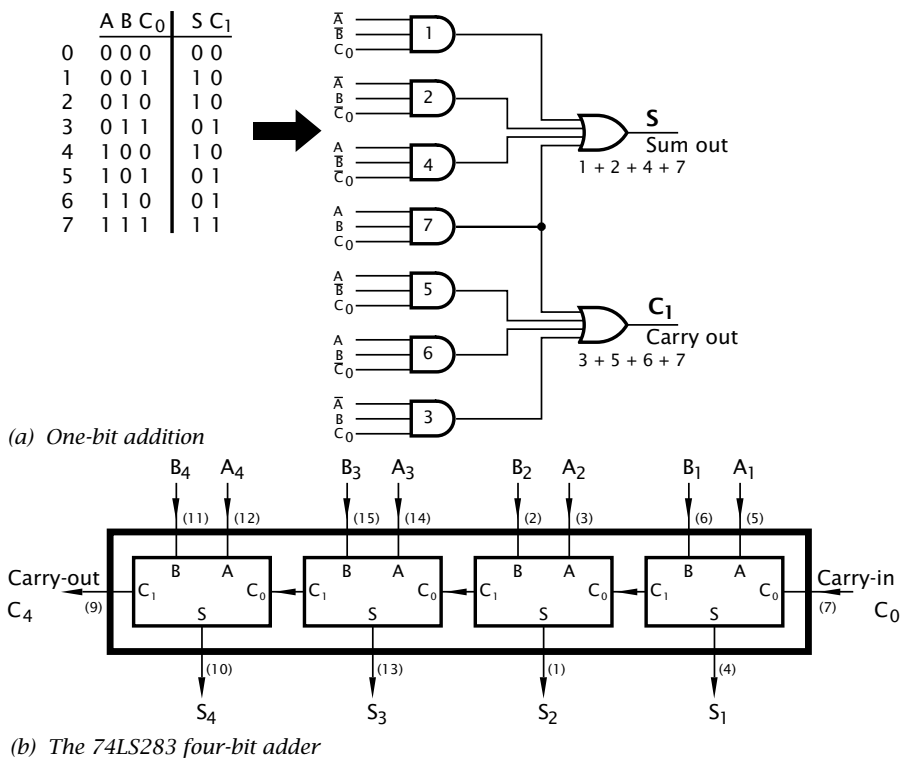
Figure 2.7 *Addition.*

implement this row we require to detect the pattern 1 1 0; that is $A \cdot B \cdot \overline{C_0}$; which is gate 6 in the logic diagram. Thus we have by ORing all applicable patterns together for each output:

$$
\begin{aligned}
S &= (\overline{A} \cdot \overline{B} \cdot C_0) + (\overline{A} \cdot B \cdot \overline{C_0}) + (A \cdot \overline{B} \cdot \overline{C_0}) + (A \cdot B \cdot C_0) \\
C_1 &= (\overline{A} \cdot B \cdot C_0) + (A \cdot \overline{B} \cdot C_0) + (A \cdot B \cdot \overline{C_0}) + (A \cdot B \cdot C_0)
\end{aligned}
$$

Using such a circuit for *each* column of a binary addition, with the Carry-Out from column $k-1$ feeding the Carry-In of column k means that the addition of any two n-bit words can be simultaneously implemented. As shown in Fig. 2.7(b), the 74LS283 adds two 4-bit nybbles in 25 ns. In practice the final Carry-Out C_4 is generated using additional circuitry to avoid the delays inherent on the carries rippling though each stage from the least to the most significant digit. n 74LS283s can be cascaded to implement addition for words of $4 \times n$ width. Thus two 74LS283s perform a 16-bit addition in 45 ns; the extra time being accounted for by the carry propagation between the two units.

Adders can of course be coaxed into subtraction by inverting the minuend and adding one, that is 2's complementation. An Adder/Subtractor circuit could be constructed by feeding the minuend word through an array of EOR gates acting as programmable inverters (see Fig. 1.4(a) on page 14). The Mode line $\overline{\text{Add}}/\text{Sub}$ in Fig. 2.8 that controls these inverters also feeds the Carry-In, effectively adding one when in the Subtract mode.

Extending this line of argument leads to the **Arithmetic Logic Unit** (**ALU**). An ALU is a circuit which can undertake a selection of arithmetic and logic processes on input data as controlled by Mode inputs. The 74LS382 in Fig. 2.9 processes two 4-bit operands in eight ways, as controlled by the three Select bits $S_2 S_1 S_0$ and tabulated in Fig. 2.9(a). Besides addition and subtraction, the logic operations of AND, OR and EOR are supported. The 74LS382 even generates the 2's complement overflow function (see page 10).

As we shall see, the ALU is the heart of the computer and microprocessor architectures. By feeding the Select inputs with a series of mode words, a program of operations can be performed by the ALU. Such operation codes are stored in an external memory, and are accessed sequentially by the computer's control circuits.
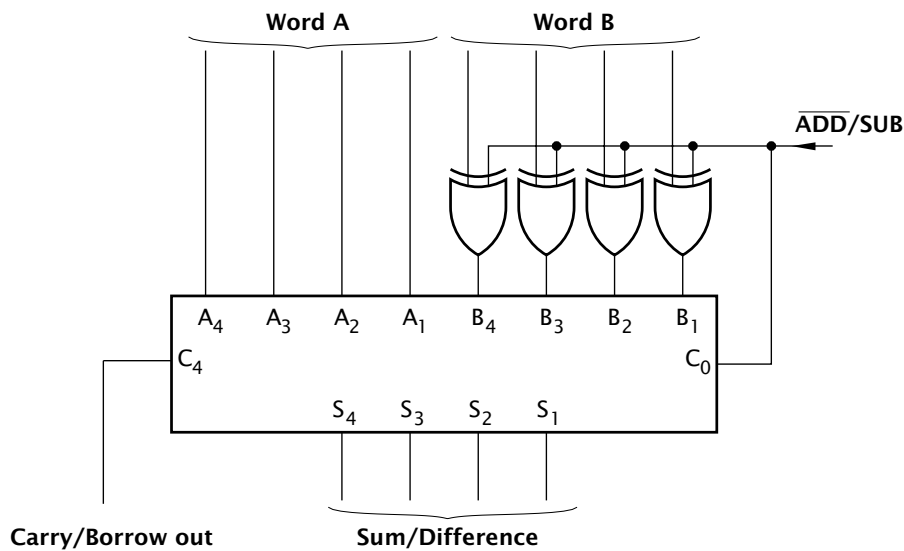
Figure 2.8 *Implementing a programmable adder/subtractor.*

Sequences of program operation codes are normally stored in an LSI Read-Only Memory (ROM). Consider the architecture illustrated in Fig. 2.10. This is essentially a 3 to 8-line decoder driving an 8×2 array of diodes. The 3-bit address selects only row n for each input combination n. If a diode is connected to this row, then it conducts and brings the appropriate column Low. The inverting 3-state output buffer consequently gives a High for each connected diode and Low where the link is broken. The pattern of diode links then defines the output code for each input. For illustrative purposes, the structure has been programmed to implement the 1-bit full adder of Fig. 2.7(a), but *any* two functions of three variables can be generated.

The diode matrix look-up table shown here is known as a **Read-Only Memory** (**ROM**), as its 'memory' is in the diode pattern, which is programmed in when the device is manufactured. Early devices, which were typically decoder/32×8 matrices, usually came in user-programmable versions in which the links were implemented with fusible links. By using

$S_2S_1S_0$	Operation	
0 0 0	Clear	(F=0000)
0 0 1	Subtract	(B-A)
0 1 0	Subtract	(A-B)
0 1 1	Add	(A+B)
1 0 0	EOR	(A ⊕ B)
1 0 1	OR	(A + B)
1 1 0	AND	(A • B)
1 1 1	Preset	(F=1111)

Word A   **Word B Mode Select S**

Carry-In

(17) (19) (1) (3)   (16) (18) (2) (4)   (7) (6) (5)   (15)

A3 A2 A1 A0   B3 B2 B1 B0   S2 S1 S0   C(n)

ALU
[74LS382]

C(n+4)   OVR   F3 F2 F1 F0

(14)   (13)   (12) (11) (9) (8)

Carry-Out

Overflow   Function Output F

(a) Function table   *(b) Logic diagram/pinning*
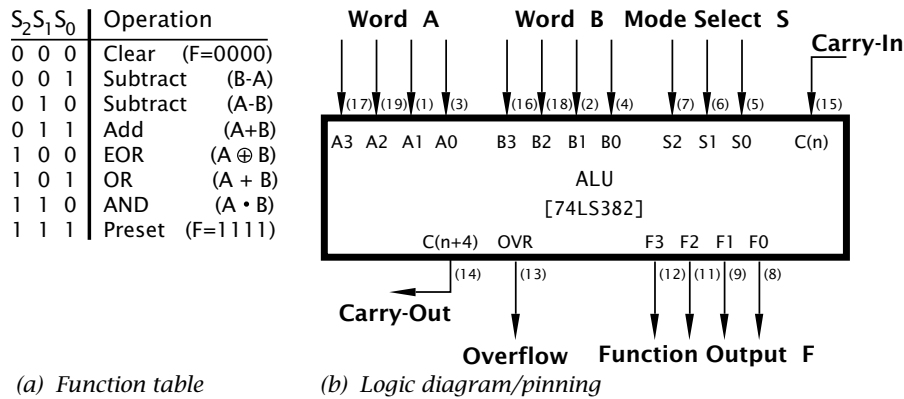
Figure 2.9 *The 74LS382 ALU.*

a high voltage, a selection of diodes could be taken out of contact. Such devices are called **Programmable ROMs (PROMs)**.

Fuses are messy when implementing the larger sizes of VLSI PROM necessary to store computer programs. For example, the 2764 PROM shown in Fig. 2.11 has the equivalent of 65,536 fuse/diode pairs, and this is a relatively small device capable of storing 8192 bytes of memory. The 2764 uses electrical charge on the floating gate of a metal-oxide field-effect transistor (MOSFET) as the programmable link, with another MOSFET to replace the diode. Charge can be tunnelled onto this isolated gate by, again, using a high voltage. Once on the gate, the electric field keeps the link MOSFET conducting. This charge takes many decades to leak away, but this can be dramatically reduced to about 30 minutes by exposure to intensive ultra-violet radiation. For this reason the 2764 is known as an **Erasable PROM** (**EPROM**). When an EPROM is designed for reusability, a quartz window is integrated into the package, as shown in Fig. 2.11. Programming is normally done externally with special equipment, known as PROM programmers, or colloquially as PROM blasters. Versions without windows are referred to as One-Time Programmable ROMs (OTPROMs), as they cannot easily be erased once programmed. They are however, much cheaper to produce and are thus suitable for small to medium-scale production runs.
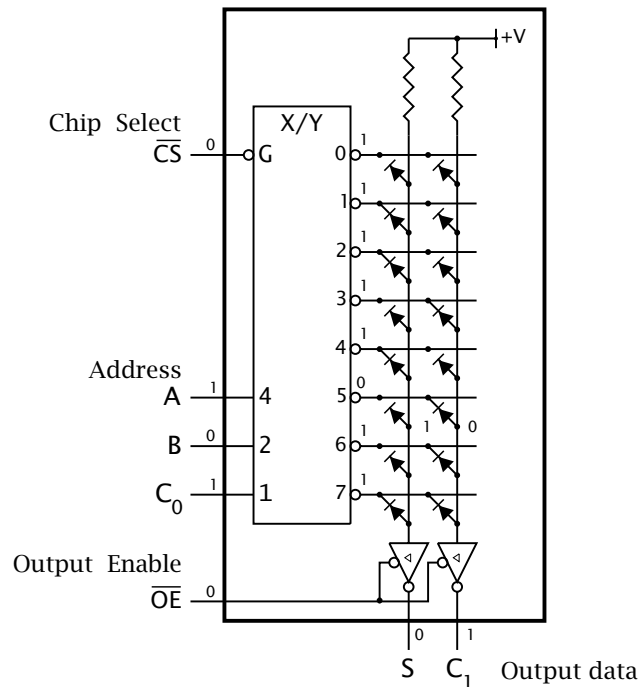
Figure 2.10 *A ROM-implemented 1-bit adder.*

There are PROM structures which can be erased electrically, often in situe in the circuit. These are known variously as Electrically-Erasable PROMs (EEPROMs) or flash memories. In the former case a large negative pulse at V_{PP} causes the captured electrons on the buried gate to tunnel back out. Generally the negative voltage is generated on the chip, which saves having to provide an additional external supply. The **flash** variant of EEPROM relies on hot electron injection rather than tunneling to charge the floating gate. The geometry of the cell is approximately half the size of a conventional EEPROM cell, which increases the memory density. Programming voltages are also somewhat lower.

Most modern EPROM/EEPROMs are fairly fast, taking around 150 ns to access and read. Programming is slow, at perhaps 10 ms per word, but

(a) Dual in-line package   *(b) Logic symbol*
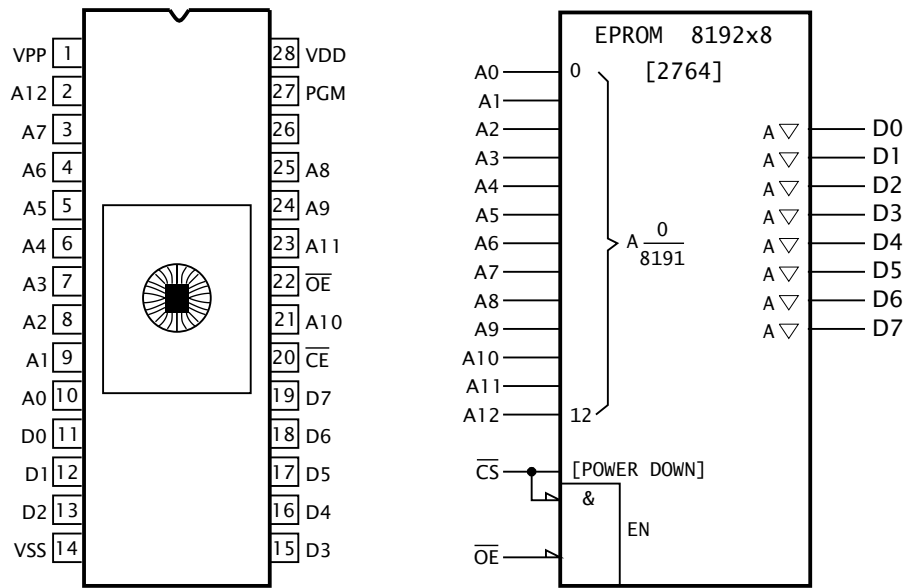
Figure 2.11 *The 2764 Erasable PROM.*

this is an infrequent activity. Flash EEPROM programs around 1000 times faster, in around $10\,\mu$s per cell.

All the circuits shown thus far are categorised as **combinational logic**. They have no memory in the sense that the output simply depends only on the present input, and not the sequence of events leading up to that input. Logic circuits, such as latches, counters, registers and read/write memories are described as **sequential logic**. Their output not only depends on the current input, but the sequence of prior inputs.

Consider a typical door bell push-switch. When you press such a switch the bell rings, and it stops as soon as you release it. This switch has no memory.

R S	Q
0 0	Q (no change)
0 1	1 (set)
1 0	0 (reset)

(a) Defining RS latch truth table
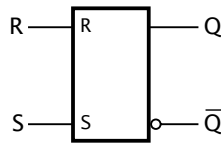
(b) Logic symbol with true/complement outputs
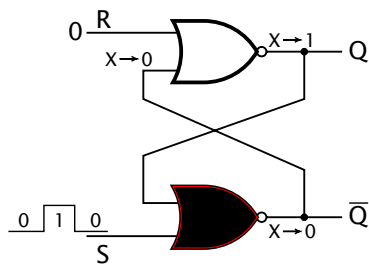
(c) Setting the latch
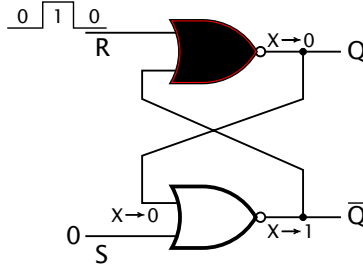
(d) Resetting the latch

Figure 2.12 *The R S latch.*

Compare this with a standard light switch. Set the switch and the light comes on. Moreover it remains on when you remove the stimulus (usually your finger!). To turn the light off you must reset the switch. Again it remains off when the input is taken away. This type of switch is known as a **bistable**, as it has two stable states. Effectively it is a 1-bit memory cell, that can store either an on or off state indefinitely.

A read-write memory, such as the 6264 device of Fig. 2.21, implements each bistable cell using two cross-coupled transistors. Here we are not concerned with this microscopic view. Instead, consider the two cross-coupled NOR gates of Fig. 2.12. Remembering from Fig. 1.3(c) on page 14 that any logic 1 into a NOR gate will always give a logic 0 output irrespective of the state of the other inputs, allows us to analyse the circuit:

- If the S input goes to 1, then output $\overline{Q}$ goes to 0. Both inputs to the top gate are now 0 and thus output Q goes to 1. If the S input now goes back to 0, then the lower gate remains 0 (as the Q feedback is 1) and the top gate output also remains unaltered. Thus the latch is *set*

by pulsing the S input.

- If the R input goes to 1, then output Q goes to 0. Both inputs to the bottom gate are now 0 and thus output $\overline{Q}$ goes to 1. If the R input now goes back to 0, then the upper gate remains 0 (as the $\overline{Q}$ feedback is 1) and the bottom gate output also remains unaltered. Thus the latch is *reset* by pulsing the R input.

In the normal course of events — that is assuming that the R and S inputs are not both active at the same time[5] then the two outputs are always complements of each other, as indicated by the logic symbol of Fig. 2.12(b).
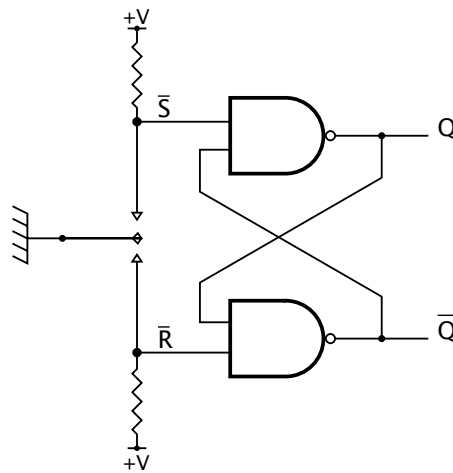
Figure 2.13 *Using a $\overline{R}\,\overline{S}$ latch to debounce a switch.*

There are many bistable implementations. For example, replacing the NOR gates by NAND gives a $\overline{R}\,\overline{S}$ latch, where the inputs are active on a logic 0. The circuit illustrated in Fig. 2.13 shows such a latch used to de-

---

[5]If they where, then both Q and $\overline{Q}$ go to 0. On relaxing the inputs, the latch will end up in one of its stable states, depending on the relaxation sequence. The response of a latch to a simultaneous Set and Reset is not part of the latch definition, shown in Fig. 2.12(a), but depends on its implementation. For example, trying to turn a light switch on and off together could end in splitting it in two!

bounce a mechanical switch. Manual switches are frequently used as inputs to logic circuits. However, most metallic contacts will bounce off the destination contact many times over a period of several tens of milliseconds before settling. For instance, using a mechanical switch to interrupt a computer/microprocessor will give entirely unpredictable results.

In Fig. 2.13, when the switch is moved up and hits the contact the latch is set. When the contact is broken, the latch remains unchanged, provided that the switch does not bounce all the way back to the lower contact. The state will remain Set no matter how many bounces occur. By symmetry, the latch will reset when the switch is moved to the bottom contact, and remain in this Reset state on subsequent bounces.

C	D	Q	
1	0	0	
1	1	1	(transparent)
0	X	Q	(freeze)

(a) D latch truth table
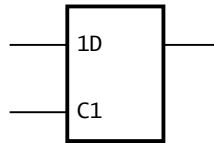
(b) D latch logic diagram

C	D	Q	
↑	0	0	
↑	1	1	(sample)
0	X	Q	
1	X	Q	(hold)
↓	X	Q	

(c) D flip flop truth table
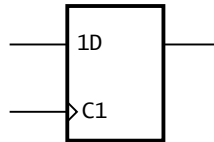
(d) D flip flop logic diagram

Figure 2.14 *The D latch and flip flop.*

The **D latch** is an extension to the R S latch, where the output follows the D (Data) input when the C (Control) input is active (logic 1 in our example) and freezes when C is inactive. The D latch can be considered to be a 1-bit memory cell where the datum is retained at its value at the end of the sample pulse.

In Fig. 2.14(b) the dependency of the Data input with its Control is shown by the symbology C1 and 1D. The 1 prefix to D shows that it de-

pends on any signal with a 1 suffix, in this case the C input. That is C1 clocks in the 1D data.

A flip flop is also a 1-bit memory cell, but the datum is only sampled on an *edge* of the control (known here as the Clock) input. The **D flip flop** described in Fig. 2.14(c) is triggered on a __/‾ (as illustrated in the truth table as ↑), but ‾__ clocked flip flops are common. The edge-triggered activity is denoted as > on a logic diagram, as shown in Fig. 2.14(d).

The 74LS74 shown in Fig. **??** on page **??** has two D flip flops in the one SSI circuit. Each flip flop has an overriding Reset ($\overline{R}$) and Set ($\overline{S}$) input, which are asynchronous — that is not controlled by the Clock input. MSI functions include arrays of four, six and eight flip flops all sampling simultaneously with a common Clock input.
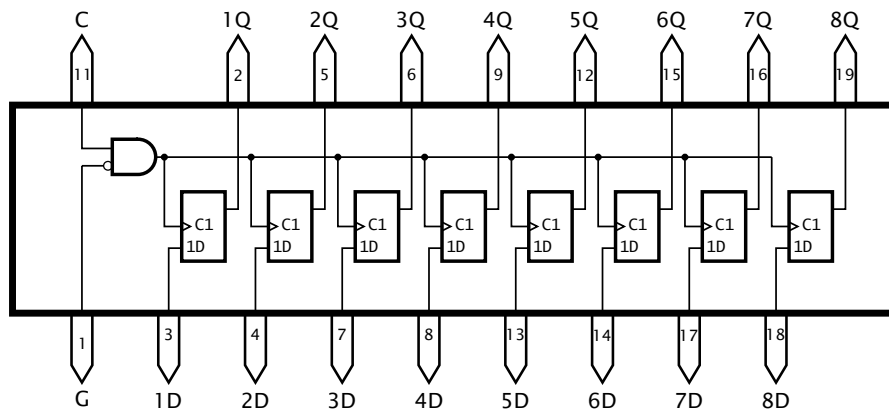
Figure 2.15 *The 74LS377 octal D flip flop array.*

The 74LS377 shown in Fig. 2.15 consists of eight D flip flops all clocked by the same single Clock input C, which is gated by input $\overline{G}$. Thus the 8-bit data 8D…1D is clocked in on the __/‾ of C if $\overline{G}$ is Low. In the ANSI/ISO logic diagram shown in Fig. **??** on page **??**, this dependency is indicated as G1→1C2→2D, which states that $\overline{G}$ enables the Clock input, which in turn acts on the Data inputs.

Arrays of D flip flops are known as **registers**; that is read/write memories that hold a single word. The 74LS377 is technically known as a

parallel-in parallel-out (PIPO) register, as data is entered in parallel (that is all in one go) and is available to read at one go. D latch arrays are also available, the 74LS373 octal PIPO register shown in Fig. **??** on page **??** is typical.

A pertinent example of the use of a PIPO register is shown in Fig. 2.16. Here an 8-bit ALU is coupled with an 8-bit PIPO register, accepting as its input the ALU output, and in turn feeding one input word back to the ALU.
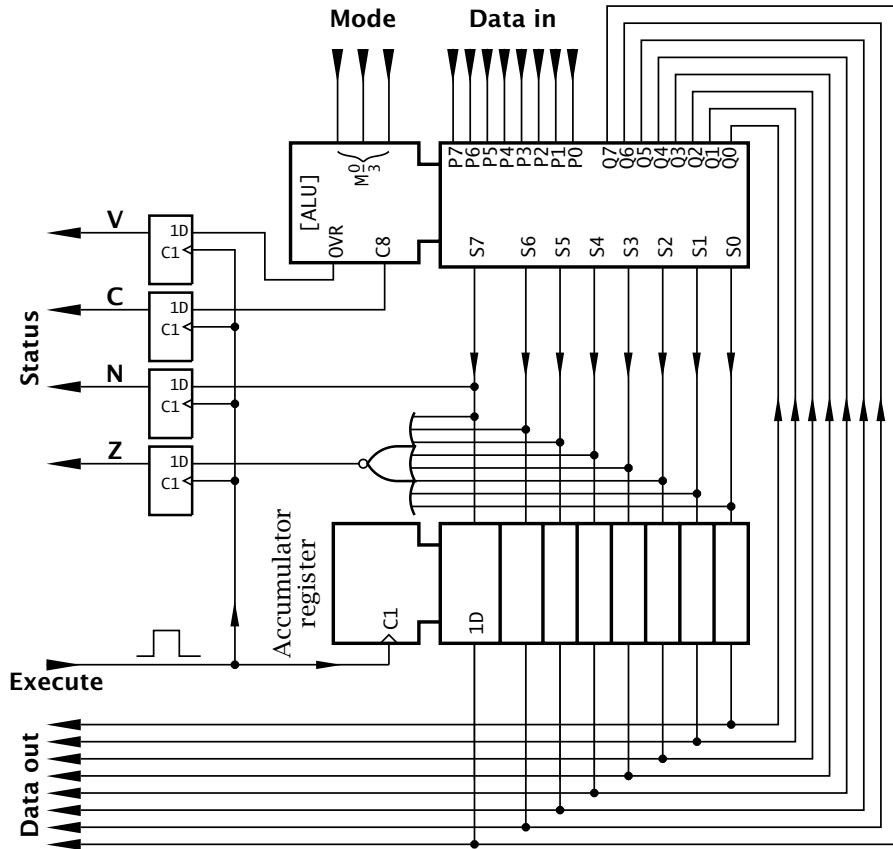
Figure 2.16 *An 8-bit ALU-accumulator processor.*

This register accumulates the outcome of a series of operations, and is sometimes called an **Accumulator** or **Working register**. To describe the operation of this circuit, consider the problem of adding two words A and B. The sequence of operations, assuming the ALU is implemented by cascading two 74LS382s might be:

1. Program step.

   - Mode = 000 (Clear).
   - Pulsing Execute loads the ALU output (0000 0000) into the register.
   - Data out is zero (0000 0000).

2. Program step.

   - Fetch Word A down to the ALU input.
   - Mode = 011 (Add).
   - _/‾_ Execute to load the ALU output (Word A + zero) into the register.
   - Data out is Word A.

3. Program step.

   - Fetch Word B down to the ALU input.
   - Mode = 011 (Add).
   - _/‾_ Execute to load the ALU output (Word B + Word A) into the register.
   - Data out is Word B plus Word A.

The sequence of operation codes, that is 000 — 100 — 100 constitutes the program. In practice each instruction would also contain the address (where relevant) in memory of the data to be processed; in this case the locations of Word A and Word B.

Each outcome of a process will have associated properties. For example it may be zero, be negative (most significant bit is 1), have a carry-out or 2's complement overflow. Such properties may be significant in the future progress of the program. In the diagram four D flip flops, clocked by Execute, are used to grab this status information. In this situation the

flip flops are usually known as **flags** (or sometimes semaphores). Thus we have **C, N, Z** and **V** flags, which form a Code Condition or Status register.

There are various other forms of register. The 4-bit **shift register** of Fig. 2.17(a) is an example of a serial-in serial-out (SISO) structure. In this instance the data held in the nth D flip flop is presented to the input of the $(n + 1)$th stage. On receipt of a clock pulse (or shift pulse in this context), this data moves into this $(n + 1)$th flip flop, i.e. effectively moving from stage n to stage $n + 1$. As all flip flops are clocked simultaneously, the entire word moves once right on each shift pulse.

In the example of Fig. 2.17 a 4-bit external data nybble is fed into the left-most stage bit by bit as synchronised by the clock. After four shift pulses the serial 4-bit word is held in the register. To get it out again, four further shifts moves the word bit by bit out of the shift register; this

(a) A 4-bit shift register

(b) Shifting 1011 into the register

(c) The ANSII/IEC logic symbol for a SIPO register
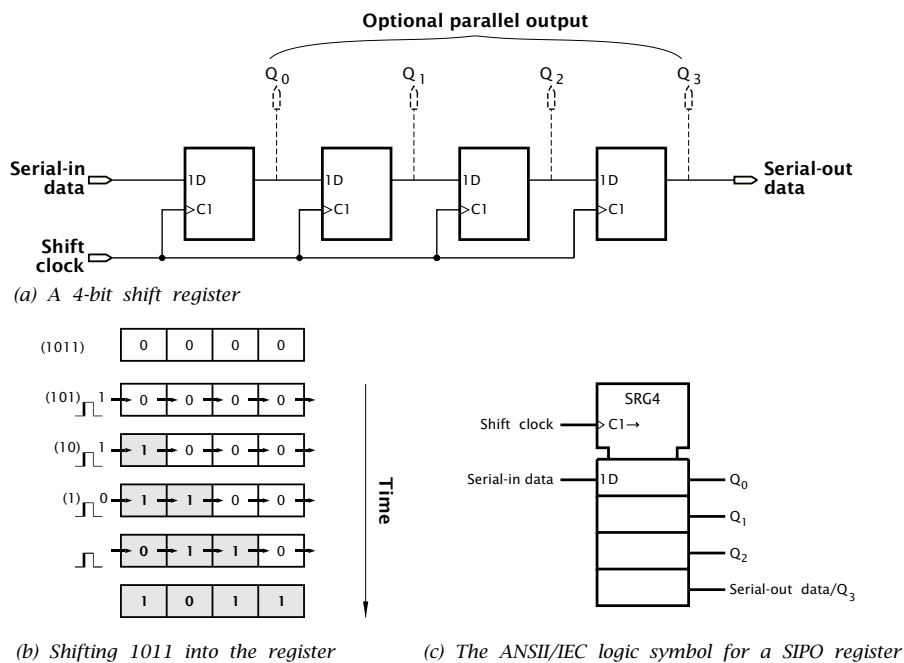
Figure 2.17 *The SISO shift register.*

is SISO. If the individual flip flops are accessible then the data can be accessed at one go, that is serial-in parallel-out (SIPO).

The logic diagram of Fig. 2.17(b) uses the → symbol affected by the clock input to indicate the shift action, C1 →. SRG4 indicates a Shift ReGister 4-stage architecture.

Other architectures include parallel-in serial-out which is useful for parallel to serial conversion. Counting registers (counters) increment or decrement on each clock pulse, according to a binary sequence. Typically an n-bit counter can perform a count of 2^n states. Some can also be loaded in parallel and thus act as a store.
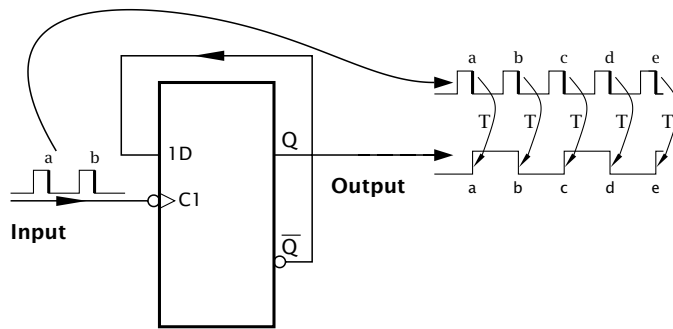
Figure 2.18 *The T flip flop.*

Consider the negative-edge triggered D flip flop shown in Fig. 2.18 where its $\overline{Q}$ output is connected back to the 1D input. On each ⌐_ at the Clock input C1 the data at the 1D input will be latched in to appear at the Q output. As it is the complement of this output that is fed back to the input, then the next time the flip flop is clocked the *opposite* logic state will be latched in. This constant alternation is called *toggling* and is depicted on the diagram by T. The output waveform resulting from a constant frequency input pulse train is half this frequency. This waveform is a precision squarewave, provided that the input frequency remains constant. This **T flip flop** is sometimes known as a binary or a divide-by-2.

T flip flops can of course be cascaded, as shown in Fig. 2.19(a). Here four ⌐_ triggered flip flops are chained, with the output of binary n clocking binary $n + 1$. Thus if the input Count frequency was 8 KHz, then
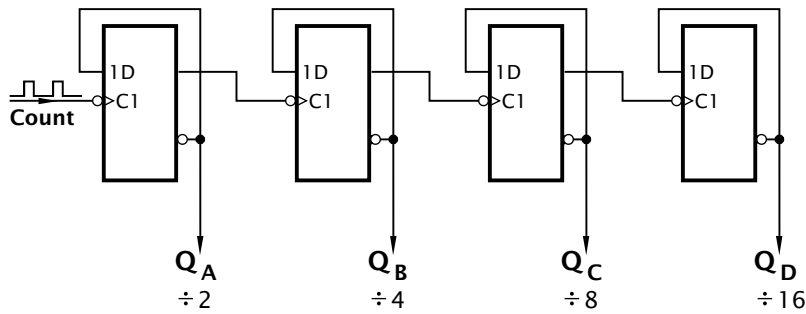
(a) Cascading toggle flip flops
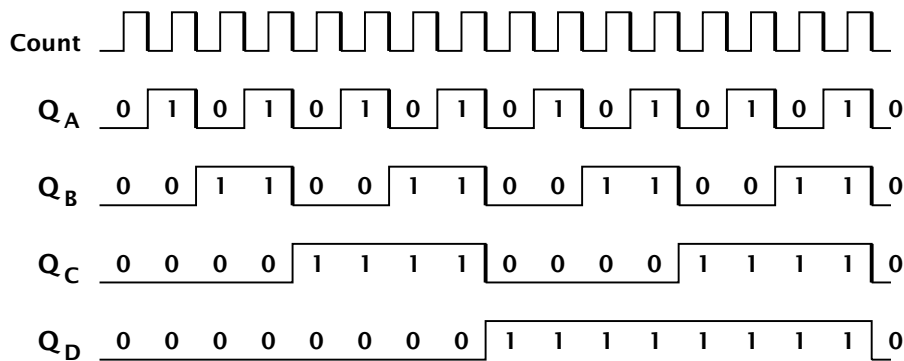
(b) Resulting waveforms

Figure 2.19 *A modulo-16 ripple counter.*

Q_A would be a 4 kHz square waveform and similarily Q_B would measure in at 2 kHz, Q_C at 1 kHz, Q_D at 500 Hz.

The waveform Q_A of Fig. 2.19(b) was derived in the same manner as in Fig. 2.18. Q_B is toggled on each ⌐‾⌐ of Q_A and likewise for the subsequent outputs. Marking a high as logic 1 and a low as logic 0 gives the 2^4 (16) positive-logic binary patterns as time advances, with the count rolling over back to state 0 on a continual basis. Each pattern remains in the register until the next event clocks the chain; an event being defined in our example as a ⌐‾⌐ at Count. Examining the sequence shows it to be a

natural 8-4-2-1 binary up count, incrementing from $0000b$ to $1111b$. In fact the circuit is a modulo-16 **binary counter**. A modulo-n count is the sequence taking only the first n numbers into account.[6]

In theory there is no limit to the number of stages that can be cascaded. Thus using eight T flip flops would give a modulo-256 (2^8) counter. In practice there is a small propagation delay through each stage and this limits the ultimate frequency. For example the 74LS74 dual D flip flop has a maximum propagation from an event at its Clock input to output of 25 ns. The maximum toggling frequency for a single stage, such as in Fig. 2.18, is given as 25 MHz. An 8-stage counter thus has a maximum ripple-through time of 200 ns. If such a **ripple counter** were clocked at the resulting 5 MHz ($\frac{1}{200\,\text{ns}}$) then no sooner than one particular code pattern has stabilized then the next one would begin to appear. This is only really a problem if the various states of the counter are to be decoded and used to control other logic. The decoding logic, such as shown in Fig. 2.20, may inadvertently respond to these short transient states and cause havoc. In such cases more sophisticated synchronous counter configurations are more applicable where the flip flops are clocked simultaneously and steered by the appropriate logic configuration to count in the desired sequence.

The circuit illustrated here implements an up count. If the complement $\overline{Q}$ lines are used as the outputs, but with the clocking arrangements remaining the same, then the count sequence will decrement, that is a down count. Likewise using $\underline{}\diagup\overline{}$ triggered flip flops, such as the 74LS74 dual flip flop (see Fig. 2.20), are used as the storage element, then the count will be down. It is easily possible to use some simple logic to combine the two functions to produce a programmable up/down counter. It is also feasible to provide logic to load the flip flop array in parallel with any number and then count up or down from that point. Such an arrangement can be thought of as a parallel-in counting register.

As well as the more obvious use of a counter register to totalize the number of events, such as cans of peas coming along a conveyor belt, there are other uses. One of these is to time a sequence of operations. In Fig. 2.20 a modulo-4 counter is used to address one section of a 74LS139

---

[6]Mathematically any number can be converted to its modulo-n equivalent by dividing by n. The remainder, or modulus, will be a number from 0 to $n-1$.
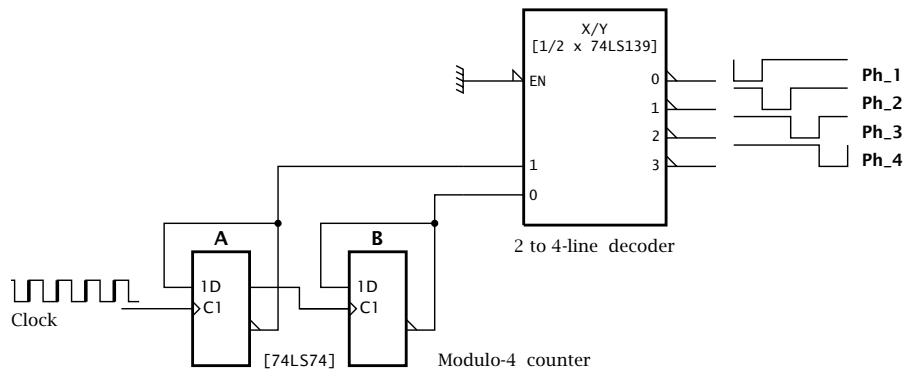
Figure 2.20 *Generating timing waveforms.*

2 to 4-line decoder, see Fig. 2.5(a). This detects each of the four states of
the counter, and the outcome is four time-separated outputs that can be
used to sequence, say, the operation of a computer's control section logic.
As a practical point, the complement $\overline{Q}$ flip flop outputs have been used
to address the decoder to compensate for the ‾‾_/‾ triggered action that
would normally give a down count. Larger counters with the appropriate
decoding circuitry can be used to generate fairly sophisticated sequences
of control operations.

The term register is commonly applied to a read/write memory that
can store a single binary word, typically 4 – 64 bits. Larger memories can
be constructed by grouping n such registers and selecting one of n. Such a
structure is sometimes known as a **register file**. For example, the 74LS670
is a 4×4 register file with a separate 4-bit data input and data output and
separate 2-bit address. This means that any register can be read at any
time, independently of any concurrent writing process.

Larger read/write memories are normally known as **read-write Random-
Access Memories**, or **RAMs** for short. The term random-access indicates
that any memory word may be selected with the same access time, ir-
respective of its position in the memory matrix.[7] This contrasts with a
magnetic tape memory, where the reel must be wound to the sector in

---

[7]Strictly speaking, ROMs should also be described as random access, but custom and
practice has reserved the term for read-write memories.

(a) Dual in-line package   *(b) Logic symbol*
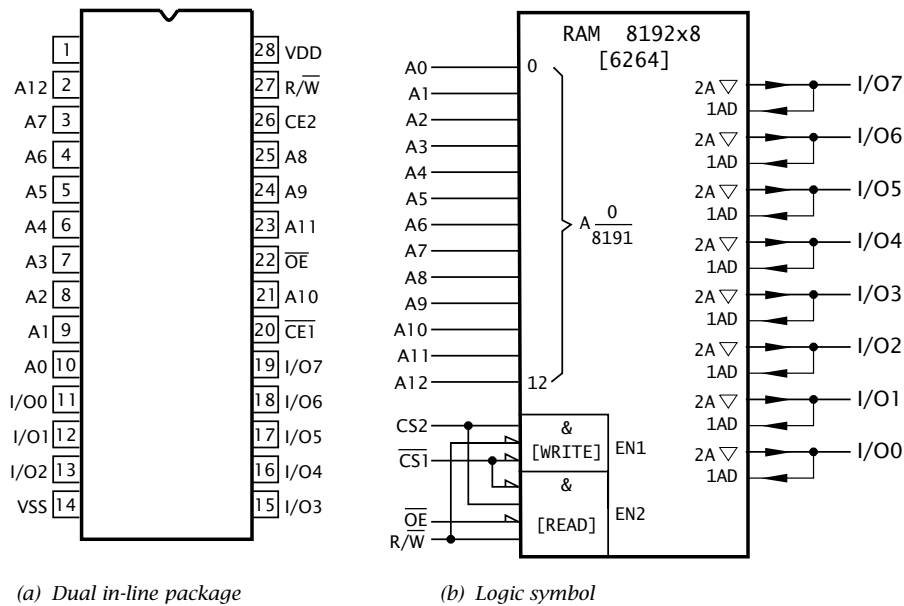
Figure 2.21 *The 6264* 8196 × 8 *RAM.*

question — and if this is at the end of the tape!

For our example, Fig. 2.21 shows the 6264 RAM. This has a matrix of 65,536 (2^{16}) bistables organized as an array of 8192 (2^{13}) words of 8 bits. Word n is accessed by placing the binary pattern of n on the 13-bit Address pins A12...A0.

When in the Read mode (Read/$\overline{\text{Write}}$ = 1), word n will appear at the eight data outputs (I/O7...I/O0) as determined by the state n of the address bits. The A symbol at the input/outputs (as was the case in Fig. 2.11) indicates this addressability. In order to enable the 3-state output buffers, the $\overline{\text{Output Enable}}$ input must be Low.

The addressed word is written into if R/$\overline{\text{W}}$ is Low. The data to be written into word n is applied by the outside controller to the eight I/O pins. This bi-directional traffic is a feature of computer buses; for example see Fig. **??** on page **??**.

In both cases, the RAM chip as a whole is enabled when $\overline{CS1}$ is Low and CS2 is High. Depending on the version of the 6264, this access from enabling takes around 100–150 ns. There is no upper limit to how long the data can be held, provided power is maintained. For this reason, the 6264 is described as static (SRAM). Rather than using a transistor pair bistable to implement each bit of storage, data can be stored as charge on the gate-source capacitance of a single field-effect transistor. Such charge leaks away in a few milliseconds, so needs refreshed on a regular basis. Dynamic RAMs (DRAMs) are cheaper to fabricate than SRAM equivalents and obtainable in larger capacities. They are usually found where very large memories are to be implemented, such as found in a personal computer. In such situations, the expense of refresh circuitry is more than amortized by the reduction in cost of the memory devices.

Both types of Read/Write memories are volatile, that is they do not retain their contents if power is removed. Some SRAMs can support existing data at a very low holding current and lower than normal power supply voltage. Thus a backup battery can be used in such circumstances to keep the contents intact for many months.

Stored program processing

If we take the Arithmetic Logic Unit (ALU)/data register pair depicted in Fig. 2.16 on page 34 and feed it with function codes, then we have in essence a programmable processing unit. These command codes may be stored in digital memory and constitute the system's **program**. By *fetching* these **instructions** down one at a time we can execute this program. Memory can also hold data on which the ALU operates. This structure, together with its associated data paths, decoders and logic circuitry is known as a digital **computer**.

In Part II we will see that microprocessor architecture is modelled on that of the computer. As a prelude to this we will look at the architecture and operating rhythm of the computer structure and some characteristics of its programming. Although this computer is strictly hypothetical, it has been very much 'designed' with our book's target microprocessor in mind.

After reading this chapter you will:

- *Appreciate the von Neumann structure, with its* common *Data highway connecting memory, input, output and processor.*

- *Understand the fetch and execute rhythm and its interaction with memory and the Central Processing Unit's (CPU's) internal registers.*

- *Understand the concept of an address as a pointer to where data or program code is stored in memory.*

- *Comprehend the structure of an instruction and appreciate that the string of instructions necessary to implement the task is known as a program.*

- *Have an understanding of a basic instruction set, covering data movement, arithmetic, logic and conditional branching categories.*

- *Understand how Immediate, Direct and Indirect address modes permit an instruction to target an operand for processing.*
- *To be able to write short programs using a symbolic assembly-level language and appreciate its 1:1 relationship to machine code.*
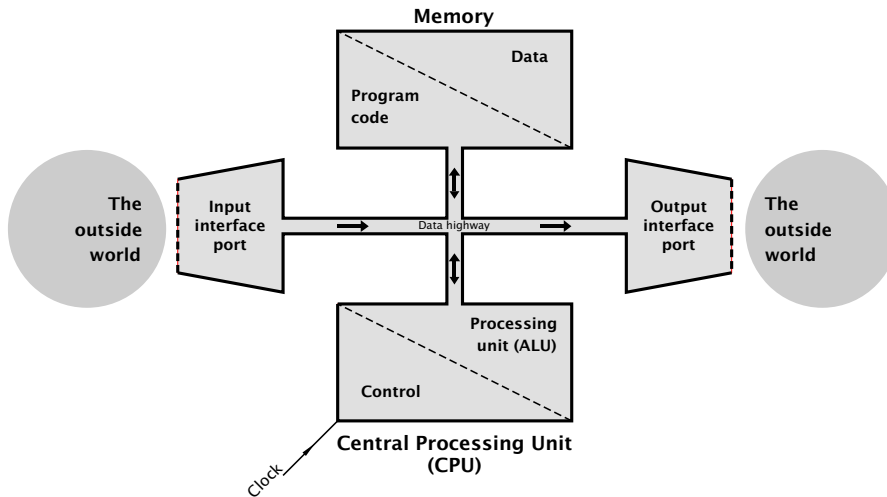
Figure 3.1 *An elementary von Neumann computer.*

The architecture of the great majority of general-purpose computers and microprocessors is modelled after the **von Neumann** model shown in Fig. 3.1.[1] The few electronic computers in use up to the late 1940s either only ever ran one program (like the war time code breaking Colossus) or else needed partly rewired to change their behavior (for example the

---

[1]Von Neumann was a Hungarian mathematician working for the American Manhattan nuclear weapons program during the 2nd World war. After the war he became a consultant for the Moore School of Electrical Engineering at the University of Pennsylvania's EDVAC computer project, for which he was to employ his new concept where the program was to be stored in memory along with its data. He published his ideas in 1946 and EDVAC became operational in 1951. Ironically, a somewhat lower key project at Manchester University made use of this approach and the Mark 1 executed its first stored program in June 1948! This was closely followed by Cambridge University's EDSAC which ran its program in May 1949, almost two years ahead of EDVAC.

ENIAC). The web site
`http://www.nibec.ulst.ac.uk/sidk/essence/ch3_7.htm`
gives historical and technical details of these prehistorical machines.

Von Neumann's great leap forward was to recognise that the program could be stored in memory along with any data. The advantage of this approach is flexibility. To alter the program simply change the bit pattern in the appropriate area of memory. In essence, the von Neumann architecture comprises a Central Processing Unit (CPU), a memory and a connecting highway carrying data back and forth. In practice the CPU must also communicate with the environment outside the computer. For this purpose data to and from suitable interface ports are also funnelled through the data highway.

Looking at these elements in a little more detail.

The Central Processing Unit

The CPU consists of the ALU/Accumulator register together with the associated control logic. Under the management of the control unit, program instructions are fetched from memory, decoded and executed. Data resulting from, or used by, the program is also accessed from memory. This fetch and execute cycle constitutes the operating rhythm of the computer and continues indefinitely, as long as the system is activated.

Memory

Memory holds the bit patterns which define the program. These sequences of instructions are known as the **software**. The word is a play on the term hardware; as such patterns do not correspond to any physical rearrangement of the circuitry. Memory holding software should ideally be as fast as the CPU, and normally uses semiconductor technologies, such as that described in the last chapter.[2] This memory also holds data being processed by the program.

Program memories appear as an array of cells, each holding a bit pattern. As each cell ultimately feeds the single data highway, a decoding network is necessary to select only *one* cell at a time for interrogation. The computer must target its intended cell for connection by driving this

---

[2]This wasn't always so; the earliest practical large high-speed program memories used miniature ferrite cores (donuts) that could be magnetized in any one of two directions. Core memories were in use from the 1950s to the early 1970s, and program memory is sometimes still referred to as core.

decoder with the appropriate code or **address**. Thus if location 602Eh is
to be read, then the pattern 0110 0000 0010 1110b must be presented
to the decoder. For simplicity, this address highway is not shown here,
but see Fig. **??** on page **??**.

This addressing technique is known as random access, as it takes the
same time to access a cell regardless of where it is situated in memory.
Most computers have large backup memories, usually magnetic or optical
disk-based or magnetic tape, in which case access does depend on the
cell's physical position. Apart from this sequential access problem, such
media are normally too slow to act as the main memory and are used
for backup storage of large arrays of data (eg. student exam records) or
programs that must be loaded into main memory before execution.

The Interface Ports

To be of any use, a computer must be able to interact with its environment.
Although conventionally one thinks of a keyboard and screen, any of a
range of physical devices may be read and controlled. Thus the flow of
fuel injected into a cylinder together with engine speed may be used to
control the instant of of spark ignition in the combustion chamber of a
gas/petrol engine.

Data Highway

All the elements of our computer are wired together with the one *common*
data highway, or bus. With the CPU acting as the master controller, all
information flow is back and forward along these shared wires. Although
this is efficient, it does mean that only one thing can happen at any time,
and this phenomena is sometimes known as the von Neumann bottleneck.

The fetch instruction down — decode it — execute sequence, the so
called **fetch and execute cycle**, is fundamental to the understanding of
the operation of the von Neumann computer and microprocessor. To
illustrate this operating rhythm we look at a simple program that takes a
variable called NUM1 adds the constant 65h (101d) to it and assigns the
resultant value to the variable called NUM2. In the high-level language **C**
this may be written as:[3]

```
NUM2 = NUM1 + 101;
```

---

[3] If you are more familiar with PASCAL or Modula-2, then the program statement would
be expressed as NUM2 := NUM1 + 101

Memory

Program space

0000	0001		0003	0004		0005	0006	0007		0008	0009	000A	000B	000C
ldaa	**NUM1**		**adda**	**#65h**		**staa**	**NUM2**			— —	— —	— —	
B6	**602C**		**8B**	**65**		**B7**	**602D**						

0010		0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C
		— —	— —		— —			— —		— —		

0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C
— —		— —										

0030	0031	0032	0033	0034	0035		0037	0038	0039	003A	003B	003C
— —	— —		— —	— —	— —		— —	— —	— —	— —	— —	

Data space

6008	6009	600A	600B	600C	600D	600E	600F
— —	— —	— —	— —	— —	— —		

6018	6019	601A	601B	601C	601D	601E	601F
— —	— —	— —	— —		— —	— —	

6028	6029	602A	602B	602C	602D	602E	602F
— —	— —	— —	— —	**NUM1**	**NUM2**	— —	

6038	6039	603A	603B	603C	603D	603E	603F
— —	— —	— —	— —	— —	— —		

R/W̄ = 1 (Read)

Address bus

Data bus

BASIC

Central Processing Unit

ADD_BUFF	Address register buffer	Data register buffer	DATA_BUF
0000			B6

PC	Program Counter	Instruction register	IR
0000			B6

Arithmetic Logic Unit

ALU

A		ID

Control signals to internal resources

Accumulator

Instruction decoder

Clock

Figure 3.2 *A snapshot of the CPU fetching down the first instruction.*

A rather more detailed close-up of our computer, which I have named BASIC (for Basic All-purpose Stored Instruction Computer) is shown in Fig. 3.2. This shows the CPU and memory, together with the common data highway (or **bus**) and an Address bus. Looking first at the individual components of the CPU.

Data Buffer

The DATA_BUF holds the last one or two bytes fetched from the Data bus. As shown here, this is the code for the first instruction ldaa (LoaD Accumulator A) B6h.

Instruction Register

The last fetched instruction code (usually termed **op-code**, short for operation code) is stored in the IR, feeding the Instruction decoder.

Instruction Decoder

The ID is the 'brains' of the CPU, deciphering the op-code and sending out the appropriate sequence of signals necessary to locate the operand and to configure the ALU to execute the operation.

Arithmetic Logic Unit

The ALU carries out an arithmetic or logic operation as commanded by its function code generated by the Instruction Decoder.

Accumulator Register

A is the ALU's working register. Most instructions use A to hold either the source or the destination operand; for example suba #6 which subtracts the constant 06 from the Accumulator register and then places the result back in A.

Program Counter

Instructions are normally stored sequentially in memory, and the PC is the counter which keeps track of the current instruction word. This register is sometimes called (more sensibly) an Instruction Pointer. Loading the PC with a new value disrupts the orderly count and causes the execution sequence to jump or branch to another part of the program.

Address Buffer

When the CPU wishes to fetch an instruction word, it transfers the contents of the PC into the ADD_BUFF register. This directly addresses the memory via the Address bus. The resulting data is connected to the CPU

via the Data bus and loaded into the DATA_BUF register. During this time the R/$\overline{\text{W}}$ direction control line is 1 to indicate a read cycle. Where the CPU wishes to access data from memory (as opposed to an instruction), it places the appropriate address in ADD_BUFF. R/$\overline{\text{W}}$ is logic 1 where the CPU wishes to read data and 0 where data is to be written (from DATA_BUF to memory).

Data Bus

This bus is a set of eight (in this case) conductors acting as a bidirectional information highway between the CPU's Data register buffer and memory. Binary patterns, representing either program or data, may be read from memory (R/$\overline{\text{W}}$ = 1) or sent out and written to memory (R/$\overline{\text{W}}$ = 1) along this one *common* link.

Address Bus

As the Data bus is potentially connected to all memory cells, some means of enabling just one targeted cell at any one time is required. The Address bus carries location information as a 16-bit binary pattern. Decoding circuits will switch the addressed cell onto the Data bus when the memory circuit is enabled, see Figs 2.11 on page 29 and 2.21 on page 41.

Control Bus

The Control bus is the set of miscellaneous signals that indicate to the outside world the status of the processor or allow external circuits control over the processor operation. Our BASIC CPU has three Control signals:

- R/$\overline{\text{W}}$ is high when the CPU is reading data in from the Data bus and low when data is being written out to memory.

- Clock times the CPU with one read or write cycle taking one Clock period. Typically the Clock frequency ranges between 1 – 300 MHz.

- Reset is a signal from an outside agency (maybe the operator). In our BASIC system, Reset sets the Program Counter to address 0000h, the beginning of the program.

As shown in the diagram, I have depicted memory as an array of cells (or pigeon holes), each with an unique address. This is shown divided into two sectors; one holding the **program code** (sometimes called text) and one the **data code**. Although these two sectors may physically be part of the same memory circuits, typically different memory technologies

are used for the two functions. Thus program code (also fixed tables of constants and such like) may be located in ROM, whilst alterable data (i.e. variable objects) are in RAM. I have, quite arbitrarily, originated the program code at address $0000h$ and data at $6000h$.

Each instruction code in memory is 8 bits (a byte) long. This is usually followed by one or two bytes relating to where in memory the operand is or sometimes the operand itself.   The first instruction is stored as B6-60-2Ch.[4] As the Data bus is only 8 bits wide, *two or three* read actions are required to fetch down each instruction. The first of these fetches is shown in the diagram in which the op-code B6h (or 1011 0110b if you prefer) has been brought down through DATA_BUF into IR. The memory cell involved is shown shaded.

So much for the CPU and memory. Let us look at the program itself. There are three instructions in our illustrative software, and as we have already observed the task is to copy the value of a byte-sized variable NUM1 plus $101d$ (65h) into a variable called NUM2, i.e.

```
NUM2 = NUM1 + 101;
```

We see from our diagram that the variable named NUM1 is simply a symbolic representation for "the byte contents of 602Ch", and similarly NUM2 is a much prettier way of saying "the byte contents of 602Dh".

Now as far as the computer is concerned, our program is, starting at location $0000h$:

```
101101100110000000101100
1000101101100101
101101110110000000101101
```

Unless you are a CPU this is not much fun![5]

Using hexadecimal is a little better.

```
B6602C
8B65
B7602D
```

---

[4]Remember that we are only using hexadecimal notation as a human convenience. If you took an electron microscope and looked inside these cells you would only 'see' 1011 0110 0110 0000 0010 1100.

[5]I know; I have programmed this way back in the primitive middle 1970s.

but is still instantly forgettable. Furthermore, the CPU still only under-
stands binary, so you are likely to have to use a translator program run-
ning on, say a PC, to translate from hexadecimal to binary.

If you are going to use computer aid, it makes sense to go the whole
hog and express the program using symbolic representations of the var-
ious instructions (e.g. `clra` for CLeaR Accumulator, `suba` for SUBtract
from Accumulator) and for variables' addresses. Doing this, our program
becomes:

```
ldaa  NUM1  ; Copy the byte-sized variable NUM1 down to A
adda  #101  ; Add to it the constant 101 decimal (65h)
staa  NUM2  ; Copy NUM1+65h from A into NUM2
```

where text after a semicolon is comment.

Chapter 7 is completely devoted to the process of translation from this
assembly-level source code to machine readable binary. Here it is only
necessary to look at the general symbolic form of an instruction which is
one of these three:

```
instruction mnemonic
instruction mnemonic    <address of operand>
instruction mnemonic    <literal operand>
```

A few instructions have no explicit operand, such as `rts` (ReTurn from
Subroutine). Most instructions, including our three here, have an operand
field. This either is the address of a memory-located datum, such as
`clr 6000h` (Clear the contents of memory location $6000h$) or the literal
data constant itself, such as `suba #6` (SUBtract the constant six from the
Accumulator). Thus we cannot just say `adda`, we need to say "add some-
thing to the Accumulator", for example `adda 06000h` means "add the byte
contents of $6000h$ to the contents of the Accumulator register and place
the outcome back in the Accumulator register". This could be written
as: `(a) <- (6000) + (a)`, where the brackets mean contents of and `<-`
means becomes. This notation is called **register transfer language** (**rtl**).
In writing programs using assembly-level symbolic representation, it is
important to remember that each instruction has a *one to one correspon-
dence to the underlying machine instructions and its binary code.*

All our examples instructions use the Accumulator register A *directly*
as the source or destination of the datum. The first and last specify an

absolute address, which is the actual location in memory of the datum. It is easier for us humans to give these variables symbolic names, such as NUM1, but actually they are addresses. The middle instruction adds a *constant* number rather than a variable in memory. This constant is indicated thus as #65h, where the hash (pound in North America) symbol signifies a constant (or literal) in assembly language. The constant byte itself really is stored in memory, as the second byte of the instruction. If you leave the # out[6] then the instruction adda 65h will be translated as "add the byte-sized contents of 0065h to A", rather than "add the *constant* 65h to A". In rtl (a)<-(0065h)+(a) instead of (a)<-0065h+(a).

The essence of computer operation is the rhythm of the **fetch and execute cycle**. Here, each instruction is successively brought down from memory (fetched), interpreted and then executed. In order to illustrate this let us trace through our example program. We assume that our computer, that is the Program Counter, is reset to 0000*h*.

Fetch cycle 1(a) ....Fig. 3.3

- Program Counter (0000*h*) to ADD_BUFF and on to Address bus.
- First op-code (Load byte to A) in memory then appears on the Data bus, through to DATA_BUF and IR.
- Program Counter incremented.

Fetch cycle 1(b)

- Program Counter (0001*h*) to ADD_BUFF and on to Address bus.
- The upper byte of the 2-byte address of the variable NUM1 (60*h*) in memory then appears on the Data bus, through to DATA_BUF.
- Program Counter incremented.

Fetch cycle 1(c)

- Program Counter (0002*h*) to ADD_BUFF and on to Address bus.
- The lower byte of the 2-byte address of the variable NUM1 (2C*h*) in memory then appears on the Data bus, through to DATA_BUF.
- Program Counter incremented.

Execute 1

---

[6]As you surely will do on occasion!

(a) Fetch 1, three cycles
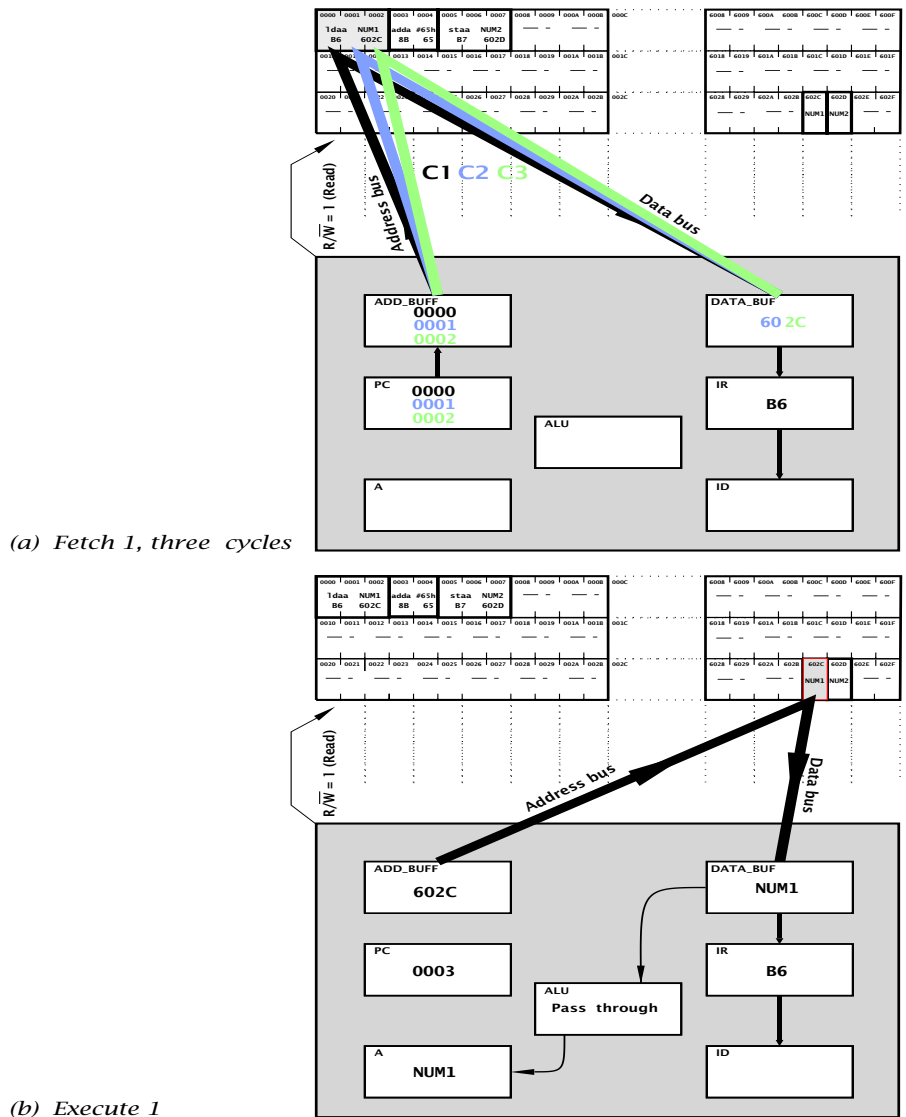
(b) Execute 1

Figure 3.3 *Fetch and execute the first instruction,* `ldaa  NUM1`.

(a) Fetch 2, two cycles
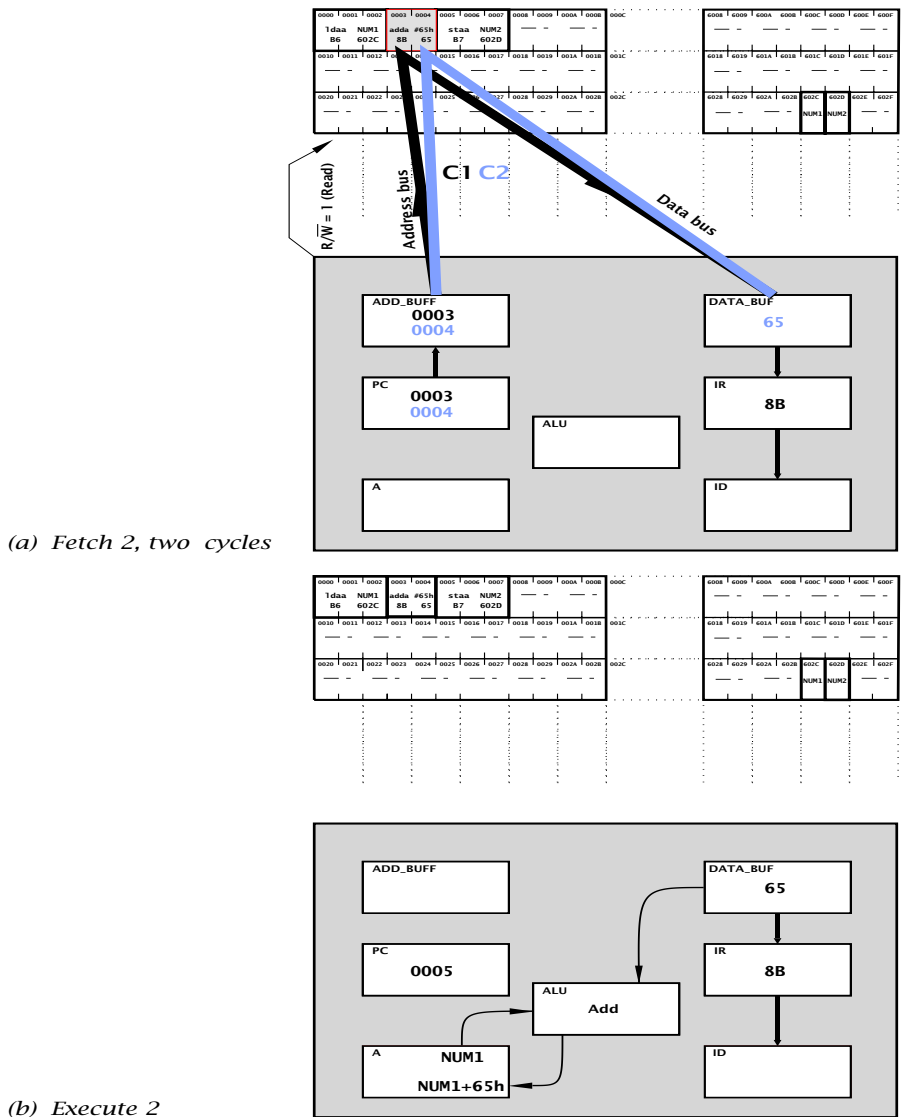
(b) Execute 2

Figure 3.4 *Fetch and execute the second instruction,* adda #65h.

(a) Fetch 3, three cycles
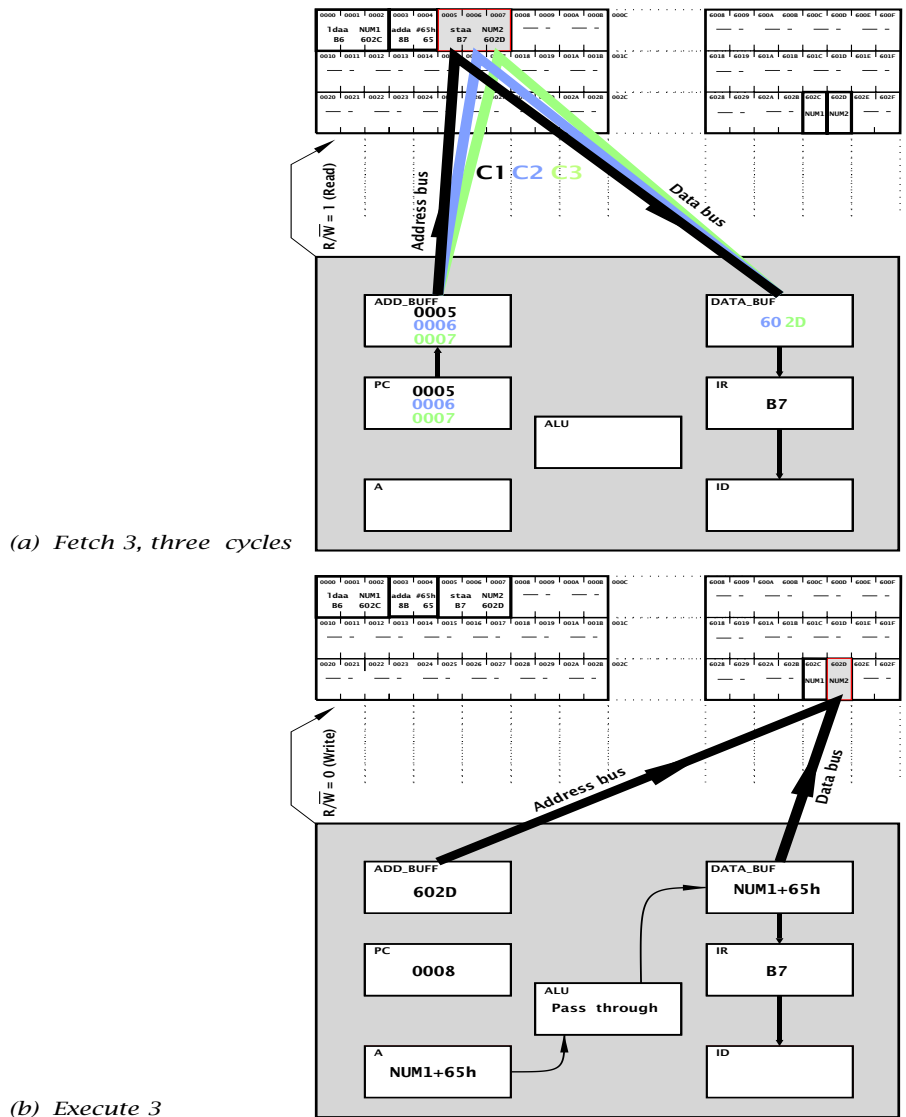
(b) Execute 3

Figure 3.5 *The final fetch and execute process,* `staa  NUM2`.

- The operand address 602Ch to ADD_BUFF and on to Address bus.
- Resulting data (NUM1) is read onto the Data bus (R/$\overline{\text{W}}$ = 0). through to DATA_BUF.
- The ALU is configured to Pass Through mode, which feeds NUM1 through to the Accumulator register A.

Fetch cycle 2(a) ....Fig. 3.4

- Program Counter (0003h) to ADD_BUFF and on to Address bus.
- Second op-code (Add *constant* byte to A) in memory then appears on the Data bus, through to DATA_BUF and IR.
- Program Counter incremented.

Fetch cycle 2(b)

- Program Counter (0004h) to ADD_BUFF and on to Address bus.
- The constant operand (65h) in memory then appears on the Data bus and through to DATA_BUF.
- Program Counter incremented.

Execute 2

- The ALU is configured to Add mode, the outcome of which (NUM1+65h) is placed in A.

Fetch cycle 3(a) ....Fig. 3.5

- Program Counter (0005h) to ADD_BUFF and on to Address bus.
- Third op-code (Store byte from A to memory) in memory then appears on the Data bus, through to DATA_BUF and IR.
- Program Counter incremented.

Fetch cycle 3(b)

- Program Counter (0006h) to ADD_BUFF and on to Address bus.
- The upper byte of the address of the variable NUM2 (60h) in memory then appears on the Data bus, through to DATA_BUF.
- Program Counter incremented.

Fetch cycle 3(c)

- Program Counter (0007h) to ADD_BUFF and on to Address bus.
- The lower byte of the address of the variable NUM2 (2Dh) in memory then appears on the Data bus, through to DATA_BUF.
- Program Counter incremented.

Execute 3

- The operand address 602Dh to ADD_BUFF and on to Address bus.
- The ALU is configured to Pass Through mode, which feeds the contents of A through to DATA_BUF and to the Data bus. The data is stored in memory by bringing R/$\overline{W}$ to logic 0 for a write action.

Notice how the Program Counter is automatically advanced during each fetch cycle. This sequential advance will continue indefinitely until an instruction to modify the PC occurs, such as jmp 0200h. This would place the address 0200h into the PC, overwriting the normal incrementing process, and effectively causing the CPU to jump to whatever instruction was located at 0200h. Thereafter, the linear progression would continue.

Although our program doesn't do very much, it only takes a few microseconds to execute each instruction. Several hundred thousand unimpressive operations each second can amount to a great deal! Nevertheless, it hardly rates highly in the annals of software, so we will wrap up our introduction to computing by looking at some slightly more sophisticated examples.

Writing a program is somewhat akin to building a house. Given a known range of building materials, the builder simply puts these together in the right order. Of course there are tremendous skills in all this; poor building techniques lead to houses that leak, are drafty and eventually may fall down!

It is possible to design a house at the same time as it is being built. Whilst this may be quite feasible for a log cabin, it is likely that the final result will not remain rain proof very long, nor will it be economical, maintainable, ergonomic or very pretty. It is rather better to employ an architect to design the edifice before building commences. Such a design is at an abstract level, although it is better if the designer is aware of the technical and economic properties of the available building materials.

Unfortunately much programming is of the 'on the hoof' variety, with little thought of any higher-level design. In the software arena this means devising strategies and designing data structures in memory. Again, it is better if the design algorithms keep in mind the materials of which the program will be built; in our case the machine instructions.

At the level of our examples in this chapter, it will be this coding (building) task we will be mostly concerned with. Later chapters will cover more advanced structures which will help this process, and we will get more practice at devising strategies and data structures.
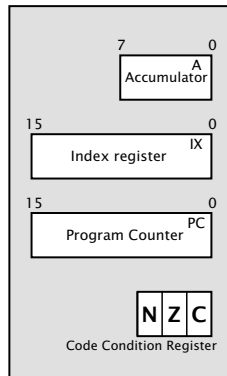
Figure 3.6 *Programmer's model.*

In order to code software we must have a knowledge of the register architecture of the computer/microprocessor and of the individual instructions. Figure 3.6 shows the **programming model** we will use for our exercises. This shows all registers that can be 'got at' by the program. I have added two registers to the previous complement. An **Index register** (**IX**) complements the Data register and is primarily meant to point to an object in memory. The Code Condition Register (**CCR**) comprises three flip flops or flags which are used to tell the software something about the outcome from an instruction. Thus the **C** flag is primarily the Carry bit from the last addition (or borrow from a subtraction). The **Z** flag is set if the operation result is zero, and the N flag is set if the operation result has its most significant bit set to 1 (which is the sign bit if the object is to

be treated by the programmer as a signed 2's complement number).

Table 3.1 shows all the instructions supported by the BASIC computer. Before looking at these, let us discuss the concept of the **address mode**. Most instructions act on data, which may be in internal CPU registers or out in memory. Thus the location of such operands must be part of the instruction. It isn't sufficient to simply state `clr` — Clear what? There are different ways of specifying the operand location; for instance `clra` and `clr 6000h` are legitimate manifestations of the same `clr` instruction. In the first case the target is the Accumulator and in the second, out in memory at the fixed address 6000*h*. The different ways of pointing out an operand's location are the address modes. Which address mode an instruction is to use is specified by some of the bits in its operation code. They are:

Inherent

Some instructions have no operand data. Such instructions either target an internal register or do not explicitly alter any register. Examples of the former are `clra` (CLeaR Accumulator). All Inherent instructions have only a one-byte op-code, with any register information being encoded within these 8 bits. Thus `clra` is coded as 4F*h* or 01001111*b*.

Immediate

This is used when the operand is *fixed* data; for example:

```
adda  #120    ; Add the constant 120 decimal to A
```

Note the use of the # symbol to denote that the following number is *constant data*. If we leave this out, e.g. `adda 120`, then this is interpreted as "add the contents of *address* 0120*d* to A". The destination of such instructions will be an internal register. Where this register is 8 bits wide (i.e. the Accumulator) then the immediate data is a single byte following the op-code in memory. For example, the instruction `adda #78h` is coded as 8B-**78***h*. For 16-bit registers (i.e. the Index register) the literal data is located as two bytes following the op-code, e.g. `ldx #1234h` is represented in program memory as CE-**12-34***h*.

Absolute

Here the absolute address of the operand follows the op-code. For example:

```
clr.b  6050h   ; Clear the byte at memory address 6050h
```

Instruction	Description	#	Inher	Absol	Index	N	Z	C
Arithmetic								
adda [ea]	Add to A	*		*	*	√	√	√
adca [ea]	Add to A with Carry	*		*	*	√	√	√
clr [ea]	Clear memory			*	*	0	1	0
clra	Clear A		*			0	1	0
dec [ea]	Decrement memory			*	*	√	√	•
deca	Decrement A		*			√	√	•
dex	Decrement IX		*			•	√	•
inc [ea]	Increment memory			*	*	√	√	•
inca	Increment A		*			√	√	•
inx	Increment IX		*			•	√	•
suba [ea]	Subtract from A	*		*	*	√	√	√
sbca [ea]	Subtract from A with Carry/borrow	*		*	*	√	√	√
Movement								
ldaa [ea]	Load (copy) to A from memory	*		*	*	√	√	•
ldx [ea]	Load (copy) to IX from memory	*		*	*	√	√	•
staa [ea]	Store (copy) from A to memory			*	*	√	√	•
stx [ea]	Store (copy) from IX to memory			*	*	√	√	•
Logic								
anda [ea]	Bitwise AND A with memory	*		*	*	√	√	•
com [ea]	Complement (NOT) memory			*	*	•	•	1
coma	Complement (NOT) A		*			•	•	1
oraa [ea]	Bitwise OR A with memory	*		*	*	√	√	•
lsl [ea][1]	Logic Shift Left memory one place			*	*	0	√	b_7
lsla[1]	Logic Shift Left A one place		*			0	√	b_7
lsr [ea]	Logic Shift Right memory one place			*	*	0	√	b_0
lsra	Logic Shift Right A one place		*			0	√	b_0
Testing								
cmpa [ea]	Compare A with memory	*		*	*	√	√	√
cpx [ea]	Compare IX with memory	*		*	*	√	√	•
Branch								
bra	BRAnch always					•	•	•
beq	Branch if EQual to zero					•	•	•
bne	Branch if Not Equal to zero					•	•	•
bcc/bhs	Branch if Carry Clear/Higher or Same					•	•	•
bcs/blo	Branch if Carry Set/LOwer than					•	•	•
bpl	Branch if PLus (bit 7 = 0)					•	•	•
bmi	Branch if MInus (bit 7 = 1)					•	•	•
jmp	JUmP (goto) directly			*	*	•	•	•

*	: Available	√	: Flag operates normally
1	: Flag set	•	: Not affected
[ea]	: Effective address	0	: Flag cleared
[1]	: Alternatively asl		

Table 3.1 *Our BASIC computer's instruction set.*

THis is coded as **7F-60-50***h*. The characteristic of this address mode is that the *location of the operand is fixed* as an integral part of the program, and this cannot be changed as execution progresses.[7] This address mode is sometimes called **direct**.

Program 3.1 *Clearing memory the linear way.*

```
CLEAR_ARR:   clr  6000h  ; Clear Array[0]
             clr  6001h  ; and Array[1]
             clr  6002h  ; Each clr occupies four bytes
             clr  6003h  ; of program memory
             clr  6004h  ; Keep on going
             .....  .....
             .....  .....
             clr  61FEh  ; Clear Array[510]; nearly there
             clr  61FFh  ; Clear Array[511]; Phew!
```

Although directly specifying its address may seem to be the obvious way to locate an object in memory, this technique is rather inflexible. Suppose we wished to clear an area of memory between $6000h - 61FFh$, say to hold an array of 512 byte elements `Array[0]`…`Array[511]`. The obvious way to do this is shown in Program 3.1, which uses a `clr` instruction for each byte. This program needs 512 3-byte instructions, totalling 1.5 Kbyte of program memory for storage! Although it works, this is highly inefficient, and the mind boggles if you wanted to clear a 4 Kbyte memory space! There has got to be a better way.

Indexed

The **Index register** is not generally used to hold data but instead holds a 16-bit address. This address is used to locate or *point to* a datum byte in memory. For example:

```
clr  ,x          ; Clear the byte pointed to by IX
```

The **effective address** (ea) is the contents of the Index register. Thus, if the Index register happened to hold the address $6010h$ at the time the

---

[7]If the code is stored in RAM, in theory the program can change itself, but self modifying code is a somewhat hair raising practice!

example instruction above was executed then the net result would be to copy the byte at $6010h$ into the Accumulator. Another term for this type of addressing is **indirect**, as the register does not hold the data itself, only a location pointer.

This seems rather an obscure way of doing things, but let us revisit our array clearing example. Repeating the same thing 512 times on successive memory locations is a doubtful way of doing this. Why not use a pointer into the array, and increment the pointer each time we do a Clear? This is just what we have done in Program 3.2. The linear structure of the previous program has been folded into a **loop**, shown shaded. The execution path keeps circulating around the `clr` instruction which is 'walked' through the array by advancing the pointer on each pass through loop. Eventually the pointer moves out of the desired range and the program then exits the loop.

Program 3.2 has many new features, especially as we haven't yet reviewed the instruction set.

1. Line 1 initializes the Index register by moving the *constant* $6000h$ (the location of `Array[0]`) into it. Note the use of `ldx` instruction. Nearly all loop structures involve some setting up before entry.

2. The actual Clear instruction uses the Indexed address mode. This line has a label associated with it; it is called `CLOOP`. The assembler knows this is a label and not an instruction by the appended colon.

3. Each pass around the loop involves an incrementation of the pointer. This is done here by simply adding one onto the Index register with the INcrement IX instruction.

4. Nearly all loops need a mechanism to eventually exit, otherwise it will become an endless loop. In our case this is done by comparing

---

Program 3.2 *Clearing memory using a repeating loop.*

```
            CLEAR_ARR: ldx    #6000h   ; Set up pointer to start of array
            CLOOP:     clr    ,x       ; Clear target byte pointed to by IX
                       inx             ; Advance pointer 1
                       cpx    #6200h   ; Has pointer reached 6200h?
                       bne    C_LOOP   ; IF not over the top THEN again
            NEXT:      .....  .....    ; ELSE next instruction
```

Go here if IX is not equal to #6200h

the contents of the Index register with the constant $6200h$. If they are *not* equal then the code execution path transfers back up to the beginning of the loop. This transfer uses the Branch if Not Equal (bne) instruction. Note the use of the label CLOOP in this instruction; this is why we labelled this entry point earlier. If the comparison test fails (when IX = $6200h$) then the branch back is ignored and execution passes to the next instruction after the loop.

Relative

The Branch instructions implicitly alter the state of the Program Counter. By adding the byte following the op-code the outcome is to cause the computer's execution order to skip to another part of the program. This is known as the **Relative address mode** which computes the effective address as the Program Counter plus offset. For example bra .+06 (I am using the . to symbolize the current value of the PC), which is coded as 20-06h, skips forward six places from where the PC is at following the fetch of the instruction. This is simply implemented by the computer adding 6 onto the PC at execution time. We will discuss Branch instructions shortly (see page 66), but examination of Table 3.1 shows that all but one of these instructions are conditional on the outcome of some test or action. Thus we can code actions such as:

```
IF object is zero
     THEN DO this;
     ELSE DO that;
```

Having covered the address modes, let us look briefly at the instruction set in Table 3.1. Instructions have been divided into five groups as follows.

Arithmetic

The fundamental operations of addition and subtraction are supported. Datum may be added to (adda) or subtracted from (suba) the Accumulator. Thus suba 6020h subtracts the byte out at memory location $6020h$ *from* the content of the Accumulator.

The variant adca also adds in the state of the Carry flag , 0 or 1 to the sum. In a similar manner sbca subtracts the state of **C** from the difference, in effect treating the Carry flag as a borrow from a previous subtraction. Both variations expedite multiple-precision arithmetic implemented as a

series of byte operations where a carry or borrow is propagated from
least- to most-significant columns; see example **??**.

Clearing is equivalent to copying the constant zero into the destina-
tion effective address and thus is convenient but not fundamental. `clra`
directly zeros the Accumulator, and is similar to `ldaa #0`. A read/write
memory location can be directly zeroed by using the `clr` instruction, e.g.
`clr 6020h`. This is comparable to `ldaa #0 -- staa 6020h`, which how-
ever has the side effect of destroying (overwriting) the previous content
of the Accumulator.

Incrementation is also not fundamental, but is equivalent to adding one
to the target location. Thus `inca` augments the Accumulator and `inx` the
16-bit Index register. Like `clr`, `inc` can operate directly on the contents
of read/write RAM. For example `inc 6020h` increments the byte datum
in $6020h$. This does not affect the state of the Accumulator. None of the
Incrementation instructions affect the Carry flag and thus cannot be used
for multiple-precision incrementation.

Decrementation mirrors the Incrementation instructions. Thus `deca`
subtracts one from the 8-bit Accumulator; `dex` subtracts one from the 16-
bit Index register and `dec` subtracts one from the 8-bit datum in memory
location $6020h$. None of these instructions affect the Carry/Borrow flag.

Movement

These instructions *copy* data from source to destination. The majority of
operations involve moving data, so this category is the most used of the in-
structions. Data can be copied directly from the Accumulator (**stored**) into
one memory byte location using the `staa` instruction. Similarly the state
of the Index register can be copied into read/write memory using the `stx`
instruction However, as this datum is two bytes wide, this occupies two
consecutive memory locations. For example the instruction `stx 6020h`
stores the contents of IX as a pair of bytes IXH:IXL thus $\boxed{\text{IXH} \mid \text{IXL}}$, where the
H suffix refers to the high byte and L for the low byte. See also Fig. 6.3(a)
on page 137.

Copying data from memory into the CPU is referred as **loading**. Thus
`ldaa` copies the datum from the effective address into the Accumulator.
For example, `ldaa 6020h`. In a similar manner `ldx` moves the 16 bits
from the ea:ea+1 into the Index register. For example `ldx 6020h` results
in $\boxed{\text{6020h} \mid \text{6021h}}$. See also Fig. 6.3(b) on page 137.

Logic
The `com` instruction COMplements (inverts) all bits in the target memory location, e.g. `com 6020h`. Likewise, `coma` toggles all bits in the Accumulator.

The `anda` instruction bitwise ANDs the contents of the Accumulator. For example, if the contents of A were 1001 0111*b*, then `anda #0Fh` will give **0000** 0111*b* in A (see page 13. The Accumulator can be ANDed with the contents of a memory location, e.g. `anda 6020h`, which ends up with the contents of A ANDed with (6020*h*) overwriting the original contents of A.

In a similar manner `oraa` bitwise ORs the contents of the Accumulator with either a constant or the contents of a memory location and places the outcome back in the Accumulator. Thus `oraa #10000000b` sets bit 7 of A to 1 and leaves all other bits unaffected (see page 1.3.
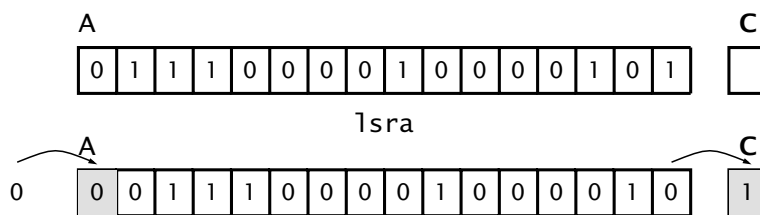
Figure 3.7 *Shifting data one place to the right.*

Two instructions are provided that can shift the contents of the Accumulator or the contents of any read/write RAM location one place either left or right, eg. `lsla` or `lsl 6020h`. As shown in Fig. 3.7, the last bit shifted out ends up in the Carry flag. The leftwise version `lsl` is often symbolized by the mnemonic `asl` (Arithmetic Shift Left), see page 121.

Testing
Mathematically the way to compare the magnitude of two numbers is to subtract them, NUM_1 − NUM_2. If they are *equal* then the outcome will be zero, that is the Z flag will be set. If NUM_1 is *higher than* NUM_2 then there will be no borrow generated, C is clear. If NUM_1 is *lower than* NUM_2 then a borrow will be generated and the C flag will be set. If NUM_1 is in the Accumulator then Table 3.2 summarizes the situation.

A *higher than* NUM: A − NUM → no Carry and non-Zero   C=0, Z=0 ($\overline{C+Z}$=1)
A *equal to*   NUM: A − NUM → gives Zero   Z=1
A *lower than* NUM: A − NUM → gives a Carry   C=1

Table 3.2 *Comparing two unsigned numbers.*

If we are only interested in the relative magnitude of two quantities, eg. "is the temperature lower than 10" and not by how much they differ, then using the `suba` instruction is overkill, in that the operand in A will be destroyed (replaced by the difference). The `cmpa` instruction uses the ALU to perform the subtraction and sets the appropriate flags but then throws away the answer (that is does not put it in A). Compare can be thought of as a non-destructive subtract.

Branch

Branch instructions make the Program Counter skip *xx* places forward or backwards; usually based on the state of the CCR flags. Thus the instruction `bcc .+8` (coded as 24-**08**h) means "Add eight to the *current* state of the PC if the C flag is clear". Notice the terminology `.+8` to mean from the current place. Remember, from Figs. 3.3(b), 3.4(b) & 3.5(b), that by the time an instruction is executed the PC is already pointing to the *next* instruction. Thus this instruction actually lands the execution point 10 (8 + 2) bytes further along from the Branch instruction. Backwards skips, eg. `bcs .-16` (coded as 25-**F0**h) use a 2's complement offset following the op-code. With a byte-sized offset following the op-code, a range of $+129$ to -126 (7Fh – 80h) bytes is possible remembering that 2 is added to the PC when the Branch instruction is fetched.

Rather than calculating these offsets by hand, use labels and allow the assembler to do the sums. Thus in Program 3.2 we have `bne CLOOP` instead of `bne .0FAh`. Eight Branches are listed. Branch Always (`bra`) is unconditional, that is the offset byte is *always* added to the PC. The `bcc:bcs` pair have the alternative mnemonic `bhs:blo` for Branch if Higher or Same and Branch if LOwer than, which is more meaningful after a Comparison operation. Branch if EQual to zero/Not Equal to zero similarly check the Z flag. Although Conditional Branches frequently come after a Compare operation (as in Program 3.2), they can follow any operation that affects the appropriate flags, such as a `ldaa` or `staa` instruction.

All Branch instructions use the Relative address mode discussed on

pages 63 and 100. The `jmp` instruction is similar to the unconditional `bra` but is absolute in that the jump is to a specified address and not *n* places from the current position. As `jmp` can be used to transfer execution to any point in the program it can be used in conjunction with any of the Condition Branch instructions to extend their range. For example:

```
        bne  NEXT    ; Continue on if Z = 0
        jmp  FRED    ; ELSE go to FRED
NEXT:   ...  ....
```

will transfer control to the far off point FRED if the `Z` flag is set to 1. Thus it is equivalent to a hypothetical `lbeq` Long Branch if EQual to zero instruction.[8]

Examples

Example 3.1
Write a program that will add the byte *contents* of memory locations $0000h$ (called NUM1) to that of $0001h$ (called NUM2). The answer is to be in $0010h$ (SUM_H) and $0011h$ (SUM_L) in the order high:low byte.

Solution
This is similar to our load-add-store program on page 50 but the second operand is a byte variable in memory rather than a constant. The three instructions to implement this are now:

Program 3.3 *Simple single-precision addition of two byte variables.*

```
SP_ADD: ldaa NUM1  ; Get the first memory byte
        adda NUM2  ; Add to it the second byte
        staa SUM_L ; Put the outcome in memory as the lower sum byte.
```

Of course this will only work if the outcome of the addition can fit into a single byte; that is no more than FFh ($255d$). If, for example,

---

[8]The 6809 MPU has a set of Long Branches that mirror the normal short Branches.

both NUM1 and NUM2 were FF*h*, then the outcome would be 1 FF*h*. As we have reserved two bytes for the sum then all we have to do is set the most-significant byte SUM_H to 1 if there is a carry out from the addition of the two variable bytes (the maximum value of SUM_H is one) otherwise zero the upper byte of the sum. One possible implementation is shown in Program 3.4. Here we simply zero the

---

Program 3.4 *A more accurate single-precision addition of two byte variables.*

```
SP_ADD: clr  SUM_H ; Prepare the upper sum byte by zeroing it

        ldaa NUM1  ; Get the first memory byte
        adda NUM2  ; Add to it the second byte
        staa SUM_L ; Put the outcome in memory as the lower sum byte.

        bcc  EXIT  ; IF zero carry THEN finished
        inc  SUM_H ; ELSE increment higher sum byte

EXIT:   ... ..... ; Next part of the program
```

---

upper byte of the sum in advance, and after the addition skip around the Increment instruction of line 6 if the Carry flag is Clear (Branch if Carry Clear, line 5).

The more general solution, shown in Program 3.5, actually adds the Carry flag state directly on to a zero constant number and then moves the outcome to the most significant byte of the sum.

Program 3.5 *An alternative single-precision addition of two byte variables.*

```
SP_ADD: ldaa NUM1  ; Get the first memory byte
        adda NUM2  ; Add to it the second byte
        staa SUM_L ; Put the outcome in memory as the lower sum byte.

        ldaa #0    ; Clear the Accumulator, but not the C flag
        adca #0    ; Add to it the carry information
        staa SUM_H ; and the outcome will be the state of the Carry flag
```

Notice that the instruction `ldaa #0` has been used to clear the Accumulator instead of the more obvious `clra`.

Example 3.2

Write a program routine that will add two 16-bit numbers giving a 17-bit sum. The augend is located in the two memory locations $0000{:}1h$ in the order high:low byte thus $\boxed{\text{AUGEND\_H}\ \text{AUGEND\_L}}$. The addend is similarly situated $\boxed{\text{ADDEND\_H}\ \text{ADDEND\_L}}$. The sum is stored as three bytes in the order high:middle:low thus $\boxed{\text{SUM\_H}\ \vert\ \text{SUM\_M}\ \vert\ \text{SUM\_L}}$.

Solution

Although BASIC is only capable of directly implementing 8-bit arithmetic, operations of any length are possible by breaking down the process into byte-sized stages. In the case of addition, this involves a sequence of byte operations from the least to the most significant digits with any carry from the nth digit byte being added into the $n + 1$th summation. The least significant addition has a presumed carry-in of 0 and the carry-out from the most significant addition becomes the highest bit of the outcome. For example FF FFh + FF FFh = 1 FF FFh $(65{,}535d + 65{,}635d = 131{,}070d)$.

The overall process is diagrammatically shown in Fig. 3.8. However, given that we need to implement the process as a sequence of steps
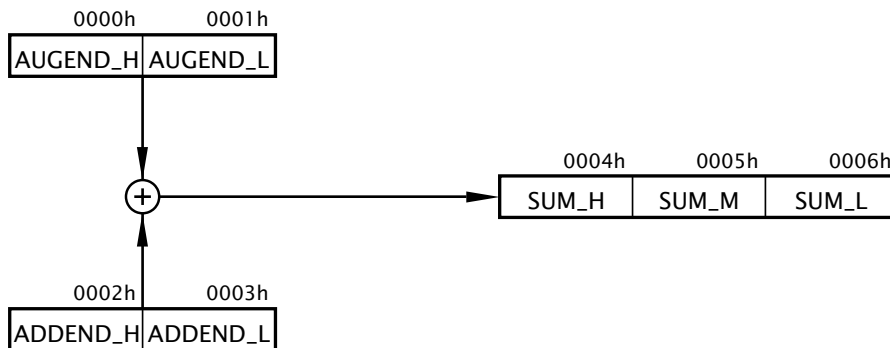
Figure 3.8 *The process.*

executable by the byte-sized instruction of Table 3.1 then the next step is to produce a task listing.

1. Add the low bytes of the augend and addend, generating the low byte of the sum and carry C1.

2. Add the high bytes of the augend and addend plus the last carry-out C1 to give the middle byte of the sum and a new carry-out C2.

3. The high byte of the sum is the last carry-out C2, either 0 or 1.
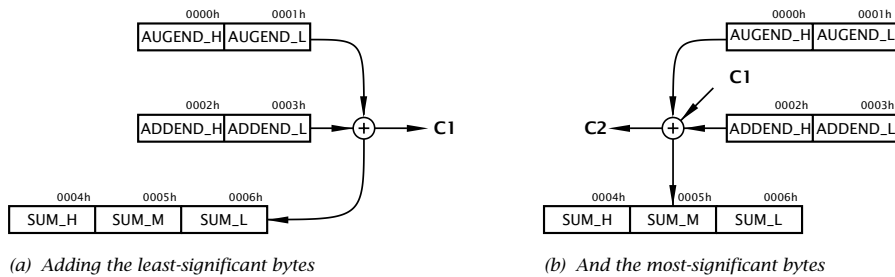
(a) Adding the least-significant bytes

(b) And the most-significant bytes
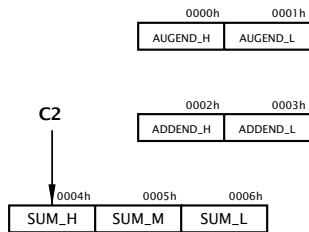
(c) The most-significant sum byte is the last carry-out

Figure 3.9 *Visualisation of the task process.*

Given that this is our first program of any substance, a detailed visualization of this task list will be useful. For most instances detail at this level is not helpful and subsequently we will use a more abstract visualization known as a flow chart (see Example 3.3).

Once a task list has been established then the next step is to implement this as a sequence of instructions, that is the program.

One possible is shown in Program 3.6.

Program 3.6 *The double-precision add program.*

```
; Task 1
D_P_ADD: ldaa AUGEND_L ; Get the lower byte of the augend
         adda ADDEND_L ; Add to the lower byte of the addend
         staa SUM_L    ; Giving the lower byte of the sum
; Task 2
         ldaa AUGEND_H ; Get the augend's higher byte
         adca ADDEND_H ; Add to the addend's like byte + carry from 1
         staa SUM_M    ; Giving the sum's middle byte
; Task 3
         ldaa #0       ; Clear Accumulator but not Carry flag
         adca #0       ; Add zero plus the Carry to the Accumulator
         staa SUM_H    ; This gives the sum's highest byte (0 or 1)
```

In the listing the three tasks are identified by an appropriate comment; each task being implemented by 3 instructions.

Task 1

This comprises a Load-Add-Store sequence, as illustrated in Figs. 3.3 — 3.5, to add the lower byte of the addend to that of the similar significant augend. The outcome byte is stored in memory at $0004h$ (SUM_L) and the Carry flag bit is set as appropriate to C_1.

Task 2

This is very similar except that the second byte of all three variables are targeted and the adca (ADd with Carry to Accumulator) replaces the plain adda of Task 1. This adds in the Carry bit C_1 as well as the two data bytes. The outcome byte is stored in memory at $0005h$ (SUM_M) and the C flag now holds C_2.

Task 3

This is slightly more complex. The objective is to make memory location $0006h$ $00h$ if the C flag is 0 and $01h$ if C is 1.

One way of doing this is to zero the Accumulator and add zero to it plus the carry C_2. The outcome ($00h$ or $01h$) is then stored as the most significant byte of the sum SUM_H. The only problem here

is zeroing the Accumulator. The obvious way instruction to use here is `clra`, but we see from Table 3.1 that this also clears the C flag! The `ldaa` instruction does not, and thus we use `ldaa #00` to zero the Accumulator and not loose C_2. `adca #0` then adds zero plus carry, which is equivalent to the instruction Add Carry Only.

Example 3.3

Write a routine that will multiply the contents of the Accumulator by ten. The outcome product is to be located in $0000{:}1h$ in the form $\boxed{\substack{0000h\\ \text{PROD\_H}}\substack{0001h\\ \text{PROD\_L}}}$. Locations $0002h$ and $0003h$ can be used for temporary storage of two bytes.

Solution

Conceptually the simplest way of implementing this function is by repetitive addition, that is to add the byte variable in the Accumulator ten times. Of course, multiplication of an 8-bit quantity will give a product needing a larger number of bits for storage. Two memory bytes will hold the maximum possible size of outcome $255 \times 10 = 2550$ ($FFh \times 0Ah = 9F6h$). The double-precision addition (see Example 3.2) has to be repeated ten times and this is best implemented as a loop (see page 62). Memory location $0002h$ can be used as a loop counter, decremented on each pass, with exit when COUNT reaches zero. $0003h$ can act as a temporary home for the multiplicand in the Accumulator.

The **flow chart** of Fig. 3.10 shows the task list to implement this algorithm. This diagrammatical representation uses a box $\boxed{}$ to indicate a process, a diamond $\Diamond$ to denote a decision and oval $\bigcirc$ to depict a termination or entry point.

The implementation of Program 3.7 follows the flow chart closely. Names for the four memory locations are shown as part of the program using the directive `.define`. More details on directives are given in Chapter 7. For now consider this as an aide mémoire for the programmer. The double-precision addition of a byte to a 2-byte variable is similar to Example 3.2. This is executed ten times, with the contents of COUNT decrementing from an initialised value of ten
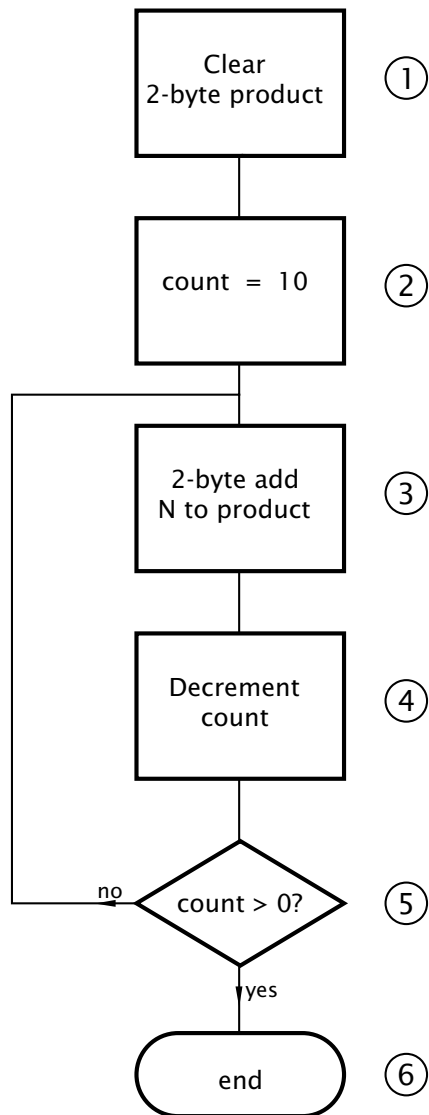
Figure 3.10 *Flow chart showing multiplication by ten*

Program 3.7 *Multiplication by ten.*

```
        .define PROD_H = 0000h,PROD_L=0001h,COUNT=0002h,TEMP=0003h
MUL_10: clr    PROD_L  ;1: Clear 2-byte product
        clr    PROD_H
        staa   TEMP    ;   Save the multiplicand in memory
        ldaa   #10     ;2: Make the fixed multiplier ten
        staa   COUNT   ;   Which becomes the loop count in memory

LOOP:   ldaa   TEMP    ;3: Get the multiplicand
        adda   PROD_L  ;3: Add it to the lower byte of the product
        staa   PROD_L  ;3: and restore in memory
        ldaa   PROD_H  ;3: Add carry to upper byte of product
        adca   #0
        staa   PROD_H

        dec    COUNT   ;4: Decrement loop count
        bne    LOOP    ;5: IF not yet zero THEN do again

        ...    .....   ;6: Next piece of program code
```

down to zero. On zero the Branch if Not Equal to zero fails and the program drops out of the loop.

This same technique could be used for any fixed multiplier between zero and 255. Could you modify the routine so that the multiplier can be a variable this range in memory on entry to the routine.

Example 3.4
The ASCII code for the character '0' (zero) is 30h. Write a program that fills an area of memory from 0000h through 01FFh with '0's.

Solution

This is similar to Program 3.2 but with the constant #30h being placed in the Accumulator *before* entering the loop (see line 2 of Program 3.8) and `clr ,x` being replaced by `staa ,x`. In Program 3.8 these two new instructions are shown shaded.
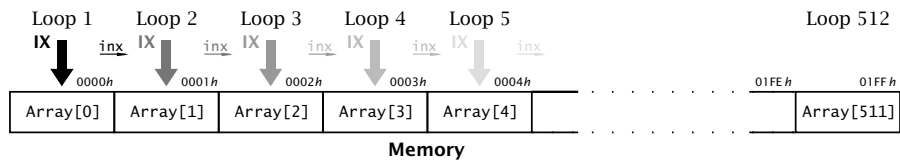
Figure 3.11 *Filling an array of memory locations with constant data.*

On each pass through the loop, the contents of the Accumulator (that is #30h) will be stored in memory at the effective address. As the Indexed address mode has been used, this effective address is simply the contents of the Index register. As this address or pointer register is initialized before entry into the loop (see line 1 of Program 3.8) and is incremented on each pass through the loop, the net effect is to progressively fill memory with the constant #30h. As before, the loop is exited when this pointer reaches $0200h$, by comparing it with the constant #0200h (line 5) and only branching back if under this value. The process is illustrated in Fig. 3.11, where the Index register is used as a pointer 'walking through' the array from $0000h$ up to $01FFh$.

Program 3.8 *Source code for the array fill program.*

```
CLEAR_ARR: ldx   #0000h  ; Set up pointer to start of array
           ldaa  #30h    ; Put the constant data in the Accumulator
CLOOP:     staa  ,x      ; Put data in memory @ location pointed to by IX
           inx           ; Advance pointer 1
           cpx   #0200h  ; Has pointer reached 0200h?
           bne   C_LOOP  ; IF not over the top THEN again
NEXT:      ..... .....   ; ELSE next instruction
```

Go here if IX is not equal to #0200h

Example 3.5
Data from an array of memory between $0000h$ and $0FFFh$ is to be transmitted byte by byte to a distant computer over the internet. In order to allow the receiver to examine the data and check for transmission errors it is proposed to append a single byte which is the 2's complement (i.e. the negative value, see page 9) of the 8-bit sum of all the data bytes together. If all the received data bytes plus

this **checksum** byte are similarly added then the outcome should be zero if no error has occurred.

Solution

This is very similar to Example 3.4 but here as we walk through the loop we add the bytes, ignoring any overflowing carries. When the grand total has been reached, it is inverted and one added (that is 2's complemented).

Program 3.9 *Generating a checksum.*

```
CHECKSUM: ldx  #0000h  ; Point to the bottom of the array
          clra          ; Clear the sum

LOOP:     adda ,x       ; Add ARRAY[n] to the sum
          inx           ; Increment n
          cpx  #7000h   ; Check. Over the top yet?
          bne  LOOP     ; IF not THEN do next add

          coma          ; ELSE invert grand total
          inca          ; and add one to give the checksum in A

          ... .....
```

Example 3.6

One simple way of encrypting a data byte is to reverse the order of bits. For example $10111100b \rightarrow 00111101b$. Write a routine to implement this reversal on a data byte in $0000h$. The encrypted outcome is to be in the Accumulator. You can use location $0001h$ as a loop counter.

Solution

Program 3.10 simply shifts left the data once and the encrypted data once right. When a left shift sets the Carry flag because the shifted out bit (see Fig. 3.7) is a 1 then the leftmost of the shifted-right encrypted byte is set to one using the `oraa` instruction (see 65).

In this way after eight passes, the encrypted data is the original reversed.

Program 3.10 *Reverse encryption.*

```
        .define DATA = 0000h, COUNT = 0001h
REVERSE_ENCRYPT:
        ldaa #8            ; Set the loop counter to 8
        staa COUNT

LOOP:   lsl  DATA          ; Shift the data left once
        bcs  SHIFT1        ; IF a 1 pops out into the C flag
                           ; skip over the next two instructions
        asra               ; Shift encrypted data once right
        bra  NEXT          ; with a 0 shifted in

SHIFT1: asra               ; Shift encrypted data once right
        oraa #100000000b   ; with a 1 shifted in

NEXT:   dec  COUNT         ; Record one more pass
        bne  LOOP          ; and repeat until counter is zero
        ... .....
```

Self-assessment questions

3.1

3.2 Write a routine to act as a receive checksum for Example 3.5. The data comes in from a location $8000h$ a byte at a time, and there are 4097 bytes, including the checksum byte. You can assume that everytime location $8000h$ is read a new value is available. If an error has occurred then the program is to jump to a label ERROR some distance away, otherwise it is to continue on to the next instruction.

3.3 Write a program that will convert a byte located in memory location $0000h$ called BINARY of value $00h$– $63h$ (0 – $99d$) to a 2-digit BCD

equivalent in locations $0010h$ for the ten's digit (called TENS) and $0011h$ for the units digit (called UNITS). For example if BINARY is $4Fh$ ($01001111b$) then the outcome will be 07 tens and 09 units; that is $4Fh = 79d$.

Hint: The easiest way to do this is to keep a tally of how many times ten can be subtracted from the binary number without the residue being less than ten. Whatever is left, the residue, will be the value of the units.

CHAPTER 4
The 6802 microprocessor

In this chapter we introduce the 6802 MPU, which we will use as our illustrative device for the rest of the text. Here we will primarily look at the internal structure, reserving external hardware considerations for later.

After reading this chapter you should:

- *Understand that the 6802 MPU has two general purpose data registers, Accumulator A and Accumulator B, both of which are interchangeable for the vast majority of instructions.*

- *Realize that the 6802 is an eight-bit device by virtue of its eight-bit ALU and Data buses.*

- *Understand that the 6802 has 2 16-bit address registers, the Index register and Stack Pointer register.*

- *Understand the function of the* **C**, **N**, **Z**, **V** *and* **H** *flags in the Code Condition register.*

What exactly is an MPU? This question is best approached from a historical perspective. In 1968, Robert Noyce (one of the inventors of the integrated circuit), Gordon Moore[1] and Andrew Grove left the Fairchild Corporation and founded their own company, which they called Intel.[2] Within three years, Intel had developed all the basic types of semiconductor memories used today — dynamic and static RAMs and EPROMs.

As a sideline Intel also designed large-scale integrated circuits to customers' specifications. In 1971 they were approached by a Japanese maker

---

[1] Moore's law stated in 1964 that the number of elements on a chip would double every 18 months, although this was subsequently revised to 2 years.

[2] Reputed to stand for INTELligence or INTegrated ELectronics.

of electronic calculators called Busicom, and asked to manufacture a suitable chip set. At that time calculators were a fast-evolving product and any LSI devices were likely to be superseded within a few years. This of course would reduce an LSI product's profitability and increase its cost. Engineer Ted Hoff — reputedly while on a topless beach in Tahiti — came up with a revolutionary way to tackle this project. Why not make a simple von Neumann CPU on silicon? This could then be programmed to implement the calculator functions, and as time progressed these could be enhanced by developing this software. Besides giving the chip a longer and more profitable life, Intel were in the business of making memories — and computer-like architectures need lots of memory. Truly a brain wave. Busicom endorsed the Intel design for its simplicity and flexibility in late 1969, rather than the conventional implementation.

Federico Faggin joined Intel in spring 1970[3] and by the end of the year had produced working samples of the first chip set. This could only be sold to Busicom, but by the middle of 1971, in return for a price reduction, Intel were given the right to sell the chip set to anyone for non-calculator purposes. Intel was dubious about the market for this device, but went ahead and advertised the 4004 "Micro-Programmable Computer on a Chip" in the *Electronic News* of November 1971. The term **MicroProcessor Unit** (**MPU**) was not coined until 1972. The 4004 created a lot of interest as a means of introducing intelligence into electronic products.

The 4004 MPU featured a four-bit Data bus, with direct addressing of 512 bytes of memory. Clocked at 108 kHz, it was implemented with a transistor count of 2300.[4] Within a year the eight-bit 200 kHz 8008 appeared, addressing 16 Kbytes and needing a 3500 transistor implementation. Four bits is satisfactory for the BCD digits used in calculators but eight bits is more appropriate for intelligent data terminals (like cash registers) which needed to handle a wide range of alphanumeric characters. The 8008 was replaced by the 8080[5] in 1974, and then the slightly modified 8085 in 1976. The 8085 is still the current Intel eight-bit device. Strangely, four-bit MPUs were to outsell all other sizes until the early 1990s.

---

[3]He was later to found Zilog.

[4]Compare with the Pentium Pro (also known as the P6 or 80686) at around 5.5 million!

[5]Designed by Masatoshi Shima, who went on to design the 8080-compatible Z80 for Zilog.

The MPU concept was such a hit that many other electronic manu-factures clambered on to the bandwagon. In addition, many designers jumped ship and set up shop on their own, such as Zilog. By 1976 there were 54 different MPUs either available or announced. One of the most successful of these was the 6800 family produced by Motorola.[6] The Motorola 6800 had a clean and flexible architecture, could be clocked at 2 MHz and address up to 64 Kbyte of memory. The 6802 (1977) even had 128 bytes of on-board memory and an internal clock oscillator. This de-vice is the subject of this text. By 1979 the improved 6809 represented the last in the line of eight-bit devices, competing mainly with the Intel 8085, Zilog Z80 and MOS Technology's 6502.

The MPU was not really devised to power conventional computers, but a small calculator company called MITS,[7] faced with bankruptcy, took a final desperate gamble in 1975 and decided to make and market a com-puter. This primitive machine, designed by Ed Roberts, was based on the 8080 MPU and interacted with the operator using front panel toggle switches and lamps — no keyboard and VDU. The Altair[8] was advertised for $500, and within a month MITS had $250,000 in the bank for advance orders.

This first **Personal Computer** (**PC**) spawned a generation of computer hackers. Thus an unknown 19-year-old Harvard computer science stu-dent, Bill Gates, and a visiting friend, Paul Allen, in December 1975 no-ticed a picture of the Altair[9] on the front cover of *Popular Electronics* and decided to write software for this primordial PC. They called Ed Robert with a bluff, telling him that they had just about finished a version of the BASIC programming language that would run on the Altair. Thus was the Microsoft Corporation born.

In a parallel development, 22 Altair owners in San Francisco set up the Home-brew club. Two members were Steve Jobs and Steve Wozniak. As a club demonstration, they built a PC which they called the Apple.[10] By

---

[6]Motorola was launched in the 1930s to manufacture motor car radios, hence the name "motor" and "ola" (as in pianola).

[7]Located next door to a massage parlor in New Mexico.

[8]After a planet in *Star Trek*.

[9]The picture was just a mock up, they actually were not yet available; an early example of computer 'vaporware'!

[10]Jobs was a fruitarian and had previously worked in an apple orchard.

1978 the Apple II made $700,000; in 1979 sales were $7 million, and then $48 million…

The Apple II was based around the low-cost 6502 MPU which was produced by a company called MOS Technology. It was designed by Chuck Peddle, who was also responsible for the 6800 MPU, and had subsequently left Motorola. The 6502 bore an uncanny resemblance the the Motorola 6800 family and indeed Motorola sued to prevent the related 6501 MPU being sold, as it even had the same pinout as the 6800. The 6502 was one of the main players in PC hardware by the end of the 1970s, being the computing engine of the BBC series and Commodore PETs amongst many others.

What really powered up Apple II sales was the VisiCalc spreadsheet package. When the business community discovered that the PC was not just a toy, but could do 'real' tasks, sales took off. The same thing happened to the IBM PC. Reluctantly introduced by IBM in 1981, the PC was powered by an Intel 8088 MPU clocked at 4.77 MHz together with 128 Kbyte of RAM, a twin 360 Kbyte disk drive and a monochrome text-only VDU. The operating system was Microsoft's PC/MS-DOS version 1.0. The spreadsheet package here was Lotus 1-2-3.

Intel had introduced the 29,000-transistor 8086 MPU in 1978 as a 16-bit version of the 8085 MPU. It was designed to be compatible with its eight-bit predecessor in both hardware and software aspects. This was wise commercially, in order to keep the 8085's extensive customer base from looking at competitor products, but technically dubious. It was such previous experience that led IBM to use the 8088 version, which had a reduced eight-bit Data bus and 20-bit Address bus[11] to save board space.

In 1979 Motorola brought out its 16-bit offering called the 68000 and its eight-bit Data bus version, the 68008 MPU. However, internally it was 32-bit, and this has provided compatibility right up to the 68060 introduced in 1995 and ColdFire RISC device launched in 1997. With a much smaller eight-bit customer base to worry about, the 68000 MPU was an entirely new design and technically much in advance of its 80X86 rivals.

The 68000 was adopted by Apple for its Macintosh series of PCs. However, the Apple Mac only accounts for less than 20% of PC sales. Motorola

---

[11]A 2^{20} address space is 1 Mbyte, and this is why for backwards compatibility MS-DOS is still limited to 1 Mbyte of conventional memory.

MPUs have been much more successful in the embedded microprocessor market, the area of smart instrumentation from egg timers to aircraft management systems. Of course, this is just the area which MPUs were developed for in the first place, and the number, if not the profile and value, of devices sold for this purpose exceeds those for computers by more than an order of magnitude. Most of these are eight-bit devices, and are single-chip microcontrollers developed from the original 6800 architecture, including the 68HC05 and 68HC11 devices, and are still very popular in applications such as automotive displays and smart cards. Thus the original 6800 MPU is still relevant.

A somewhat simplified view of the 6802 MPU[12] is shown in Fig. 4.1. If the 6802's structure has more than a passing resemblance to our make-believe computer's architecture, then this is not a coincidence and you should first read Chapter 3's discussion of the function of the various blocks.

Externally the 6802 MPU is characterised as three buses.

Data bus
Eight lines carry data both to and from memory and input/output interfaces one byte at a time. Information flow is in both directions. During a Read cycle data on these lines is taken into the MPU's Data register. During a Write cycle the contents of the MPU's Data register is placed onto the Data bus, from where it can be collected by external memory or other circuitry.

Address bus
16 lines carry information concerning the location where the state of the Data bus is to be deposited during a Write cycle or obtained from during a Read cycle. There are $2^{16} = 65,636 = 64\,K$ locations directly addressable with this size of Address bus. As each location holds a byte of data, then the maximum system memory capacity is 64 Kbytes.

Control bus
The assortment of 13 status, clock and control signals shown in Fig. 4.1 are collectively known as the Control bus. We will look at these signals

---

[12]This is virtually identical to the other members of the 6800 series: the original 6800 device which lacked the internal oscillator and RAM, and the 6808 which has no internal RAM.

Address bus

$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$

Data bus

$d_7d_6d_5d_4d_3d_2d_1d_0$

Address register

DATAR

Data register

DATAR

Program Counter

PC

Instruction register

IR

op-code

Index register X

IX

Instruction decoder

ID

ALU mode

System Stack Pointer

SSP

Arithmetic
Logic Unit

ALU

Address registers

| 1 | 1 | H | I | N | Z | V | C |

Code Condition Register (CCR)

Reset
Halt
BA
VMA
R/W̄
IRQ
NMI

Control bus

007Fh

RE

**128-byte
internal memory**

001Fh

Battery

Battery backup

0000h

RAM

MR

EXTAL

÷ 4

Freq/4

XTAL

E

Accumulator A

A

Data registers

Accumulator B

B

6802 MPU

Figure 4.1 *The 6802 structure.*

in detail in Chapter 10, here we can briefly look at a few of the more important of these.

- R/$\overline{\text{W}}$ is a status signal which is high when the MPU is doing a Read and low during a Write cycle (see Figs. 10.1 & 10.2 on pages 194 & 195).

- VMA stands for Valid Memory Access, and indicates that the pattern on the Address bus is a legitimate address.

- E is the clock signal output used to synchronize data transfers during Read and Write cycles (see Figs. 10.1 & 10.2 on pages 194 & 195). With a 4 MHz crystal across the oscillator inputs XTAL & EXTAL the E frequency is 1 MHz.

- $\overline{\text{Reset}}$ forces the contents of the top two bytes of memory (FFFE:Fh) into the Program Counter (see Fig. ?? on page ??). The programmer will have put the start address of the software in this Reset vector as part of the program.

- $\overline{\text{Halt}}$ stops the MPU at the end of a bus cycle and brings the Data and Address bus line open circuit.

- $\overline{\text{IRQ}}$ is the Interrupt ReQuest line by which an external thing can request that the MPU quits its current processing and jump to a software routine that will service the interrupting device. The $\overline{\text{NMI}}$ Non-Maskable Interrupt line is similar, but cannot be locked out by the I bit in the Code Condition register (see page 88).

Of importance to the programmer are the data and address registers. Our BASIC computer had only one 8-bit Accumulator data register. Here we now have two **Accumulator registers**, A and B, which may be used interchangeably for the vast majority of instructions. That means that virtually any instruction that can operate on Accumulator A has a counterpart than targets Accumulator B in the same manner, e.g. `clra` and `clrb`. This general-purpose property is sometimes known as orthogonality, and contrasts with the competing Intel 8080 family where most of the registers have specialized functions. An Accumulator is normally used by the ALU either as a source operand or/and to hold the outcome of an instruction. For example `adda 6000h` adds the contents of 6000h to that of the source operand in Accumulator A, putting the outcome back in Accumulator A. In dealing with either of these data registers, the programmer

must remember that data can only be handled in byte sized chunks. The size specification of an MPU is usually on the basis of the processing capacity of the ALU. The Data bus size usually, but not always, matches this size. For example the 68008 MPU has an eight-bit Data bus, 16-bit ALU and 32-bit internal registers! Objects larger than 8 bits have to be processed a byte at a time (for example see Example 3.2 on page 69).

The 6802 MPU also has two address registers; the **Index register** (**IX**) and the **Stack Pointer register** (**SP**) which are 16 bits long. Unlike the accumulators, the address registers are specialized, each having a specific task. The Index register is normally used to point to an object in memory which can be accessed using the Indexed address mode, as has been described in page 61. Motorola intended the address registers to hold pointers into memory and not to be used for other nefarious purposes. Because of this, only a rather limited repertoire of instructions can modify an address register. However, as it is possible to both increment and decrement (`inx` and `dex`) the Index register, it is sometimes used to keep a 16-bit (65,536) count, for example Program **??**. This is useful as the equivalent using an accumulator or single memory location is restricted to eight bits; a count range up to 256. The Stack Pointer is used to act as a type of book mark pointer to an area of memory, known as a stack, used for last-in first-out temporary storage — but we will leave this to Chapter 8. The `SP` is rarely used for anything other than its legitimate task.

Like the address registers, the **Program Counter** (**PC**) also holds an address; this time the location of the current instruction. It is logical that it also should be 16 bits wide. The `PC` is initialised to the start of the program on reset, as described on page **??**. Normally the `PC` automatically advances as each instruction byte is fetched from memory to the Instruction register for decoding, as shown in Figs. 3.2—3.5 in Chapter **??**. Only Jump and Branch instructions can directly modify the `PC`.

The five flags in the eight-bit **Code Condition Register** (**CCR**) provide a status report on the `ALU`'s activity. The Carry, Zero and Negative flags are standard, and are described on page 58.

Two flags are added to the complement that was available to our BASIC computer. The **V** flag is set when two numbers of the same sign (that is the MSBs are the same) are added or subtracted and give a different sign

Code Condition Register

```
┌─┬─┬─┬─┬─┬─┬─┬─┐
│1│1│H│I│N│Z│V│C│
└─┴─┴─┴─┴─┴─┴─┴─┘
     :   :   :   :   :   :   :   :
     :   :   :   :   :   :   :   `···Carry/Borrow
     :   :   :   :   :   :   `·····2's complement oVerflow
     :   :   :   :   :   `········Zero outcome
     :   :   :   :   `··········Negative  (MSBit = 1)
     :   :   :   `············Interrupt  mask
     :   :   `··············Half carry  from  bit 3 to bit 4
```
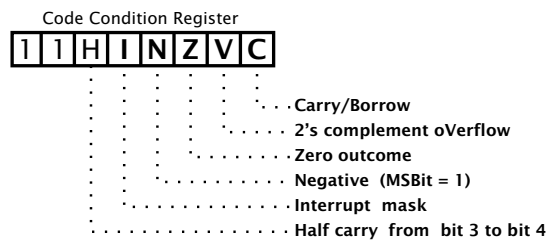
Figure 4.2 *The* Code Condition Register.

for the outcome. This oVerflow of the number into the sign position was described back on page 10.

The **H** flag needs some explanation. It is possible for the programmer to treat a byte as two 4-bit Binary Coded Decimal (BCD) digits rather than as natural binary, as described on page 5. This format is called **packed BCD**. If two such packed BCD digits are added then the Add instruction will apply the normal natural binary rules to give the outcome. For example $26 + 59 = 7F h$ $(00100110 + 01011001 = 01111111b)$. The outcome needs to be corrected to give 85 $(1000\ 0101b)$. The instruction `daa` (Decimal Adjust A) can perform the correction after an addition where the outcome is in Accumulator A. `daa` does this by following the algorithm:

1. Add six if the least significant nybble in A is over nine.
2. or add six if there has been a carry between the least most significant nybble.
3. Add six to the most significant nybble if it is then greater than nine.
4. or add six to the most significant nybble if there has been a carry out.

where six is significant in that six of the hexadecimal digits A...F are illegal in the BCD number system and have to be skipped over. Item 2 means that the instruction has to have access to carry information between bits 3 and 4. This is the **Half-carry flag**. As we will see from Table **??** on page **??** only the Addition instructions activate the H flag, so `daa` can only be used after an addition of two packed BCD bytes using Accumulator A. There is no equivalent for Accumulator B.

In summary we have:

- Carry (**C**): This flag is set if an Add operation generates a carry-out or a Subtract/Compare needs a borrow. Otherwise it resets. During a Shift operation it holds the last bit shifted out.

- oVerflow (**V**): If an arithmetic operation produces an incorrect result as seen from a signed 2's complement number perspective, this flag is set. This occurs when the addition of two positive numbers gives a negative sum or two negative numbers gives a positive outcome. Otherwise it is cleared.

- Zero (**Z**): If an operation has a zero outcome this flag is set; otherwise it is cleared.

- Negative (**N**): This shadows the most significant bit of the result of an Arithmetic or Shift operation. If the number is to be treated as a signed entity, then this may be interpreted as negative (= 1) or positive (= 0). The bpl (Branch if PLus) and bmi (Branch if MInus) instructions test this flag, see Table 6.6 on page 134.

- Half carry (**H**): This flag is set if an Add instruction generates a carry between the lower and upper nybble of the sum outcome in an accumulator. It is only activated by the Add instructions and is used by the daa instruction.

With a few exceptions the flags do not function where operations are carried out on the 16-bit address registers. The Index register does activate **Z** but Table 6.7 on page 136 should be carefully referred to when following address register manipulations by a Branch instruction.

The **I mask** is not a flag, but a *control bit*. When set to 1 an interrupt request signal at the $\overline{\text{IRQ}}$ pin will be masked out; that is ignored. **I** is set to 1 automatically when the MPU is reset and can also be set and cleared by the programmer by using the sei (SEt I) and cli (Clear I) instructions respectively (see Table ?? on page ??). Interrupts are the subject of Chapters ?? and ??.

The 6800 family programmer's model shown in Fig. 4.3 will be used for the rest of the text. This shows the disembodied registers and sizes that are available to the programmer, as well as the individual flags and mask bit in the **CCR**. This simplified vision of the processor is all that a programmer needs to know of the MPU structure in order to construct software circuits, that is programs.
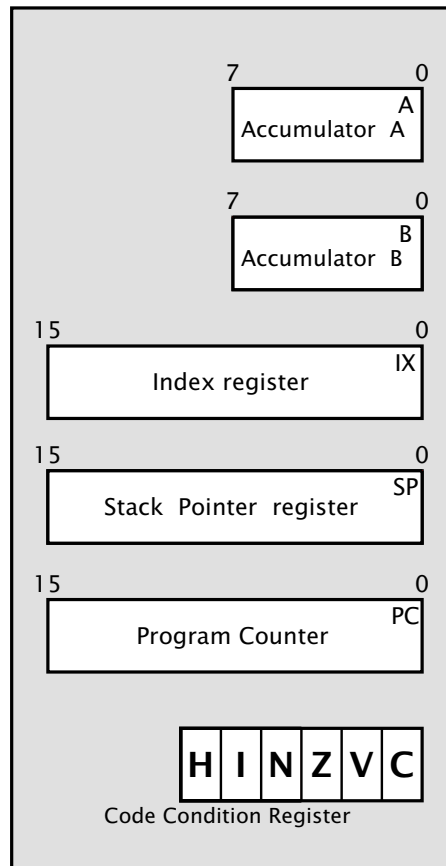
Figure 4.3 *Programmers' model for the 6800 series MPU.*

Examples

Example 4.1
Given that there is no instruction available to add the contents of Accumulator B, treated as an unsigned byte, to the 16-bit contents of the Index register, devise a routine to simulate this missing abx instruction.

Solution
The coding of Program 4.1 uses the crude method of incrementing IX while at the same time decrementing B until zero. Thus if the contents of B were 6 then IX would be incremented six times. Although this is slow, the maximum value in B is only $255d$, so the technique is acceptable. Later variants of the 6800 MPU, the 6801, 6809 and 68HC11 all introduced this missing instruction.

Program 4.1 *Simulating a abx instruction.*

```
ABX:    subb #1      ; Decrement the operand in B
        bcs  END     ; IF it falls below zero THEN finished
        inx          ; ELSE increment IX
        bra  ABX     ; and repeat
END:    ... .....
```

How could you simulate a `sbx` instruction?

Example 4.2
It is possible to multiply a binary number by first multiplying by two (shifting left once) and then adding the original number; that is $\times 3 = \times 2 + \times 1 = N << 1 + N$, where $<<$ is the Shift-Left operator (see page 11. Given that we have now a second accumulator, devise a routine to multiply the byte contents of Accumulator B by three with the outcome being represented by the 16-bit word in Accumulator A:Accumulator B. For example if B = 0FFh then the outcome will be 2FDh in A:B; that is A = 02h and B = FDh. Memory location 0000h can be used as a temporary store for the multiplicand.

Solution
The coding shown in Program **??** simply copies the multiplicand into memory for safekeeping and proceeds to shift the multiplicand once left to generate the $\times 2$ subproduct. We see from Fig. 3.7 on page 65 that this shift ejects the most significant bit into the Carry flag. As shifting an 8-bit object left will require a double-byte place holder, the precleared upper byte (which is defined to be in Accumulator A)

is incremented if this shift makes the Carry flag 1. Effectively this simulates this bit shifting into the upper byte.

Program 4.2 *Multiplying a byte by 3.*

```
        .define TEMP = 0000h

MUL_3:  stab TEMP  ; Put the multiplicand away for temporary storage
        clra       ; Zero the upper byte of the eventual product
; Generate the x2 of the multiplicand by shifting left once
        aslb       ; Shift left the lower byte of the multiplicand
        bcc  NEXT  ; IF no carry THEN leave upper byte zero
        inca       ; ELSE upper byte is 1
; Now add the x1 of the multiplicand
NEXT:   addb TEMP  ; Add the unshifted multiplicand
        adca #0    ; with carry to the upper byte giving the product
```

Example 4.3

A certain digital filter program has to generate the constant N as given by the relationship:

$$N = \cos\left(\frac{2\pi \times f}{f_s}\right)$$

where f is the center frequency and f_s is the sampling rate. Given $f = 60\,\text{Hz}$ and a sampling rate of 500 per second determine a coding to multiply an unsigned byte in Accumulator B by this constant.

Solution

We have:

$$N = \cos\left(\frac{2\pi \times 60}{500}\right) = 0.7289686$$

Now we can approximate this multiplier as a summation of power-of-two fractions:

$$N = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} = 0.7265625$$

which has an error of only around 0.3% (actually adding $\frac{1}{512} + \frac{1}{2048}$ to the series would give an accuracy to 0.004%, but as we are dealing with an eight-bit quantity this sort of accuracy doesn't accord with the resolution of around 0.5%).

Division by 2^k is simply affected by shifting right k times (see page 11). As the outcome is less than the original byte value, we can use Accumulator A as a summation as the data in Accumulator B is logically shifted right. To do this it will be useful to use the instruction aba (Add B to A), which will be discussed on page 115. The listing of Program 4.3 is then self explanatory.

Program 4.3 *Division by repetitive shift and add.*

```
DIG_FILTER: clra          ; Zero the summation

            lsrb          ; N/2
            aba           ; SUM = N(1/2)

            lsrb
            lsrb          ; N/8
            aba           ; SUM = N(1/2+1/8)

            lsrb
            lsrb          ; N/32
            aba           ; SUM = N(1/2+1/8+1/32)

            lsrb
            lsrb          ; N/128
            aba           ; SUM = N(1/2+1/8+1/32+1/128)
```

Self-assessment questions

4.1 Given that there is no Clear Index register instruction, can you deduce at least one way of zeroing an address register?

4.2 The 6800's ALU has only an eight-bit capacity. How do you think it implements operations on word-sized address registers, such as inx and what speed implication does this have?

4.3   The 6502 MPU (the computing engine of, amongst others, the BBC microcomputer, APPLE II and Commodore PET PCs) was designed by Chuck Peddel who designed the 6800 MPU and then left Motorola and set up (with others) MOS Technology (later bought over by Commodore). This popular device has one 8-bit Accumulator A and two 8-bit Index registers, IX and IY. Can you figure out some advantages and disadvantages of this architecture over the 6800 structure? In particular, what form would Indexed addressing take in this processor?

4.4   Test your knowledge by answering the following short questions without looking back.

- What is the difference between a MPU's hardware and software?
- Which three buses interconnect the MPU with memory and peripheral interface circuitry and what is their function?
- What two broad types of data are stored in memory?
- How many bits of data are stored in each memory location of a microprocessor-based circuit built with a 6800-type MPU and how many such memory locations can be directly accessed?
- Name the various registers in the programmer's model of the 6800 and briefly give their function.
- Name the various flags and mask bit in the Code Condition Register and briefly state their function.

Address modes

The majority of instructions act on data. This data may lie either in an internal register or out in memory. Thus the location of such operands must be part of the instruction. There are several different ways of specifying the **effective address** (**ea**) of an operand. The various address modes have characteristics which are advantageous in appropriate situations. In this chapter we will look at the more common of these address modes, their properties and application areas.

We have already covered most of these back in Chapter 3 on pages 59–63 in conjunction with our BASIC computer, and now would be a good time to review this material. As we will not formally look at the 6800's instruction set until the next chapter, we will use BASIC's instructions listed in Table 3.1 on page 60 for our illustrative examples.

After reading this chapter you will:

- *Know that an address mode is the way an instruction pin-points its data.*
- *Know that data can be a literal constant, located in an internal register or out in memory.*
- *Know that all instructions are represented as a one-byte operation code (op-code) followed by zero, one or two bytes representing additional information regarding the location of the operand.*
- *Know that **Inherent** instructions are completely specified by the op-code byte alone and usually have their operands explicitly in or implicitly affect internal MPU registers; for example* clra *coded as* 4Fh.
- *Know that constant operands are specified using the **Immediate** address mode, where the byte or bytes following the op-code is the literal data e.g.* ldaa #8 *coded as* 86-08h *or* ldx #1234h *coded as* CE-12-34h.

- *Know that operands that are located in fixed memory locations can be specified using an **Absolute** address mode.*

  - *Short Absolute addressing, known as **Direct** can be used for many instructions where the data lies in the first 256 bytes of memory, between* (00)00 – (00)FF*h; for example* `ldaa  80h` *coded as* 96–80h.

  - *Long Absolute addressing, known as **Extended** can be used for data located anywhere in memory, between* 0000 — FFFF*h; for example* `ldaa  6000h` *coded as* B6–60–00h.

- *Know that operands that are located in variable memory locations can be specified by using the Index register as a pointer with **Indexed** addressing; for example* `ldaa  6,x` *coded as* A6–06h.

- *Know that Branch instructions use **Relative** addressing where the trailing byte is treated as a 2's complement addend to the Program Counter, causing program execution to skip to an instruction forwards or backwards; for example* `bra  .+08` *coded as* 20–08h.

Virtually all instructions act on data; either outside the processor in its memory space, or in an internal register. Thus the op-code must include bits which inform the MPU's Instruction register where this data is being held. There are a few exceptions to this, the so called Inherent operations, such as `nop` (No OPeration) and `rts` (ReTurn from Subroutine). Single-byte instructions whose operand is a single register, for example `inca` (INCrement accumulator A), are also classified as Inherent.

With the exception of Inherent instructions, the bytes following the op-code are either the (constant) operand itself, or more usually an address of or pointer to where in memory the operand can be found. The simplest of these, where the absolute address itself follows, as in:

```
ldaa  1000h            ; [A] <- [1000] {Coded as B6-20-00h}
```

Absolute addressing is rather inflexible, as the address is fixed as part of the program, and this must be allocated by the programmer.

One of the most important features of a processor is its range of **address modes**, that is different techniques for evaluating the operand address. To see why this is important, consider, say, the problem of adding the constant 30*h* to each element of an array of 256 data bytes stored

consecutively between 1000*h* and 10FF*h*. If we had only absolute ad-dressing, the routine would look something like the listing in Table 5.1(a), which is a pity because the same action is repeated 256 times, and takes 2048 bytes of program memory.

An alternative strategy is to use an address mode where the address is stored in a register which can be incremented, and fold our program into a **loop** as shown in Table 5.1(b). This only takes 15 bytes, less than 1% of the absolute version. Furthermore, the array can be of any length without increasing the size of the program. However, there is a penalty to

---

Program 5.1 *Initializing a 256-byte array.*

```
BEGIN: ldaa  1000h ; Get array[0], 4˜
       adda  #30h  ; Add the constant (#) 30h, 2˜
       staa  1000h ; Restore it, 5˜
       ldaa  1001h ; Get array[1]
       staa  #30h  ; Add the constant 30h
       staa  1001h ; Restore it
       ldaa  1002h ; Get array[3]
        "     "    ; and so on
        "     "
        "     "
        "     "
       ldaa  10FFh ; Get array[255]
       adda  #30h  ; Add the constant 30h
END:   staa  10FFh ; Restore it (phew!)
```

(a) Linear coding.

```
BEGIN: ldx  #1000h ; Point IX to array [0], 3˜
; While address less than 1100h add 30h to the contents of that address
LOOP:  ldaa 0,X    ; Get array [IX], 5˜
       adda #30h   ; Add the constant 30h, 2˜
       staa 0,X    ; Put it away at [IX], 6˜
       inx         ; and increment pointer, 4˜
       cpx  #1100h ; Check for past array [256], 3˜
       bne  LOOP   ; and repeat if not, 4˜
END:
```

(b) Equivalent circular mode.

pay for this flexibility. The more complex address mode takes longer to execute and the loop construct has the Test and Branch overhead. Thus, the absolute array program would take 2816 (256 × 11˜) clock cycles, whilst the loop equivalent takes considerably longer at 6147 (3+256×24˜) cycles to execute. At a clock rate of 1 MHz this can be read as μs.

In the remainder of this section, we will look at the 6800 address modes. In this catalog, ☐ represents one byte and in a similar way ☐ represents two bytes.

Inherent

op-code

All the operand information is contained in the op-code, with no specific address-related bytes following. An example is `nop` (No OPeration) coded as `01h`. Motorola also classify most Register-Direct instructions as inherent, for example `inca` (INCrement A) coded as `4Ch`. Such instructions contain all the register information in the single-byte coding.

Immediate

| op-code | constant | ....8 bit (0 – 255) |

| op-code | constant | ....16 bit (0 – 65535) |

With Immediate addressing, the byte or two bytes following the op-code is treated as a *constant* datum and not an address. Some examples are:

```
adda #30h    ; Add the const 30h to Acc. B     {Coded as CB-30h}
cmpb #50     ; Compare [B] with the const 50d  {Coded as C1-32h}
ldx  #2000h  ; Put the const 2000h in IX       {Coded as 8E-20-00h}
cpx  #21FFh  ; Compare [IX] with const 21FFh   {Coded as 8C-21-FFh}
lds  #0D000h ; Set the Stack Pointer to 0D000h {Coded as 8E-D0-00h}
```

The pound (hash) symbol `#` is commonly used to indicate a constant number. The instruction `adda  30h` would be interpreted as "add the byte out in memory at 30h to Accumulator A" — a perfectly legitimate command,

but very different from what was desired. A constant can *never* be a destination, for example `stab #6` is obviously nonsense — you cannot put something in the literal 6!

Another frequent error is mismatch of size; for example `adda #500`. The literal 500 cannot fit in a byte — you cannot put a quart into a pint pot! For the two accumulators the legitimate range of literals is $00 – FFh$ $(0 – 255d)$ and for the two 16-bit address registers it is $0000 – FFFFh$ $(0 – 65,535)$.

Absolute

op-code	Page 0 address	...............................Short (Direct)

op-code	2-byte address	.......Long (Extended Direct)

In Absolute addressing, the address itself — either in whole or part — directly follows the op-code. Motorola terms the long 16-bit address version as Extended Direct. There is a short version just called Direct, where the single-byte address byte is extended to a full 16-bit address by the processor appending a zero byte. The effective address (ea) then lies in the range $(00)00 – (00)FFh$. If the address space is conceptually divided into segments or pages of 256 bytes, then this short range can be called **page 0** addressing.

Absolute addressing is used when the operand is located in a known fixed address. This might be an assigned position used to store a datum or the fixed location of a hardware port (see Chapter **??**). Some examples are:

```
ldaa 80h   ; Copy the contents of 0080h into A   {Coded as 86-80h}
stab E9h   ; Copy the contents of B out to 00E9h {Coded as D7-E9h}
ldaa 9000h ; Copy the contents of 9000h into A   {Coded as B6-90-00h}
inc  2000h ; Increment the contents of 2000h     {Coded as 7C-20-00h}
```

The short Direct form of absolute addressing is one byte shorter and takes one bus cycle less in execution time (see Figs. 10.1 & 10.2 on pages 194 & 195). Thus with instructions that have short and long extended forms of this type of address mode, it is advantageous to locate data in page 0 of memory. These first 256 bytes of memory can be thought of as being an extension of the two accumulators, in allowing relatively high-speed access to data. Unfortunately, many instructions that operate

directly on memory, such as Increment, Decrement and Clear, have only the Extended version of absolute addressing[1] (see Appendix B).

Usually the assembler that is translating the program to machine code will pick the most efficient form of absolute addressing automatically. Some assemblers allow the programmer to override this and specify the form directly.

Indexed

op-code	Unsigned offset

Rather than specify an absolute address as part of the instruction, another way of pin-pointing an object in memory is to put its address into the Index register (typically using the `ldx` instruction, as in line 1 of Program 5.1(b)) and then use the Indexed address mode to access this data. For example if you wanted to get the data at memory location $9000h$ into Accumulator B then the following code would work as an alternative to the instruction `ldab 9000h`:

```
ldx  #9000h ; Point IX to datum
ldab 0,x    ; Copy the data pin-pointed by the contents of IX
```

If you want to access a single byte then this double-barrelled approach is not very efficient. However, indirect addressing like this comes into its own when arrays or tables of data must be processed. In this situation the fact that the contents of the Index register can be altered as the program progresses, typically by incrementing and decrementing, means that large data arrays can be processed inside a loop. A comparison between Programs 5.1(a) and (b) shows the power of this technique. In essence this latter program illustrates the advantages of using a *variable* address to locate data in memory rather than a *fixed* address. The Index register is often called an **Address register** or **pointer**. Some examples are:

```
clr 0,x   ; Clear the byte located at the pointer address
adda 4,x  ; Add to A the byte located at 4 bytes above the pointer address
cmpb 55,x ; Compare the byte located at 55 bytes above the pointer address
```

---

[1]The son of 6800, the 6809 MPU, has both types of absolute addressing for all relevant instructions. Furthermore, the page 0 restriction for the short version has been relaxed in that an additional internal register, the Page register, is concatenated with the lower byte of the address to give the effective long address. Thus by altering the contents of this Page register the block of 256 direct location can be placed in any page in the memory space and altered as conditions dictate.

The 6800 MPU has only one type of Indexed address mode. All instructions using this mode have a single byte following the op-code. This is used as a positive offset to the address in IX to create the effective address of [IX] + #offset. The offset is internally extended to a 16-bit value ranging from (00)00h– (00)FFh (0 – 255). Negative offsets are not implemented. Indexed addressing, often called Address register indirect, is such a powerful technique that newer devices have many variants of this type of address mode. For example the 6809 MPU has 24 Indexed modes and four Index registers!

Relative

op-code	signed offset

This address mode is reserved in the 6800 MPU exclusively for the Branch instructions.

A Branch instruction implicitly alters the smooth flow of the program by adding or subtracting a fixed offset byte to/from the Program Counter. Effectively this causes the program to skip to another instruction either in advance or behind the instruction that would normally follow next. All such instructions are coded as an op-code byte followed by a single offset byte which is treated as a 2's complement signed constant in order to calculate the new value of the PC which will overwrite the existing value. In implementing a skip, this offset is sign extended before addition. Effectively this means that offsets between 80h and FFh are treated as negative. For example the instruction bra .-06 (where the notation .nn means nn bytes from the current position), coded as 20-FAh (FAh is the 2's complement of 06h), when the PC is at C108h is implemented as:

```
   1100  0001  0000  1000        [PC] = C108h
+  1111  1111  1111  1010        Offset = FFFAh = −6
1  1110  0001  0000  0010        [PC] = C102h, which is C108h − 0006h
```

If calculating this by hand, it must be remembered that the PC is *already pointing to the next instruction*; thus the maximum forward point is (00)7Fh + 2 = 127 + 2 = 129 bytes from the Branch instruction's op-code and (FF)80h + 2 = −128 + 2 = 126 bytes back. Fortunately it is rarely necessary to figure out the offset by hand. The assembler translator will allow you to use a label at the destination instruction. Thus line 6 in Program 5.2 is bra LOOP which is rather more readable and less error prone

than the equivalent `bra .0F8h` (`bra .-8`).

Examples

Example 5.1
A table of data has been preloaded into memory (see Program 7.4 on page 162) between $1000h$ and $100Fh$. Write a routine that will extract the nth byte into Accumulator B where n is a number from 0 – 15 in Accumulator A.

Solution
As n is a relatively small number, the easiest way to approach this is:

1. Point the Index register to the first element of the array.

2. WHILE n > zero DO

    • Decrement n.
    • Increment the Index register.

3. Use the Index register as a pointer to load the datum into Accumulator B.

Program 5.2 *Extracting the nth element of a table.*

```
GET_IT:   ldx  #1000h   ; Set IX to point to the datum
LOOP:     cmpa #0       ; Is n zero?
          bne  CONTINUE ; IF it is THEN continue
          inx           ; ELSE advance the pointer
          deca          ; and reduce n
          bra  LOOP     ; and do again

CONTINUE: ldab 0,x      ; Get the datum at 100nh
```

Actually most MPUs permit a data register to be used as the offset rather than just a fixed offset. For example in the 6809 MPU an accumulator can act as an offset. In this case the program reduces to:

```
GET_IT:    ldx  #1000h    ; Set IX to point to the datum

           ldab a,x       ; Get the datum at 100nh
```

which shows the power of using a variety of Index address modes.

Example 5.2

The temperature of a biological system is sampled on a regular basis by a 6800-based system and sequentially stored in memory between $1000h$ and $2FFFh$. When the experiment is over the 8192 byte samples in this memory array is to be transmitted as a time series to a Personal Computer (PC) byte by byte through the serial port. For the purposes of this example it may be assumed that the serial link appears as a memory location at $8020h$. When the transmission is over a check byte is to be sent as a key that can be used the receiver to verify the integrity of the transmission (see also SAE 3.**??**). This verification byte is to be a **checksum** of all 8192 bytes added together modulo-256 (2^8). That is all bytes are to be added together with any carries out being ignored. The PC will also sum incoming data bytes and if everything is correct should get the same value of checksum. When this local checksum is subtracted from the transmitted checksum the outcome should be zero unless transmission errors have occurred.

Your task is to transmit the data whilst calculating the checksum on an on-going basis and finally transmit this checksum to the PC.

Solution

The straightforward way of doing this, as shown in Program 5.3, simply copies each byte at its absolute address down into Accumulator A and then out to the serial port at $8020h$. As this is done the memory byte is added to the initially cleared Accumulator B. At the end of the process the checksum will be in Accumulator B.

Although this works, there will be a total of $8192 \times 3 + 2 = 24,578$ instructions in the program. Furthermore, taking each instruction coded as 3 bytes, over 72,000 bytes of program storage will be required. However, there is only 65,636 (64 K) bytes available!

Program 5.3 *Generating a checksum.*

```
UPLOAD: clrb        ; Zero the checksum byte

        ldaa 1000h  ; Copy ARRAY[0] into MPU
        staa 8020h  ; Send it out to the serial port
        addb 1000h  ; Also add it to the checksum byte

        ldaa 1001h  ; Copy ARRAY[1] into MPU
        staa 8020h  ; Send it out to the serial port
        addb 1001h  ; Also add it to the checksum byte


        ldaa 1002h  ; Copy ARRAY[2] into MPU
        staa 8020h  ; Send it out to the serial port
        addb 1002h  ; Also add it to the checksum byte

        ldaa 1003h  ; Copy ARRAY[3] into MPU
        staa 8020h  ; Send it out to the serial port
        addb 1003h  ; Also add it to the checksum byte

        ...  .....  ; Continue on for each array byte
        ...  .....
        ...  .....
        ...  .....


        ldaa 2FFFh  ; Copy ARRAY[8091] into MPU
        staa 8020h  ; Send it out to the serial port
        addb 2FFFh  ; Also add it to the checksum byte
                    ; Phew!!!!
        stab 8020h  ; Finally send the checksum out the serial port
```

As we are doing the same thing 8192 times, with the only difference being the sequentially advancing address, this is obviously a candidate for the use of a loop structure. Using this loop, as in Program 5.4, the Index register is utilized as a pointer to ARRAY[n], effectively simulating the array index n. Initializing it to the location

of ARRAY[0] at the beginning, incrementing it on each pass through the loop and inspecting it for n reaching 8192 gives the process coded in the program. A task list based on this structure would be:

1.Clear checksum.

2.n = 0.

3.WHILE n < 8192 DO

•Copy ARRAY[n] to the serial port.

•Add ARRAY[n] onto checksum modulo-256

•Increment n.

4.Copy checksum to serial port.

Program 5.4 *Generating a checksum in a loop.*

```
UPLOAD: clrb       ; Task1: Zero the checksum byte

        ldx  #1000h ; Task2: Point to ARRAY[0]

; Task 3
LOOP:   ldaa 0,x    ; Copy ARRAY[n] into the MPU
        staa 8020h  ; Send it out to the serial port
        adda 0,x    ; Also add it to the checksum byte

        inx         ; Advance pointer (n++)
        cpx  #3000h ; Over the top yet?
        bne  LOOP   ; IF not THEN again

        stab 8020h  ; Task 4: Send the checksum out the serial port
```

The coding follows the task list closely. The total length is nine instructions, requiring 20 bytes of program storage. Although it is dramatically shorter than the linear equivalent, it does take rather longer to execute. This is because the increment pointer and loop test instructions supporting the loop are executed 8192 times. Also the Indexed address mode takes longer to execute than the Extended equivalent. For example from Appendix B ldaa 1000h takes 4 cycles to implement and ldx 0,x takes 5 cycles — shown as 4˜ and 5˜

respectively. At a clock rate of 1 MHz this translates to 4 μs and 5 μs respectively. From this instruction set the execution time is 229.386 ms against the linear equivalent of 106.503 ms. In practice the relatively slow time of transmission over the serial link would make this processing time irrelevant.

Example 5.3
The 6809 MPU has a set of long Branch instructions that mirror the normal instruction which have a skip range of only +129 through −126 bytes and can hop to anywhere in program memory. Show how you might construct a routine that simulates the long Branch `lbeq` FRED, where FRED is outside the normal short skip range.

Solution
The only instruction that alters the state of the Program Counter is the Jump instruction. Thus if we use a normal conditional Branch instruction to hop over a `jmp` FRED instruction if the outcome is not equal to zero thus:

```
; Test of data
      bne  NEXT    ; IF Not Equal to zero THEN do not go to FRED
      jmp  FRED    ; ELSE go to FRED
NEXT: ...  ....
      ...  ....
      ...  ....


      ...  ....
      ...  ....
FRED: ...  ....    ; A long way away!
```

Actually the assembler used in this test has the ability to generate this code as a type of macro if it can determine that the destination if outside the short range. If the programmer uses suffixes the Branch mnemonic by j, e.g. `jbeq` (with the exception of `jbr` instead of `jbra`) then this conversion will be done automatically. However, features like this are at the whim of the producer of the assembler software.

Example 5.4
In Program 5.2 in Example 5.1 evaluate the offset bytes for the two Branch instructions.

Solution

1. The instruction `bne    CONTINUE` is indicating that a skip of four bytes is necessary to move the PC down from pointing to the `inx` instruction down to the `ldab    0,x` instruction. Counting from the instruction following `bne` gives $1 + 1 + 2$ bytes to advance to the instruction labelled CONTINUE. Given the op-code for `bne` is 26h then the instruction is coded as 26-04h.

2. The instruction `bra LOOP` is indicating a backwards skip from the following instruction to the instruction labelled LOOP. Counting back from `ldab  0,x` gives $2 + 1 + 1 + 2 + 2 = -8$ bytes. To calculate the offset byte we have to figure out the 2's complement of 08h or 00001000b. From page 9 we need to invert and add one:

$$\overline{00001000} \Longrightarrow 11110111; +1 = 11111000 = \text{F8}h$$

With the op-code for `bra` of 20h we have a coding of 20-F8h for this instruction.

With the help of Appendix B the machine code for the program becomes:

```
CE–10–00
81–00
26–04
08
4A
20–F8
E6–00
```

Self-assessment questions

5.1 Using the Instruction set of Appendix B determine the machine coding for Program 5.4 in Example 5.2.

5.2 A electrocardiogram (ECG) signal is sampled 256 times and the digitized values stored in memory page 1000–10FFh. Design a program using Indexed addressing to scan through this data looking for the maximum value. This value is to be in Accumulator B at the end of the routine.

5.3 Interfacing digital electronics to the analog world invariably introduces noise into the signal, even if there was none there before. One of the simplest filtering algorithms to enhance the signal to noise ratio is digital smoothing. This technique involves generating a composite value in which each point is replaced by an average of itself with its nearest neighbours; i.e. post samples. This is expressed by the formula:

$$F_t = (0.25)A_{t-2} + (0.5)A_{t-1} + (0.25)A_t$$

Assuming that the byte representing the reading two samples ago, A_{t-2}, is in memory at 0080h, the last sample A_{t-1} in 0081h and the current sample A_t in 0082h code a routine using the *Indexed address mode* that produces the smoothed sample F_t in Accumulator B. You can easily divide by powers of two to give $\frac{1}{2}$ and $\frac{1}{4}$ by shifting the data in situe, as described on page 11, before adding together.

5.4 Repeat Example 3.2 on page 71 to add 2 double-byte numbers out in memory to give a 17-bit outcome. This time use Indexed addressing to pin-point the data bytes. Compare the length and execution time of your outcome as against the Extended addressing shown in the solution Program 3.6.

5.5 Assuming that the data array of Fig. 3.8 on page 69 is moved to memory at 0000–0006h redo the coding of SAQ 5.4 but using Direct addressing. In this case evaluate the length of the coding

and execution time and compare with those of the Extended and Indexed versions. What potential disadvantage is there in using page 0 memory to locate arrays of data?

CHAPTER 6

The instruction set

If you like to think of writing a program as analogous to preparing an elaborate meal, then the address modes discussed in the last chapter are the various ingredients available to the cook. For any given cooking appliance, such as a microwave oven or electric stove (the hardware) there are a range of processes: steaming, frying, boiling etc. Each process will be listed with properties in the appliance's manual, and in our frame of reference translate to the **instruction set**.

The 6800 MPU uses 197 of the possible 256 combinations provided for with a one-byte op-code. If we factor out the various address modes this gives 72 unique instructions. Many of these are variations on the same theme (for example Load A, Load B, Load IX, Load SP) or are rarely used. Up to the moment we have survived quite well on the diet of 34 instructions listed in Table 3.1 on page 60. As these instructions are directly usable for our MPU, now would be a good time to review this material. Here we will look at the majority of 6800 instructions, but will prospone detailed discussion of instructions mainly associated with subroutines and interrupts to Chapters 8 and **??**. A convenient table of commonly used instructions is given at the end of this chapter in Table 6.10 and a full instruction set with op-codes, instruction lengths and execution times in Appendix B.

After reading this chapter you will:

- *Know that Movement instructions, copying data in-between registers and memory, are the most used and flexible of the instruction categories.*

- *Appreciate that the processor can directly implement the common arithmetic operations of Addition, Subtraction, Incrementation and Decrementation.*

- *Know that data can be shifted logically or arithmetically, or rotated*

through the C flag, either directly in memory or in an accumulator register.

- *Understand how to use the four basic logic instructions to invert, set, clear, toggle, bit test and differentiate data.*

- *Know how to compare or test signed or unsigned data for differences and relative magnitude, and take appropriate action.*

- *Recognize the various different Conditional Branches, and especially the different usage depending on whether the compared data is signed or unsigned.*

- *Know how to explicitly alter the state of the various flags/mask bits in the CCR.*

- *Understand the meaning of the term read–modify–write instruction.*

The 6800's instruction set can conveniently be divided into nine groups as follows.

Movement instructions

Around one in three instructions move data around without alteration in between registers and memory.[1] With this in mind the instructions in Table 6.1 will be the most used in the repertoire.

The Load and Store instructions copy data in-between memory and register.

- The operation of copying out to memory is known as **storing** and here the contents of the destination memory byte becomes an image of the accumulator. The `staa` and `stab` instructions are used for Accumulator A and B respectively. Note that the original data is unaltered by this process.

- The converse operation of copying data from a memory byte into one of the accumulators is known as **loading**. The two instructions of relevance here are `ldaa` and `ldab` . Again the original data remains unaltered.

It is possible to copy double bytes between the two address registers and memory, and the pertinent instructions here are given in Table 6.7.

---

[1]A straw poll of the programs in the last chapter produced a figure of around 52%.

Table 6.1 *Move instructions.*

Operation		Mnemonic	Flags				Description
			V	N	Z	C	
Load							Moves memory data to register
	to A	ldaa	0	√	√	•	[A]<-[M]
	to B	ldab	0	√	√	•	[B]<-[M]
Push							Moves Accumulator onto Stack
	A to stack	psha	•	•	•	•	[A] onto stack; SP--
	B to stack	pshb	•	•	•	•	[B] onto stack; SP--
Pull							Moves stack data to Accumulator
	from stack to A	pula	•	•	•	•	From stack to (A); ++SP
	from stack to B	pulb	•	•	•	•	From stack to (B); ++SP
Store							Moves data from register to memory
	from A	staa	0	√	√	•	[M]<-[A]
	from B	stab	0	√	√	•	[M]<-[B]
Transfer							Copy between accumulators
	A $\longrightarrow$ B	tab	0	√	√	•	[B]<-[A]
	B $\longrightarrow$ A	tba	0	√	√	•	[A]<-[B]

0   Flag always reset
1   Flag always set
•   Flag not affected
√   Flag operates in the normal way
++   Incremented automatically before use
--   Decremented automatically after use
[ ] Contents of

Data may be replicated from one accumulator to another by using the two **Transfer** instructions tab and tba (see Program 6.8). Thus if [A] were 55h, then tab would result in both accumulators holding the datum 55h.

The **Push** and **Pull** instructions need some clarification. In virtually all programs of any consequence the programmer sets aside an area of memory used for temporary storage known as a **stack**. As can be seen in Fig. 6.1 this can be visualized as a notebook whose leaves each represent a memory location and with a bookmark pointing to the currently open
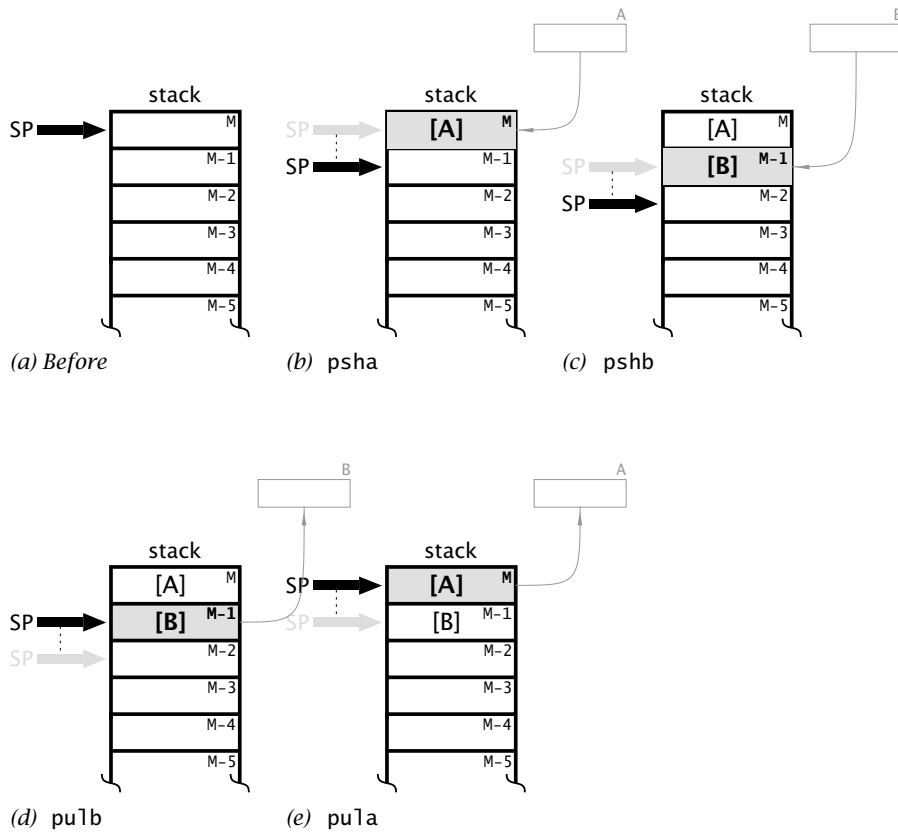
Figure 6.1 *Pushing and pulling with the stack.*

blank page known as a **Stack Pointer**. If the programmer wishes to save the contents of, say, Accumulator A, the instruction psha will copy the contents of A into the memory location 'pointed to' by the Stack Pointer address register, which is then *automatically* decremented (SP--) to point to the next empty location (equivalent to turning the page in the notebook and moving the marker before closing). In a similar manner the contents of Accumulator B can be pushed onto the stack by using pshb. Each

time a datum is pushed out to the stack the Stack Pointer automatically decrements to point to the next available leaf. The situation after two pushes is shown in Fig. 6.1(c).

In a comparable way data may be retrieved in a last-in first-out (LIFO) fashion using the two Pull[2] instructions. The effect of pulling out the last byte into Accumulator B and then the penultimate byte into A is shown in Figs. 6.1(d) and (e). Notice how the Stack Pointer is automatically preincremented (++SP) each time a datum is retrieved. This is equivalent to opening the notebook at the marked page and turning back one page with the marker to get at the last note.

The LIFO nature of this push-down stack must be remembered when pulling out data. Thus the following code fragment:

```
    psha    ; Push out [A]
    pshb    ; and [B]
    ...

; Later
    pula    ; Pull out data into [A]
    pulb    ; and [B]
```

will end up exchanging the contents of A and B, i.e. [A] $\longleftrightarrow$ [B]. Of course this may be what you wanted, but on the other hand...[3] Normally you would pull out data in the **opposite** sequence to that pushed in.

Pushing and pulling is a useful and efficient (single-byte inherent instruction) way of temporarily saving the contents of an accumulator in memory. We will see in Chapters **??** and **??** that the stack plays a vital role in supporting subroutines and interrupt handling. Setting up an area of memory to be a stack is normally done once at the beginning of the program by simply loading in the address of the top location of the stack into the Stack Pointer. From Table 6.7 the Load Stack Pointer instruction is available to do this. Thus, for example to set up a stack with its top at AFFF*h* is simply a matter of executing a `lds #0AFFFh` instruction!

Arithmetic instructions
The 6800 implements the normal Add and Subtract instructions, with and

---

[2]Some microprocessor manufacturers call this operation popping.

[3]The 6809 MPU does in fact have the instruction `exg A,B` which does the same thing but does not pester the stack.

without carry, as discussed on page 63, to add or subtract constants or memory contents to/from any of the accumulators. In addition the instruction `aba` can be used to add the contents of the two accumulators together, with the outcome being in Accumulator A. Similarly `sba` (Subtract B from A) generates the difference between A and B with the outcome being in the former.

As an example, consider the problem of dividing the unsigned contents of Accumulator A (the dividend) by that in Accumulator B (the divisor). The outcome quotient is to be located in B with the remainder in A. The simplest way of doing this is to continually subtract the divisor from the dividend, keeping a count until a borrow is generated. This count is the quotient. The remainder can be found by adding the divisor back to the residue in A, left after the last subtraction. This compensates for that last underflowing subtraction. A possible implementation is given in Program 6.1.

Program 6.1 *Division by repetitive subtraction.*

```
; Reserve a byte in memory for the quotient
          .define QUOTIENT = 0000h
DIVISION: clr  QUOTIENT ; Zero the subtract count

D_LOOP:   sba            ; Subtract divisor from dividend
          bcs  NEXT     ; IF borrow/carry THEN finished
          inc  QUOTIENT ; ELSE record one more successful sub
          bra  D_LOOP   ; and repeat

NEXT:     aba            ; Restore last subtract gives remainder
          ldab QUOTIENT ; Quotient in B
          ... .....
```

Data in memory can be incremented or decremented apparently in situ, as can the contents of either accumulator (or indeed an address register). This is especially useful in counting passes through a loop, as in Program 6.1 where QUOTIENT is located in memory at $0000h$. However, `inc` is not quite the same as `add #1` in that it does not alter the state of the Carry flag. Thus if you wanted to increment a 32-bit number in memory

Table 6.2 *Arithmetic operations*

Operation	Mnemonic	H	V	N	Z	C	Description
Add							Binary addition
to A	adda	√	√	√	√	√	[A]<-[A]+[M]
to B	addb	√	√	√	√	√	[B]<-[B]+[M]
B to A	aba	√	√	√	√	√	[A]<-[B]+[A]
Add with Carry							Includes carry
to A	adca	√	√	√	√	√	[A]<-[A]+[M]+C
to B	adcb	√	√	√	√	√	[B]<-[B]+[M]+C
Clear							Destination contents zeroed
memory	clr	•	0	0	1	0	[M]<-#00
A	clra	•	0	0	1	0	[A]<-#00
B	clrb	•	0	0	1	0	[B]<-#00
Decimal Adjust A							Correct a binary addition of BCD bytes
A	daa	•	√	√	√	√	See text
Decrement							Subtract one, produce no carry
memory	dec	•	[1]	√	√	•	[M]<-[M]−#1
A	deca	•	[1]	√	√	•	[A]<-[A]−#1
B	decb	•	[1]	√	√	•	[B]<-[B]−#1
Increment							Add one, produce no carry
memory	inc	•	[2]	√	√	•	[M]<-[M]+#1
A	inca	•	[2]	√	√	•	[A]<-[A]+#1
B	incb	•	[2]	√	√	•	[B]<-[B]+#1
Negate							2's complement
memory	neg	•	[3]	√	√	[4]	[M]<- −[M]
A	nega	•	[3]	√	√	[4]	[A]<- −[A]
B	negb	•	[3]	√	√	[4]	[B]<- −[B]
Subtract							Binary subtraction
from A	suba	•	√	√	√	√	[A]<-[A]−[M]
from B	subb	•	√	√	√	√	[B]<-[B]−[M]
B from A	sba	•	√	√	√	√	[A]<-[A]−[B]
Subtract with Carry							Included carry (borrow)
from A	sbca	•	√	√	√	√	[A]<-[A]−[M]+C
from B	sbcb	•	√	√	√	√	[B]<-[B]−[M]+C

Note 1: Overflow set when passes from *1*0000000 to *0*1111111, i.e. a seeming sign change.
Note 2: Overflow set when passes from *0*1111111 to *1*0000000, i.e. a seeming sign change.
Note 3: Overflow set if original data is 10000000 (−128), as there is no +128.
Note 4: Carry set if original data is 00000000; for multiple-byte negation.

at $0000{:}1{:}2{:}3h$ then this is how you could do it:

```
QP_INC: ldaa  0000h  ; Get the LSByte
        adda  #1      ; Add one and generate a carry
        staa  0000h  ; Put it back

        ldaa  0001h  ; Get the next byte
        adca  #0      ; Add the Carry plus 0
        staa  0001h  ; and put it back

        ldaa  0002h  ; Get the next byte
        adca  #0      ; Add the Carry
        staa  0002h  ; and put it back

        ldaa  0003h  ; Get the MSbyte
        adca  #0      ; Add the Carry
        staa  0003h  ; and put it back
```

Replacing lines 1–3 by `inc 0000h` will not work as no carry to higher bytes is generated. In a similar manner `dec` is not quite the same as `sub #1`. Can you think of an alternative way using the Zero flag?

The `neg` instructions enable the programmer to 2's complement a datum directly out in memory or in an accumulator. Remember from page 9 that this is equivalent to logic inversion plus one. This is used to convert between positive and negative when the datum is treated as a 2's complement signed number.

Instructions like `inc`, `dec` and `neg` that appear to the programmer to be carried out directly in memory are in fact implemented by loading into a temporary register in the MPU (invisible to the programmer), processing and sending back out again. This category of instructions are known as **read–modify–write** instructions. Actually the Clear memory instruction is also read–modify–write, even though the original datum is irrelevant.

An 8-bit byte may be used to hold two 4-bit Binary Coded Decimal (BCD) digits, such as 1001 1001 for decimal 99. Keeping two BCD digits in each byte is sometimes known as **packed BCD**. The addition of BCD bytes (see page 5) using common binary rules, as applied by the MPU's `add` instructions, requires a correction process if the output is to be in BCD

form. For example 0000 0101 + 0000 0111 (05 + 07) gives 0000 1100 after a normal add instruction, but should give 0001 0010 (12) if the outcome is to be in BCD form. Similarly, 0000 1001 + 0000 1001 (09 + 09) gives 0001 0010 rather than 0001 1000 (18). From these examples it can be seen that whenever the sum of two BCD digits exceeds nine, a correction must be made by adding six. This compensates for the six illegal BCD combinations (i.e. 1010 → 1111), which must be skipped over. Thus a normal addition of two packed BCD digits must be followed by the correction algorithm:

1. Commence with the least significant digit.

2. Add the two digits using a normal binary addition.

3. Examine the outcome.
   - IF the resultant 4-bit nybble is greater than nine, then add six.
   - ELSE IF there was a carry from bit 3 then add six

4. Now add the two most significant BCD digits.

5. Examine the outcome.
   - IF the resultant 4-bit nybble is greater than nine, then add six.
   - ELSE IF there was a carry from bit 7 then add six.

Consider the following addition:

0001 0010 + 1001 0011   (12 + 93)

The following steps implement this as a BCD summation but using binary additions:

```
1:   0010 + 0011 =   0101    ; adding LS decades, no correction
2:   0001 + 1001 =   1010    ; adding MS decades
3:   1010 + 0110 = 1 0000    ; correcting by addition of six.
```

Answer   1 0000 0101 (105).

The daa instruction will apply this correction to an addition of two packed BCD numbers where Accumulator A holds one of the operands. The instruction depends on information on the carry from the lower nybble to the higher nybble (from bit 3 to bit 4). This is known as a **Half carry**.[4] Thus it only works with instructions that activate the **H** flag. From

---

[4]Some microprocessors call this the Digit Carry.

Table 6.2 we see that only the addition instructions have any effect on **H**. However, daa will work after inca, as a BCD digit will never be greater than 1010*b* (9) before augmentation so there will never be a half carry. From Appendix B it can be seen that these instructions are the only ones in the complete repertoire of 6800 instructions that alter this flag, so it not shown in any other table in this chapter. daa is one of a very few instructions that has no counterpart for Accumulator B.

A frequent mistake is to assume that daa will convert a natural binary pattern in A to the equivalent BCD value. daa can only give a sensible outcome if the original two datum bytes are *already* in packed BCD form.

A simple routine using this instruction to convert a binary byte in B to a packed BCD nybble in A, simply continually decrements the binary number whilst incrementing with a BCD correction.

```
BIN_2_BCD: clra       ; Zero the BCD outcome to 00
LOOP:      decb       ; Loose one from down count in binary
           beq  EXIT ; IF down to zero THEN finished
           inca       ; Gain one up count
           daa        ; Correct to BCD format
           bra  LOOP ; and repeat
EXIT:      ... .....
```

Logic instructions

All four basic logic operations are provided, as shown in Table 6.3.

com, coma and comb inverts (or 1's COMplements) all bits in either a memory location or an Accumulator register. For example:

10001110 A → coma → 01110001 A

The anda and andb instructions bitwise AND the source in the appropriate Accumulator with the destination operand in memory or with a constant. ANDing an input with a 0 *always* gives a 0 output, whilst with a 1 does not change the logic value. For example:

10001110 A → anda #0Fh → **0000**1110 A

which clears the upper nybble of Accumulator A.

ANDing is normally used to *clear* any bit or bits in the destination operand. Thus andb #00000011b clears the upper 6 bits in Accumulator B and leaves the lower two bits untouched.

Another use of ANDing is to check the state of any bit or bits in a datum; for example:

```
anda  #0100b  ; Check bit 2 of A
beq   FRED    ; IF EQual to zero THEN go to FRED
```

By ANDing Accumulator A with 00000*1*00*b*, the outcome will be either all zero or not if bit 2 of A is 0 or 1 respectively. In the former case, the **Z** flag will be set and the following Branch if EQual to zero will be taken. Similarly, a Branch may be executed if a *group* of bits are all zero; for example, `andb #0111b` will cause the **Z** flag to be set only if bits 2, 1 and 0 are *all* zero.

The `oraa` and `orab` instructions work in the same way as for `and`. ORing with a 0 leaves the source bit unchanged whereas ORing with a 1 sets the bit to a 1 irrespective. Thus ORing is normally used to set any bit or bits in the destination operand. For example:

The `eora` and `eorb` instructions provide for the Exclusive-OR operation. You will recall from page 15 that EORing with a 0 leaves a data bit

Table 6.3 *Logic instructions.*

Operation		Mnemonic	V	N	Z	C	Description
AND							Logic bitwise AND
	A	anda	0	√	√	•	$[A] \leftarrow [A] \cdot [M]$
	B	andb	0	√	√	•	$[B] \leftarrow [B] \cdot [M]$
Complement							Invert or NOT (1's complement)
	memory	com	0	√	√	1	$[M] \leftarrow [\overline{M}]$
	A	coma	0	√	√	1	$[A] \leftarrow [\overline{A}]$
	B	comb	0	√	√	1	$[B] \leftarrow [\overline{B}]$
Exclusive-OR							Logic bitwise Exclusive-OR
	A	eora	0	√	√	•	$[A] \leftarrow [A] \oplus [M]$
	B	eorb	0	√	√	•	$[B] \leftarrow [B] \oplus [M]$
OR (inclusive)							Logic bitwise Inclusive-OR
	A	oraa	0	√	√	•	$[A] \leftarrow [A] + [M]$
	B	orab	0	√	√	•	$[B] \leftarrow [B] + [M]$

unchanged, whilst EORing with a 1 inverts (or toggles) that bit. Thus, for example if we wished to invert both bits 0 and 7 of A:

$$\boxed{10001110}_A \quad \text{eora \#81h} \quad \boxed{0000111\mathit{1}}_A$$

Another use for EOR is to isolate changes between two bit patterns. From the truth table on page 14 we see that only when the two input bits *differ* is the output 1. Consider as an example a program routine that continually monitors a memory location that reflects the state of eight control switches (see page **??** for how this is done). This routine is waiting until someone moves a switch.

```
        .define   SWITCH = 09000h  ; Switch port is at 09000h
START:  ldab      SWITCH           ; Get initial state of switches
S_LOOP: eorb      SWITCH           ; Check for alterations
        beq       S_LOOP           ; until a change occurs
```

Two possible scenarios are:

$$\boxed{10011110}_{SWITCH} \quad \text{eorb SWITCH} \quad \boxed{10011110}_{B} \;=\; \boxed{00000000}_{B} \qquad \mathbf{Z} = 1$$

$$\boxed{10001110}_{SWITCH} \quad \text{eorb SWITCH} \quad \boxed{10011110}_{B} \;=\; \boxed{000\mathit{1}0000}_{B} \qquad \mathbf{Z} = 0$$
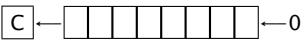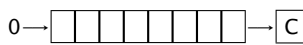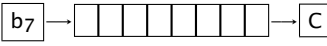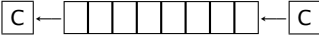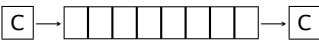
The outcome in **B** reflects any changes. In the first case there are no changes; in the second Switch 4 has just been thrown from 1 to 0. You can determine which switch changed by shifting the outcome (the change bit) right, counting until the 1 pops out into the **C** flag (see Program 6.2). You can also determine the type of change ($0 \rightarrow 1$ or $1 \rightarrow 0$) by ANDing the change byte to the new settings. If the outcome at bit 4 is a 0, then the change must have been $1 \rightarrow 0$, and vice versa.

Shifting instructions

The 6800 MPU has five categories of instructions which can shift a datum one place either left or right. Each category can target either of the accumulators or operate directly using the read–modify–write mechanism on any read/write memory location.

The linear Shift instructions lsr, lsra, lsrb (Logic Shift Right) and asl, asla, aslb (Arithmetic Shift Left) move the 8-bit operand left or right with the Carry flag catching the emerging bit. In both cases a logic 0 is shifted in. Thus:

Table 6.4 *Shifting Instructions.*

Operation	Mnemonic	V	N	Z	C	Description
			Flags			
Shift left, arithmetic or logic						Linear shift left into carry
memory	asl[1]	2	$\sqrt{}$	$\sqrt{}$	b_7	
A	asla[1]	2	$\sqrt{}$	$\sqrt{}$	b_7	$C \leftarrow \boxed{\ \ \ \ \ \ \ \ } \leftarrow 0$
B	aslb[1]	2	$\sqrt{}$	$\sqrt{}$	b_7	
Shift right, logic						Linear shift right into carry
memory	lsr	2	0	$\sqrt{}$	b_0	
A	lsra	2	0	$\sqrt{}$	b_0	$0 \rightarrow \boxed{\ \ \ \ \ \ \ \ } \rightarrow C$
B	lsrb	2	0	$\sqrt{}$	b_0	
Shift right, arithmetic						As above but keeps sign bit
memory	asr	2	$\sqrt{}$	$\sqrt{}$	b_0	
A	asra	2	$\sqrt{}$	$\sqrt{}$	b_0	$b_7 \rightarrow \boxed{\ \ \ \ \ \ \ \ } \rightarrow C$
B	asrb	2	$\sqrt{}$	$\sqrt{}$	b_0	
Rotate left						Circular shift left into carry
memory	rol	2	$\sqrt{}$	$\sqrt{}$	b_7	
A	rola	2	$\sqrt{}$	$\sqrt{}$	b_7	$C \leftarrow \boxed{\ \ \ \ \ \ \ \ } \leftarrow C$
B	rolb	2	$\sqrt{}$	$\sqrt{}$	b_7	
Rotate right						Circular shift right into carry
memory	ror	2	$\sqrt{}$	$\sqrt{}$	b_0	
A	rora	2	$\sqrt{}$	$\sqrt{}$	b_0	$C \rightarrow \boxed{\ \ \ \ \ \ \ \ } \rightarrow C$
B	rorb	2	$\sqrt{}$	$\sqrt{}$	b_0	

Note 1: Some assemblers accept the mnemonics lsl, lsla and lslb as alternatives.

Note 2: $V = b_7 \oplus b_6$ before shift.

$\boxed{10001110}_B \ \overset{\text{lsrb}}{\rightsquigarrow} \ \boxed{\mathbf{0}1000111}_B \ 0 \mapsto \boxed{C}$
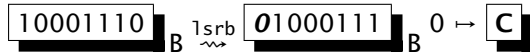
Linear Shifting operations are often used to bitwise examine a word. Say, you want to determine the leftmost logic 1 bit in Accumulator B with the position number being put in Accumulator A. For example if the pattern is:

$$\boxed{00\mathit{1}01111}_{B} \rightsquigarrow \boxed{00000101}_{A} \quad \text{(bit 5)}$$

This can be realized by continually shifting the pattern under investigation right, counting the number of times until the residue is zero. The answer given in Program 6.2 uses Accumulator A as a counter. The data settings are successively shifted right and the count incremented. As the Logic Shift Left operation brings in logic 0s from the left; eventually the residue will become all zeros, and the process terminated. Thus 00010111 (1) $\rightsquigarrow$ 00001011 (2) $\rightsquigarrow$ 00000101 (3) $\rightsquigarrow$ 00000010 (4) $\rightsquigarrow$ 00000001 (5) $\rightsquigarrow$ 00000000.

Program 6.2 *Shifting to find the highest set bit.*

```
; Data is in B, position of highest set bit to be in A
HIGH_BIT:  clra        ; Zero count

; WHILE data is not zero, shift right and increment counter
HLOOP:     lsrb        ; Shift rightmost bit into Carry
           beq  EXIT   ; IF residue is zero THEN finished
           inca        ; ELSE increment count
           bra  HLOOP  ; and do another shift

EXIT:      ... .....   ; Next program segment
```

Shifting right pops out the rightmost bit into the Carry flag. Here its value was ignored, but in many situations this can be used to examine the data on a bit by bit basis. For instance, instead of using the Increment instruction we could modify our program to add all the carry bits to Accumulator A, thus counting the total number of set bits in the byte (see Program 6.3).

Program 6.2 does not distinguish between no bits set ($00000000b$) and bit 0 set ($00000001b$). How could you modify the program to do so?

One of the major uses of shifting is to multiply and divide by powers of two. For example to divide by eight, shift left three times:

$$\boxed{00011000}_{A} \ (24) \ \overset{\text{lsra}}{\rightsquigarrow} \ \boxed{\mathit{0}0001100}_{A} \ (12) \qquad\qquad \div 2$$

$$\boxed{00\mathit{0}01100}_{A} \ (12) \ \overset{\text{lsra}}{\rightsquigarrow} \ \boxed{\mathit{00}000110}_{A} \ (6) \qquad\qquad \div 4$$

`00000110` A (6)   lsra   `000`00011 A (3)   ÷8

This technique can be used to divide signed 2's complement numbers as well, but with some modification. Let us repeat the above example, but this time for +12. The sign bit has been shown delineated with a comma for clarity, as described on page 9.
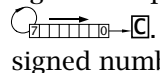
`0,0011000` A (+24)   lsra   `0,0001100` A (+12)   ÷2

`0,0001100` A (+12)   lsra   `0,0000110` A (+6)   ÷4

`0,0000110` A (+6)   lsra   `0,0000011` A (+3)   ÷8

However, we have difficulties if we try to do this for negative numbers.

`1,1101000` A (−24)   lsra   `0,1110100` A (+116)   ÷2!!!!

−24 ÷ 2 is most definitely not +116. Changing the rules so that instead of automatically shifting in zeros from the left to propagating the sign bit, that is 0s for positive and 1s for negative numbers, gives:

`1,1101000` A (−24)   asra   `1,1110100` A (−12)   ÷2

`1,1110100` A (−12)   asra   `1,1111010` A (−6)   ÷4

`1,1111010` A (−6)   asra   `1,1111010` A (−3)   ÷8

The `asr`, `asra` and `asrb` (Arithmetic Shift Right) instructions differ from the Linear Shift Right equivalents in that the sign bit is propagated right as required for division of 2's complement signed numbers thus: . This operation is normally *only* used for 2's complement signed numbers.

In all types of Shifts the oVerflow flag is set when bit 7 changes after such a shift. This is internally implemented by exclusive ORing the **C** and **N** flags *after* the shift, which translates to bit 7 and bit 6 *before* the shift. The EOR gate detects differences in inputs, see page 14. If the programmer is treating the shifted object as a 2's complement signed number, this signals a change of sign.

The final category of Shifting instructions are known as circular or Rotate instructions. These Rotate Right — — and Rotate Left — — instructions; `ror, rora, rorb, rol, rola` and `rolb`, are similar to Add with Carry, in that they can be used for multiple-precision operations. A Rotate takes in the Carry from any previous Shift and in turn saves its ejected bit in the Carry flag. As an example consider a 24-bit word stored in memory at $\boxed{_{24}\ \ 0030h\ \ _{16}\ |\ _{15}\ \ 0031h\ \ _8\ |\ _7\ \ 0032h\ \ _0}$ which can be shifted right once by the sequence:

As an example consider a 24-bit number stored in memory in three consecutive bytes at $0030{:}1{:}2h$. It is necessary to count the number of bits set to 1 in this triple-byte number.

Program 6.3 *Multiple-precision shifting to find the number of set bits.*

```
; 24-bit Data is in 0032:31:30h
; Number of bits set to be in A
COUNT_BIT: clra        ; Zero count
           ldab #24    ; Shift count
; WHILE data is not zero, shift right and increment counter
BLOOP:     lsl  0032h ; Shift rightmost byte left once
           rol  0031h ; and the middle byte
           rol  0030h ; and the leftmost byte
           adca #0    ; Add Carry to bit sum
           decb       ; Decrement shift count
           bne  BLOOP ; and repeat 32 times

EXIT:      ... ..... ; Next program segment
```

One solution is shown in Program 6.3. Here the 24-bit word is shifted left (it could equally well be done shifting right) 24 times, with the state

of the Carry flag being added to the bit count in Accumulator A after each shift. The loop count is kept in Accumulator B and decremented on each multiple-precision shift.

The setting of the CCR flags can be used after an operation to deduce, and hence act on, the state of the operand data. Thus, to determine if the value of a port located at, say, $9000h$ is zero, then:

```
ldaa 9000h        ; Get the port value
beq  SOMEWHERE    ; If all zero THEN skip to SOMEWHERE
```

will bring its contents into Accumulator A and set the **Z** flag if all eight bits are zero. The Branch if EQual to zero instruction will then cause the program to skip to another place. The **N** flag is also set if bit 7 is logic 1, and thus a Load can also enable us to test the state of this bit. The problem is Load destroys the old contents of Accumulator A, and the new data is probably of little interest. A non-destructive equivalent of Load is Test, as shown in Table 6.5. The sequence now becomes:

```
tst 9000h         ; Check the port value in situe
beq SOMEWHERE     ; IF all zero THEN skip to SOMEWHERE
```

but the Accumulator contents are *not overwritten*. It is also possible to non-destructively check the contents of either accumulator in the same manner by using the `tsta` and `tstb` instructions.

As an example, consider Program 6.2 where a shift-and-count loop was used to determine the number of the highest set bit. At that point it was observed that the given listing did not distinguish between bit 0 as the highest bit and no bits set at all. We can modify this routine by checking for all zeros *before* entering the shift-and-count loop. In Program 6.4 this is done with the `tstb` instruction following the zeroing of the count. If the data is zero then the count is decremented to give FFh toindicate this situation, and the loop is skipped over.

Test can only check for all bits zero or the state of bit 7 (negative). For data already in an accumulator, ANDing can check the state of any bit, as shown on page 119. Consider a slightly more complex problem where the program is to skip to FRED if bit 0 of memory location $9000h$ is set or to JIM if bit 3 is set or to JILL if bit 6 is clear, otherwise continue:

Table 6.5 *Data test operations.*

Operation	Mnemonic	V	N	Z	C	Description
Bit Test						Non-destructive AND
A	bita	0	√	√	•	[A]·[M]
B	bitb	0	√	√	•	[B]·[M]
Compare						Non-destructive subtract
with A	cmpa	√	√	√	√	[A] − [M]
with B	cmpb	√	√	√	√	[B] − [M]
Test for Zero or Minus						Non-destructive subtract from zero
memory	tst	0	√	√	0	[M] − 00
A	tsta	0	√	√	0	[A] − 00
B	tstb	0	√	√	0	[B] − 00

Program 6.4 *Shifting to find the highest set bit.*

```
; Data is in B, position of highest set bit to be in A
HIGH_BIT:  clra        ; Zero count
           tstb        ; Check the state of the data
           bne  HLOOP  ; IF non-zero THEN go to the loop
           deca        ; ELSE make count FFh (-1)
           bra  EXIT   ; and finished

; WHILE data is not zero, shift right and increment counter
HLOOP:     lsrb        ; Shift rightmost bit into Carry
           beq  EXIT   ; IF residue is zero THEN finished
           inca        ; ELSE increment count
           bra  HLOOP  ; and do another shift

EXIT:      ... .....   ; Next program segment
```

```
    ldaa  9000h        ; Get the data
    anda  #00000010b   ; Clear all but bit 1
    bne   FRED         ; IF non zero THEN go to FRED

    ldaa  9000h        ; Get the data again
```

```
anda  #00001000b  ; Clear all but bit 3
bne   JIM         ; IF non zero THEN go to JIM

ldaa  9000h       ; Get the data yet again!
anda  #01000000b  ; Clear all but bit 6
beq   JILL        ; IF zero THEN go to JILL

...   .....       ; ELSE continue
```

Although this code fragment works it is not very efficient, as testing the data destroys it. The equivalent non-destructive AND test operation from Table 6.5 are the Bit Test instructions, bita and bitb. Our code fragment now becomes:

```
ldaa  9000h       ; Get the data
bita  #00000010b  ; Clear all but bit 1
bne   FRED        ; IF non zero THEN go to FRED

bita  #00001000b  ; Clear all but bit 3
bne   JIM         ; IF non zero THEN go to JIM

bita  #01000000b  ; Clear all but bit 6
beq   JIM         ; IF zero THEN go to JILL

...   .....       ; ELSE continue
```

which does the same thing, but with the contents of Accumulator A remaining unchanged. Thus more tests can subsequently be carried out without reloading.

In the more general case it is often necessary to *compare* the magnitude of two numbers. Mathematically this can be done by *subtracting* the datum ([M] or a constant) from the contents of the accumulator [A] (see page 65) and checking the state of the various CCR flags. Which flags are relevant depend on whether the numbers are to be treated as unsigned (magnitude only) or signed. Where the actual magnitude of the difference between the operands is required, then the appropriate Subtract instruction can be used. However, in most cases it is sufficient to determine the relative magnitude of the quantities.

Taking the more common magnitude only case first gives:

Accumulator *higher than* datum .... No borrow, non-zero
Accumulator *equal to* datum .... Zero
Accumulator *lower than* datum .... Borrow, non-zero

In terms of our MPU, the **C** flag represents a borrow after subtraction and the **Z** flag is set on a zero outcome. This gives:

[A] *Higher than* [M] : [A]−[M] gives no borrow & non-Zero; C=0, Z=0 ($\overline{C + Z}$=1).
[A] *Equal to*   [M] : [A]−[M] gives Zero;   (Z=1).
[A] *Lower than* [M] : [A]−[M] gives a borrow;   (C=1).

Consider as an example a fuel tank with a capacity of 255 liters, with a sensor indicating the remaining volume of fuel. Assume that the sensor represents this as a byte that can be accessed from a read-only memory location called FUEL; see page **??** for how this could be done. We wish to write a routine that will light an 'empty' light if the remaining capacity is below 20 liters and ring an alarm bell if below 5 liters. This is how it could be coded:

```
ALARM: ldaa  FUEL   ; Read fuel gauge into Accumulator A
       tab          ; Copy to Acc. B for safekeeping
       suba  #5     ; FUEL - 5 to compare
       bcs   BUZZER ; IF Carry set THEN FUEL lower than 5
       subb  #20    ; FUEL - 20 to compare
       bcs   EMPTY  ; IF FUEL lower than 20 THEN EMPTY lamp
NEXT:  .....  .....
```

After the subtraction the Carry/borrow flag will be set if the contents of the accumulator (the fuel reading) is lower than the constant being subtracted (it is being compared with). Other Branches after a subtract of unsigned numbers, outlined in Table 6.6, are `bcc` (equivalent to `bhs` for Branch if Higher or Same), `beq` for Branch if EQual and `bne` for Branch if Not Equal.

As we are only interested in the relative magnitude of the two quantities, then using a Subtract instruction is overkill, in that the operand in the register will be destroyed — replaced by the difference. That is why a copy of FUEL had to be made into B above, so that the second Subtract could be executed. The Compare instruction uses the ALU to perform the subtraction and set the appropriate flag and then 'throws away' the answer, i.e. does not overwrite the datum. Compare can be thought of

as a non-destructive subtract. Like the Subtract instructions, there are variants for each accumulator (and indeed for the 16-bit Index register as well, as shown in Table 6.7). `cmpa` is used for Accumulator A and `cmpb` for Accumulator B. Using `cmpa` our code fragment becomes:

```
ALARM: cmpa  #5     ; FUEL – 5 to compare
       bcs   BUZZER ; IF LOwer than 5 THEN sound BUZZER
       cmpa  #20    ; FUEL – 20 to compare
       bcs   EMPTY  ; IF LOwer than 20 THEN EMPTY lamp
NEXT:  .....  .....
```

Where the operands under the microscope are signed 2's complement quantities then the same Subtract or Compare instructions are used. However, their relative magnitude has to be gauged by 'looking' at the **V** and **Z** flags, i.e. checking if the outcome is positive or negative, taking overflow into account (see page 10). After a subtraction of the datum from the accumulator we have:

- If the signed accumulator content is *Greater Than* the signed datum, then
    - There will be a non-zero positive outcome with no overflow.
      ELSE
    - There will be an overflow with an apparently negative non-zero outcome.

    This can be expressed as $(N \oplus V) + Z = 0$.

- If the signed accumulator content and signed datum are *EQual* then the **Z** flag will be set.

- If the signed accumulator is *Less Than* the signed datum, then:
    - There will be a negative outcome with no overflow.
      ELSE
    - There will be an overflow with an apparently positive outcome.

    This can be expressed as $N \oplus V = 1$.

This is summarized as:

[A] *Greater than* [M] : [A]−[M] ⤳ non-zero +ve result; $(\overline{N \oplus V} \cdot \overline{Z} = 1$ or $N \oplus V + Z = 0)$.
[A] *Equal to*   [M] : [A]−[M] ⤳ zero; $(Z=1)$.
[A] *Less than*   [M] : [A]−[M] ⤳ a negative result; $(N \oplus V = 1)$.

Although this seems rather complicated, all the programmer has to remember is to use Conditional Branches with the words HIgher (HI) or LOwer (LO) if the operands are unsigned, and Greater Than (GT) or Less Than (LT) if signed. Equality (or sameness) does not depend on the number representation.

As an example consider a commercial glasshouse in which a heating and ventilation system is to keep the environment at a reasonable temperature. The temperature can be read by the microprocessor as an 8-bit signed 2's complement datum as degrees Celsius at $9004h$. The algorithm is:

- IF the temperature is Greater Than $+20°$C THEN sound alarm.

- ELSE IF the temperature is Greater Than $+16°$C THEN open ventilators.

- ELSE IF the temperature is Greater Than $+10°$C THEN turn on heater.

- ELSE IF the temperature is Greater Than $+4°$C THEN turn on heater booster.

- ELSE IF the temperature is Less Than $-2°$C THEN sound alarm.

This is shown diagrammatically in the flow diagram of Fig. 6.2.

A possible coding for this task using Subtract instructions is shown in Program 6.5. Although a working coding, this implementation is inefficient in that each comparative subtraction destroys the temperature previously copied into Accumulator A. This means that the thermometer will have to be interrogated again and the value (hopefully unchanged) will have to be recopied into the MPU. The listing of Program 6.6 is an equivalent coding but this time using the non-destructive cmpa in place of suba. The length of the more efficient coding is 23 bytes as against 35 bytes for the Subtract version. Notice that in both versions the Branch instructions bgt (Branch if Greater Than) and blt (Branch if Less Than) applicable to 2's complement signed operands are used to implement the decision skip. This is rather than the bhi (Branch if HIgher than) and bls (Branch if Less than or Same) instructions which should only be used
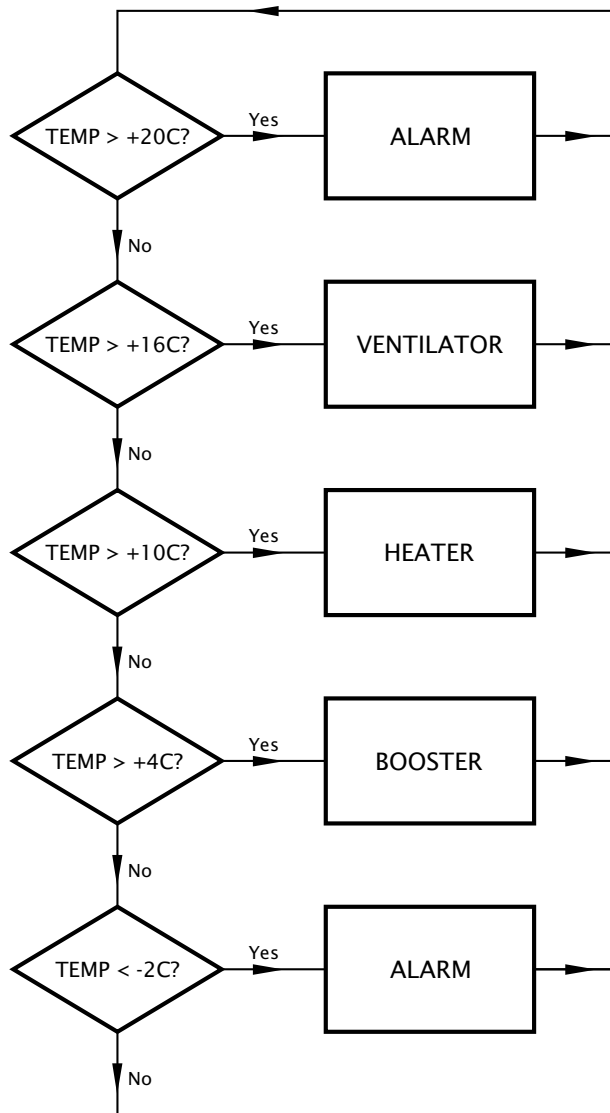
Figure 6.2 *Glasshouse environment control.*

Program 6.5 *Environment control I.*

```
        .define TEMP = 9004h

START:  ldaa  TEMP        ; Get temperature
        suba  #+20        ; Compare with +20C
        bgt   ALARM       ; IF Greater Than THEN sound alarm

        ldaa  TEMP        ; ELSE get temperature again
        suba  #+16        ; Compare with +16C
        bgt   VENTILATOR  ; IF Greater Than THEN open ventilators

        ldaa  TEMP        ; ELSE get temperature yet again
        suba  #+10        ; Compare with +10C
        bgt   HEATER      ; IF Greater Than THEN turn on HEATER

        ldaa  TEMP        ; ELSE get temp yet again and again
        suba  #+4         ; Compare with +4C
        bgt   BOOSTER     ; IF Greater Than THEN boost heater

        ldaa  TEMP        ; ELSE get temp yet again & again...
        suba  #-2         ; Compare with -2C
        blt   BOOSTER     ; IF Less Than THEN sound alarm
```

---

where unsigned data is being compared. This confusion between signed and unsigned data comparisons is a fruitful area for errors.

Program Counter operations

As shown in Table 6.6 there are 14 Conditional Branches which cause the offset to be added to the Program Counter if the indicated combination of flags is true.[5] Effectively this causes the program stream to skip forwards or backwards. There also is a BRanch Always bra which always skips no matter what the state of the flags are.

In assembly language, the programmer can directly specify this signed 2's complement byte offset; thus beq .+16 means "Add 16 to the *current* state of the PC". The current state of the PC is actually pointing to the instruction op-code *after* the Branch instruction; see Fig. 3.3(b) on page 53. The details of this augmentation of the PC are given on page 100 in the previous chapter, under the heading Relative addressing.

---

[5]Although these usually follow a sub, cmp or tst instruction, they can follow any instruction that affects the appropriate flag(s)—for example, ldab MEM then beq FRED.

Program 6.6 *Environment control II.*

```
        .define TEMP = 9004h

START:  ldaa   TEMP        ; Get temperature
        cmpa   #+20        ; Compare with +20C
        bgt    ALARM       ; IF Greater Than THEN sound alarm

        cmpa   #+16        ; Compare with +16C
        bgt    VENTILATOR  ; IF Greater Than THEN open ventilators

        cmpa   #+10        ; Compare with +10C
        bgt    HEATER      ; IF Greater Than THEN turn on HEATER

        cmpa   #+4         ; Compare with +4C
        bgt    BOOSTER     ; IF Greater Than THEN boost heater

        cmpa   #-2         ; Compare with -2C
        blt    BOOSTER     ; IF Lower Than THEN sound alarm
```

Rather than calculating these offsets by hand, the destination instruction should be labelled. If the program is subsequently altered, when it is reassembled all offsets will be automatically recalculated — a major advantage. For example repeating Program 6.3 but not using labels gives:

```
; 24-bit Data is in 0032:31:30h
; Number of bits set to be in A
          clra       ; Zero count
          ldab #24   ; Shift count
; WHILE data is not zero, shift right and increment counter
          lsl  0032h ; Shift rightmost byte left once
          rol  0031h ; and the middle byte
          rol  0030h ; and the leftmost byte
          adca #0    ; Add Carry to bit sum
          decb       ; Decrement shift count
          bne  .0F2h ; and repeat 32 times


          ... ..... ; Next program segment
```

Whilst this is technically correct it is difficult to detect any errors in the offset calculation, apart from the effort in calculating the offset ($-14d$ places from the following instruction) in the first place. Furthermore if the program is subsequently altered, perhaps by putting in an extra

Table 6.6 *Operations which affect the Program Counter.*

Operation	Mnemonic	Description	
B*xx*		*xx* is the logical condition tested	
Always (True)	bra	Always affirmed regardless of flags	
EQual[3,4]	beq	Z flag set	(Zero result)
Not Equal[3,4]	bne	Z flag clear	(Non-zero result)
Carry Set	bcs[1]	[Acc] Lower Than	(Carry = 1)
Carry Clear	bcc[2]	[Acc] Higher or Same as	(Carry = 0)
Lower or Same[3]	bls	[Acc] Lower or Same as	(C+Z=1)
Higher Than[3]	bhi	[Acc] Higher Than	(C+Z=0)
MInus	bmi	N flag set	(Bit 7 = 1)
PLus	bpl	N flag clear	(Bit 7 = 0)
oVerflow Set	bvs	V flag set	
oVerflow Clear	bvc	V flag clear	
Greater Than[4]	bgt	[Acc] Greater Than	$(\overline{N \oplus V} \cdot \overline{Z} = 1)$
Less than or Equal[4]	ble	[Acc] Less than or Equal	$(N \oplus V \cdot Z = 0)$
Greater than or Equal[4]	bge	[Acc] Greater than or Equal	$(N \oplus V = 1)$
Less Than[4]	blt	[Acc] Less Than	$(N \oplus V = 0)$
Jump	jmp	Absolute unconditional goto	
No Operation	nop	Only increments Program Counter	

Note 1: Some assemblers allow the alternative blo[3].
Note 2: Some assemblers allow the alternative bhs[3].
Note 3   After a Subtract or Compare of unsigned data.
Note 4   After a Subtract or Compare of 2's complement signed data.

instruction inside the loop, then the offset will have to be recalculated. Whilst this may seem a trivial task, remember that the average embedded microprocessor program has over 30,000 lines!

The jmp (JuMP) instruction is a go-to operation as compared to a relative skip. This can use any appropriate address mode and go directly

anywhere in the address space. A combination of a Conditional branch and `jmp` can be used to simulate a Long branch, as coded in Example 5.3 on page 105, where the skip range is outside the single-byte offset range of +129 through −126 bytes from the Branch instruction.

The single-byte `nop` (No OPeration) instruction does nothing, but as a consequence of its fetch the Program Counter will increment in the normal way. This 'useless' process takes two clock cycles to execute and `nop` is often used to create a short delay with no other side effects (see Program 8.1 on page 174).

Address register instructions

The 6800 MPU has two 16-bit registers accessible to the programmer which are designed to hold addresses. The function of both the Stack Pointer and Index registers have been described in Chapter 4. Briefly, the Index register is designed to hold a pointer address to be used in conjunction with the Indexed address mode (see page 99). The Stack Pointer register is used to keep track of an area of memory designated by the programmer, known as a stack, to temporarily hold data which can be pushed out to and pulled from this area of memory, as shown in Fig. 6.1.

As can be seen in Table 6.7 it is possible to Increment (`inx` and `ins`) and Decrement (`dex` and `des`) either of the registers. Notice that the only flag affected by these instructions is the **Z** flag, and thus only the Conditional instructions `beq` and `bne` can be used following these instructions.   In a similar manner the ComPare indeX instruction (`cpx`) only correctly affects the **Z** flag; although in this case both **V** and **N** flags are altered, they only reflect the subtraction of the high byte of the Index register, IXH, and should not be used for Conditional branching. Thus the code fragment:

```
LOOP: inx        ; Increment pointer
      cpx #1000h ; Compare with the address 1000h
      bls LOOP   ; IF Less than or the Same repeat loop
```

is illegal as the Conditional branch instruction `bls` will not always operate properly with the `cpx` instruction. This is because the 6800's ALU cannot directly implement 16-bit arithmetic, so operations on 16-bit registers are executed by first processing the lower byte followed by the high byte. The logic for setting the flags in this situation only works for the **Z** flag. Notice the the **C** flag is not altered by this instruction either.[6]

---

[6]The 8-bit 6809 MPU has fixed this problem and the full range of Condition branches

Table 6.7 *Address register instructions.*

Operation	Mnemonic	V	N	Z	C	Description
Compare						Non-destructive 16-bit subtract
IX	cpx	1	2	√	•	[IXH:IXL]−[M:M−1]
Decrement						16-bit subtract one with no borrow
IX	dex	•	•	√	•	[IXH:IXL] <- [IXH:IXL]−1
SP	des	•	•	•	•	[SPH:SPL] <- [SPH:SPL]−1
Increment						16-bit addition of one with no carry
IX	inx	•	•	√	•	[IXH:IXL] <- [IXH:IXL]+1
SP	ins	•	•	•	•	[SPH:SPL] <- [SPH:SPL]+1
Load						Copies two memory bytes to register
to IX	ldx	0	√	√	•	[IXH:IXL]<-[M:M+1]
to SP	lds	0	√	√	•	[SPH:SPL]<-[M:M+1]
Store						Copies 2-byte register out to memory
from X	stx	0	√	√	•	[M:M+1]<-[IXH:IXL]
from S	sts	0	√	√	•	[M:M+1]<-[SPH:SPL]
Transfer						Copy between address registers
IX + 1 $\longrightarrow$ SP	txs	•	•	•	•	[SPH:SPL]<-[IXH:IXL]+1[3]
SP − 1 $\longrightarrow$ IX	tsx	•	•	•	•	[IXH:IXL]<-[SPH:SPL]−1[4]

Note 1: Only affected if the subtraction of the high IX byte overflows.
Note 2: Only affected if the subtraction of the high IX byte would give $b_{15} = 1$.
Note 3: The value of IX less one is transferred.
Note 4: The value of SP plus one is transferred.

The contents of an Address register can be copied into memory using the stx (STore indeX) and sts (STore Stack pointer) instructions. In a similar manner data in memory can be copied into either 16-bit register using the ldx (LoaD indeX) and lds (LoaD Stack pointer) instructions.

Once again, when using these instructions you must remember that memory is organized as 8-bit bytes, that is an address of a memory datum locates a *byte*. In using an address in conjunction with these double-byte registers a problems arises in that 2-byte data is being referenced. For

can be used after 16-bit register operations.
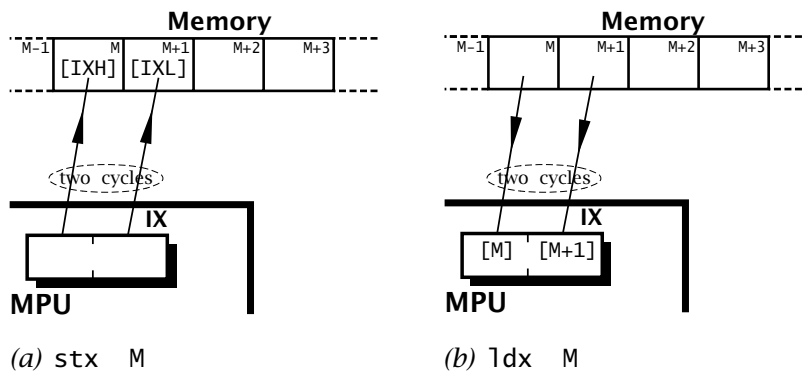
Figure 6.3 *16-bit Store and Load to/from memory operations.*

example `ldx 0030h` seems to say "Load the byte datum in memory in $0030h$ into the 2-byte address register". What really happens is shown in Fig. 6.3(a) where the *two bytes* in memory at $0030h$ and $0031h$ are actually copied although only the address $0030h$ was specified in the instruction. As the Data bus is only eight bits wide it actually takes two Read cycles to bring down down the full 16 bits (actually the high byte at $0030h$ is read first followed immediately by the byte datum at $0031h$ — but this is invisable to the programmer). Similarily when storing data from an address register the lower address (that is for the most significant datum byte) is specified. For example `stx 0032h` actually copies the contents of the Index register IXH:IXL into $0032:3h$, as shown in Fig. 6.3(b).

Flag instructions

Finally Table 6.8 shows a few instructions that can be used to directly set or clear the **C** (`sec` and `clc`) and **V** (`sev` and `clv`) flags. It is also possible to set and clear the **I** mask bit using `sei` and `cli` instructions to inhibit and permit maskable interrupts (see page **??**). All the flags/mask can be simultaneously primed by copying the byte in Accumulator A into the Code Condition register by using the (puzzling mnemoniced) `tap` (Transfer A to **??**) instruction. In a similar manner the CCR can be transferred to Accumulator A for futher examination using `tpa`.

Table 6.8 *Direct flag operations*

Operation	Mnemonic	H	I	V	N	Z	C	Description
		Flags						
CLear Carry	clc	•	•	•	•	•	0	[C] <- #0
SEt Carry	sec	•	•	•	•	•	1	[C] <- #1
CLear oVerflow	clv	•	•	•	•	0	•	[V] <- #0
SEt oVerflow	sev	•	•	•	•	1	•	[V] <- #1
CLear Interrupt mask	cli	•	0	•	•	•	•	[I] <- #0
SEt Carry	sec	•	1	•	•	•	•	[I] <- #1
Transfer A to CCR	tap	A5	A4	A3	A2	A1	A0	[CCR] <- [A]
Transfer CCR to A	tpa	•	•	•	•	•	•	[A] <- [CCR]

Table 6.9 *Direct flag operations.*

The shortform instruction set of Table 6.10 gives a summary of the more commonly used instructions. A full instruction set is laid out in Appendix B.

Instruction[1]	Description	Address modes				Flags		
		#	Inher	Absol	Index	N	Z	C
Arithmetic								
adda [ea]	ADD to A	*		*	*	√	√	√
adca [ea]	AdD with Carry to A	*		*	*	√	√	√
clr [ea]	CLeaR memory			*	*	0	1	0
clra	CLeaR A		*			0	1	0
dec [ea]	DECrement memory			*	*	√	√	•
deca	DECrement A		*			√	√	•
dex	DECrement IX		*			•	√	•
inc [ea]	INCrement memory			*	*	√	√	•
inca	INCrement A		*			√	√	•
inx	INCrement IX		*			•	√	•
suba [ea]	SUBtract from A	*		*	*	√	√	√
sbca [ea]	SuBtract with Carry/borrow from A	*		*	*	√	√	√
Movement								
ldaa [ea]	LoaD (copy) to A from memory	*		*	*	√	√	•
ldx [ea]	LoaD (copy) to IX from memory	*		*	*	√	√	•
staa [ea]	STore (copy) from A to memory			*	*	√	√	•
stx [ea]	STore (copy) from IX to memory			*	*	√	√	•
Logic								
anda [ea]	Bitwise AND A with memory	*		*	*	√	√	•
com [ea]	COMplement (NOT) memory			*	*	•	•	1
coma	COMplement (NOT) A		*			•	•	1
eora [ea]	Bitwise EOR A with memory	*		*	*	√	√	•
oraa [ea]	Bitwise OR A with memory	*		*	*	√	√	•
asl[2] [ea]	Arithmetic Shift Left memory one place			*	*	0	√	b_7
asla[2]	Arithmetic Shift Left A one place		*			0	√	b_7
lsr [ea]	Logic Shift Right memory one place			*	*	0	√	b_0
lsra	Logic Shift Right A one place		*			0	√	b_0
Testing								
cmpa [ea]	CoMPare A with memory	*		*	*	√	√	√
cpx [ea]	CoMPare IX with memory	*		*	*	√	√	•
tst [ea]	TeST memory for zero or negative (bit 7 = 1)			*	*	√	√	•
tsta	TeST A for zero or negative (bit 7 = 1)			*	*	√	√	•
Branch (see over)								
bra	BRAnch always					•	•	•
beq	Branch if EQual to zero					•	•	•
bne	Branch if Not Equal to zero					•	•	•
bcc/bhs	Branch if Carry Clear/Higher or Same					•	•	•
bcs/blo	Branch if Carry Set/LOwer than					•	•	•
bpl	Branch if PLus (bit 7 = 0)					•	•	•
bmi	Branch if MInus (bit 7 = 1)					•	•	•
jmp	JUmP (goto) directly			*	*	•	•	•
Subroutine								
jsr	Jump to SubRoutine			*	*	•	•	•
rts	ReTurn from Subroutine		*			•	•	•

* : Available	√ : Flag operates normally		
1 : Flag set	• : Not affected	[1]	: Replace a by b for Acc. B
[ea] : Effective address	0 : Flag cleared	[2]	: Alternatively lsl/lsla

Conditional Branches, B*xx*					
beq	EQual	Z=1	bvc	oVerflow Clear	V=0
bne	Not Equal	Z=0	bvs	oVerflow Set	V=1
bcc	Carry Clear	C=0	bpl	PLus	N=0
bcs	Carry Set	C=1	bmi	MInus	N=1
bhs	Higher or Same	C=0	bge	Greater or Equal	$N\oplus V=0$
bhi	HIgher than	C+Z=0	blt	Less Than	$N\oplus V=1$
bls	Lower or Same	C+Z=1	bgt	Greater Than	$\overline{N\oplus V}\cdot\overline{Z}=1$
bcs	LOwer than	C=1	ble	Less or Equal	$\overline{N\oplus V}\cdot\overline{Z}=0$

Table 6.10 *Shortform 6800 instruction set (continued next page).*

Examples

Example 6.1

The circuit diagram of Fig. 6.4 shows a 7-bit pseudo-random number generator (PRNG) based on a shift register with an Exclusive-OR gate feedback. Devise a routine to continually send these 127 binary random numbers to a port located at $9001h$. The routine is to initialize the number to any non-zero value.

Solution

A suitable task list is:

1. Initialize the number to 01.

2. DO forever.

    (a) Shift number left once to align bits 5 & 6.

    (b) Bitwise EOR the number and its shifted copy.

    (c) Shift the outcome twice left to pop out bit 6 into the **C** flag, which will be F6⊕F5.

    (d) Shift the original number left once with **C** becoming the new bit 0.

The listing in Program 6.7 follows the task list fairly closely. The value of the number is temporarily saved in memory so that it can be shifted in its accumulator and then Exclusive-ORed as required. It can be retrieved later and the new bit F5⊕F6 shifted in using the Rotate Left instruction to form the next random number.

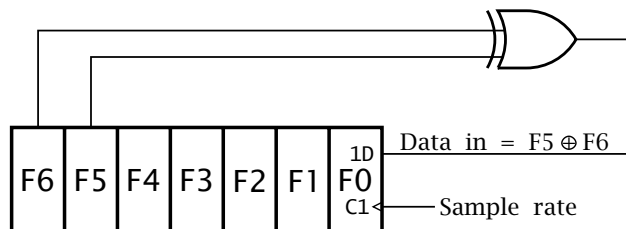What would happen if the initial value of the random number was zero?

Figure 6.4 *A 7-bit pseudo-random number generator.*

Program 6.7 *A 7-bit pseudo-random number generator.*

```
        .define PORT = 9001h, MEMORY = 0030h
PRNG:   ldab  #1    ; Initial value of random number is 01

P_LOOP: stab MEMORY ; Make a copy in memory

        aslb          ; Shift number to align bits 5 & 6
        eorb MEMORY ; Bitwize EOR them
        aslb          ; Shift twice left to put 5EOR6 in Carry
        aslb

        ldab MEMORY ; Get back original number from memory
        rolb          ; Shift left with carry coming in

        stab PORT   ; and send this new random number out

        bra  P_LOOP ; and repeat
```

Example 6.2

A certain electronic game is based on the generation of a random number stored as a byte in Accumulator A.. This is to drive an array of seven LEDs designed to mimic the layout of a die (see Fig. **??** on page **??**). As a die (singular for dice) can only represent six different values it is necessary to convert this byte to a random number in the range 1–6. Show how this may be coded.

Solution

Mathematically this function may be implemented by dividing the byte random number by six. The random remainder will then range from 0–5. This remainder is known as the modulo-6 equivalent of the original number. Adding one gives the desired range 1–6.[7]

In Program 6.8 the ÷6 operation is implemented by the repetitive subtraction of six. The residue is checked against six at the beginning

---

[7]In the **C** language the modulo operator is %, so this function could be expressed as (N%6)+1.

Program 6.8 *Modulo-6 generation.*

```
; Random number N in Accumulator A on entry
; (Modulo-6)+1 equivalent to be generated into Accumulator B
MOD_6:      tab               ; Copy N into Accumulator B
MOD_6_LOOP: cmpb #6           ; Is the residue less than six?
            bcs  FINISHED     ; IF underflow (lower than) then finished
            subb #6           ; ELSE take away six
            bra  MOD_6_LOOP   ; and go again
FINISHED:   addb #1           ; Residue converted to 1--6
            ... ......        ; Exit with (N%6)+1 in Accumulator B
```

---

of each pass through the loop. If this remainder is lower than six (a borrow/carry is generated by the `cmpa` subtract) then the process is finished. The residue is the modulo-6 version of the original datum.

As we have to subtract six anyway, it is more efficient to only use `subb` instead of both `cmpb` and `subb`. Can you write a more efficent coding (one instruction less) to implement this function?

Example 6.3
A certain television show has eight contests evenly divided into Team A and Team B. Each member has a switch, giving logic 1 when pressed, which may all be read simultaneously by the microprocessor at memory location $9001h$. Team A switches appear on the lower four bits of the byte.

Write a routine that will:

- Decide when a response to the question has been made — any switch closed.

- Determine the team identity that has responded, by clearing Accumulator A for Team A and setting it to any non-zero value to signify Team B.

- Ascertain which team member pressed his or her switch by putting the member number 0–3 in Accumulator B.

You may assume that memory in page 0 can be used for temporary storage of variables.

Solution

The task list for this program is:

    1.WHEN SWITCHES are non zero DO:

    2.IF lower nybble of SWITCHES non zero THEN DO:

        (a)Set TEAM_ID to zero for Team A

        (b)COUNT = 0

        (c)DO FOREVER

            i.SHIFT SWITCHES once right

            ii.BREAK IF Carry = 1

            iii.COUNT++

    3.ELSE DO:

        (a)Set TEAM_ID = non zero for Team B

        (b)COUNT = 3

        (c)DO FOREVER

            i.SHIFT SWITCHES once left

            ii.BREAK IF Carry = 1

            iii.COUNT--

In essence when the value at $9001h$, which we call SWITCHES, is non zero then one half is blanked out by ANDing with zero. Based on the state of the remaining nybble we can determine if it was Team A or Team B. By shifting the switch state right or left (Team A or Team B respectively) and counting up or down respectively we can find who pressed the switch. For example if Team A contestant 2 pressed the switch and the settings have been loaded into Accumulator A:

B = 0 (COUNT)

`00000100` A $\xrightarrow{lsra}$ `00000010` A $\mapsto$ C = 0

B = 1

`00000010` A $\xrightarrow{lsra}$ `00000001` A $\mapsto$ C = 0

B = 2

`00000001` A $\xrightarrow{lsra}$ `00000000` A $\mapsto$ C = 1

giving the requisite answer of two in Accumulator B.

A possible coding of this list is:

Program 6.9 *A television quiz ennunciator.*

```
        .define TEAM_ID = 0000h, SWITCHES = 9001h
; Task 1
QUIZ:   ldaa    SWITCHES   ; Keep checking for a switch closure
        bne     QUIZ

; Task 1A
        clr     TEAM_ID    ; Zero the temp location for team id.
        bita    #00001111b ; Blank off Team B (Team A data unchanged)
        beq     TEAM_B     ; IF zero THEN must be Team B

; Team A
        clrb               ; COUNT = 0
A_LOOP: lsra               ; Shift switch data once right
        bcs     FINI       ; BREAK IF Carry = 1
        incb               ; Increment COUNT in B
        bra     B_LOOP     ; and try again

; Task 1B for Team B
TEAM_B: dec     TEAM_ID    ; Make team id non zero
        ldab    #3         ; COUNT = 3
B_LOOP: asla               ; Shift switch data once left
        bcs     FINI       ; BREAK IF Carry = 1
        decb               ; Decrement COUNT in B
        bra     B_LOOP     ; and try again

FINI:   ldaa    TEAM_ID    ; Team id. in Accumulator A
        ...     .....
```

- Is the program biased if more than one panel member presses his/her switch?

- What problems could arise in practice?

Self-assessment questions

6.1 Code a program that reads a bank of switches controlling a dish-washer, which can be read by the MPU at a port located at 9000*h* which:

- Continually samples Switch 7 (bit 7, the most-significant bit) and only moves on when this reads as a 1.

- Goes to a program line ECOMONY if Switch 0 is 0.

- Otherwise continues to the program labelled NORMAL.

6.2 Design a program that will branch to a routine called GREEN (that turns on a green lamp) if the temperature read at a port at 9000*h* in integer 2's complement form is greater than +2°C and go to FLASH_RED if less than −1°C otherwise continues on to a routine called AMBER that activates an amber lamp.

6.3 The 6800/2 MPU has no direct way to push and pull the contents of the Index register onto/off the stack. Later members of the family, such as the 6809 and 6811 processors have a pshx and pulx instruction. Show how you might simulate these absent instructions. Your technique should be such that the contents of the accumulators should be unchanged following both routines, however, you can use up to three bytes of memory.

6.4 How could you count the number of set bits in an array of 256 bytes between 0000*h* and 00FF*h*? Make use of two bytes in memory at 0200:1*h* to hold the bit count (which can be up to 2048).

6.5 Parity is a method of error protection whereby each byte of data has the most significant bit set in such a way to ensure that the overall number of bits in the byte is odd or even. Write a routine that will convert a byte at 0030*h* to odd 1's parity. You may assume that the existing pattern has a 0 in the most significant bit position 7.
　You will:

1. Count up the number of bits in this byte (see Program 6.3)

2. Determine if this number is odd or even. The weight of the least significant bit is one (2^0), whilst all other bits are even (eg. 2, 4, 8...). Thus an odd binary number always has bit 0 = 1.

3. IF even THEN set bit 7 of the number to 1 ELSE leave bit 7 at 0.

6.6 Design a coding that will test read/write memory between $0000h$ and $1FFFh$. The procedure is to store a test vector, for example $01010101b$ in the byte under test and then compare it with the vector. If they are the same then the memory location is considered to be functioning correctly. Repeat this for all 8 Kbytes. If the check shows a problem the routine is to immediately exit with the problem byte in the Index register.

6.7 Repeat the last SAQ, but this time use all numbers from $00000000b$ to $11111111b$ to test each byte. That is each byte is tested 256 times.

6.8 Write a program that will convert a byte located in memory location $0030h$ called BINARY of value $00h$– FFh (0 – 255d) to a 3-digit BCD equivalent in locations $0100h$ for the hundred's digit (called HUNDS), $0101h$ for the ten's digit (called TENS) and $0102h$ for the units digit (called UNITS). For example if BINARY is FEh ($11111110b$) then the outcome will be 02 hundreds, 05 tens and 04 units; that is $FEh = 255d$.

   Hint: The easiest way to do this is to keep a tally of how many times a hundred can be subtracted from the binary number without underflowing (generating a borrow), then how many tens can be subtracted from the residue number without underflow. Whatever is left will be the value of the units.

Assembly language

We have now been writing programs with gay abandon since Chapter 3. For clarity these listings were written in a human-readable form. Thus, instructions are represented as a short mnemonic, such as `inc`; the registers similarly have mnemonics, such as `x`; lines have been labelled and comments attached. Such symbolic representations are only for human consumption. The MPU knows nothing beyond the binary codes making up operation codes and address modes, such as shown on page 50.

With the help of the programmer's manual supplied by the manufacturer, it is possible to translate from the human-readable symbolic form to machine-readable binary. However, it really isn't practical to do this for programs of more than a few dozen instructions. As well as being excruciatingly slow and tedious, it is error-prone and difficult to maintain whenever there are changes to be made.

Computers are good at doing boring things quickly and accurately; and translating from symbolic to machine code definitely falls into this category. Here we will briefly look at the various software packages that aid in this translation process.

After reading this chapter you will:

- *Know what assembly-level language is and how it relates to machine code.*

- *Appreciate the advantages of a symbolic representation over machine-readable code.*

- *Understand the function of the assembler.*

- *Appreciate the process involved in translating and locating an assembly-level language program to absolute machine code.*

- *Understand the structure of a machine-code file.*

- *Understand the role of a loader.*

The essence of the conversion process is shown in Fig. 7.1. Here the program is prepared by the tame human in symbolic form, digested by the computer and output in machine-readable form. Of course this simple statement belies a rather complex process, and we want to examine this in just enough detail to help you in writing your programs.
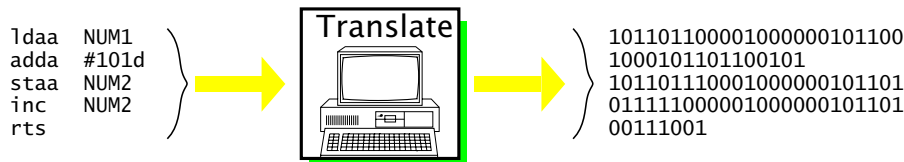
```
ldaa  NUM1
adda  #101d
staa  NUM2
inc   NUM2
rts
```

Translate

```
10110110000100000101100
1000101101100101
10110111000100000101101
01111100000100000101101
00111001
```

Figure 7.1 *Conversion from assembly-level source code to machine code.*

The various translator and utility computer packages are written and sold by many software companies, and thus the actual details and procedures differ somewhat between the various commercial products. Here we will utilize Real Time Systems products[1] for illustrative purposes. Although most products are broadly similar, you will have to consult the documentation of the particular packages you are using for specific details.

Using the computer to aid in translating code from more user-friendly forms (known as **source code**) to binary machine code (known as **object code**) and loading this into memory began in the late 1940s for mainframe computers. At the very least it permitted the use of higher-order number bases, such as hexadecimal.[2] In this base the code fragment of Fig. 7.1 becomes:

```
B6102C
8B65
B7102D
7C102D
39
```

---

[1]Written by and available from RTS, M & G House, Head Road, Douglas, Isle of Man, British Isles. Details on `http://mannet.mcb.net/rts/xa8.html`.

[2]Actually base-8 (octal) was the popular choice for several decades.

A **hexadecimal loader** will translate this into binary and put the code in designated memory locations. This loader might be the software in your EPROM programmer or part of the operating system of the target computer.[3] Hexadecimal coding has little to commend it, except that the number of keystrokes is reduced — but there are more keys — and it is slightly easier to spot certain types of errors.

As a minimum, a symbolic translator, or **assembler**,[4] is required for serious programming. This allows the programmer to use mnemonics for the instructions and internal registers, with names for constants, variables and addresses. The symbolic language used in the source code is known as **assembly language**. Unlike high-level languages, such as **C** or PASCAL, assembly language has a *one-to-one relationship* with the generated machine code, i.e. one line of source code produces one instruction. As an example, Program 7.1 shows the source code of a module that will compute the average of an array of 24 data bytes (perhaps daily temperature sampled hourly) located in memory from $1000h$ upwards. The program code itself begins at $C100h$ and data memory from $0000h$ upwards is reserved for the 24-byte array and for a 1-byte location holding temporarily the average.

Giving names to addresses and constants is especially valuable for longer programs, which may easily exceed 10,000 lines. Together with the use of comments, this makes code easier to debug, develop and maintain. Thus, if we wished to change the size of the array to 144 (say, sampling the temperature every 10 minutes), then we need only alter the first line to:

```
.define NUMBER = 144
```

or even, as most assemblers can do simple constant arithmetic (see also line 9):

```
.define NUMBER = 24*6
```

and then retranslate to machine code. In a program with, say, 50 references to the constant NUMBER, the alternative of altering *all* these constants from 24 to 144 is laborious and error-prone.

---

[3]For example, MS-DOS or the monitor ROM in your trainer board.

[4]The name is very old; it refers to the task of translating and *assembling* together the various modules making up a program.

Program 7.1 *Absolute assembly-level source code for our averaging module.*

```
         .define NUMBER = 24 ; 1:  Number of elements n=24
         .org 0C100h         ; 2:  Prog text begins @ C100h
AVERAGE: ldx  #ARRAY         ; 3:  Point to start of array
         clra                ; 4:  Zero the double-byte sum total
         clrb                ; 5:  with the 2 accums. holding sum
LOOP:    ldab 0,x            ; 6:  Add array byte element n
         adcb #0             ; 7:  Add carry into top byte

         inx                 ; 8:  Increment pointer
         cpx  #(ARRAY+NUMBER); 9:  Over the top yet?
         bne  LOOP           ; 10: IF not THEN again
; 11: Now divide by 24 by subtraction

         clr  AV             ; 12: Zero the average byte
DIV_LOOP:subb #24            ; 13: Take away 24 from lower byte
         sbca #0             ; 14: & any borrow from the upper byte
         bcs  EXIT           ; 15: IF borrow produced then finished
         inc  AV             ; 16: ELSE note a successful subtraction
         bra  DIV_LOOP       ; 17: and go again
         ldaa AV             ; 18: Get average into Acc. A
EXIT:    rts                 ; 19: and return to caller

         .org   0h           ; 20: Data area
ARRAY:   .byte  [NUMBER]     ; 21: Reserve 24 data bytes for array
AV:      .byte  [1]          ; 22: and one byte for the average
```

Of course symbolic translators demand more of the computer running them than simple hexadecimal loaders, especially in the area of memory and backup store. Because of this, their use in small MPU-based projects was limited until the late 1970s, when powerful personal computers appeared. Prior to this, either mainframe and minicomputers or special-purpose MPU development systems were required. Such solutions were inevitably expensive.

Translation involves two tasks:

• Conversion of the various instruction mnemonics and labels to their machine-code equivalent.
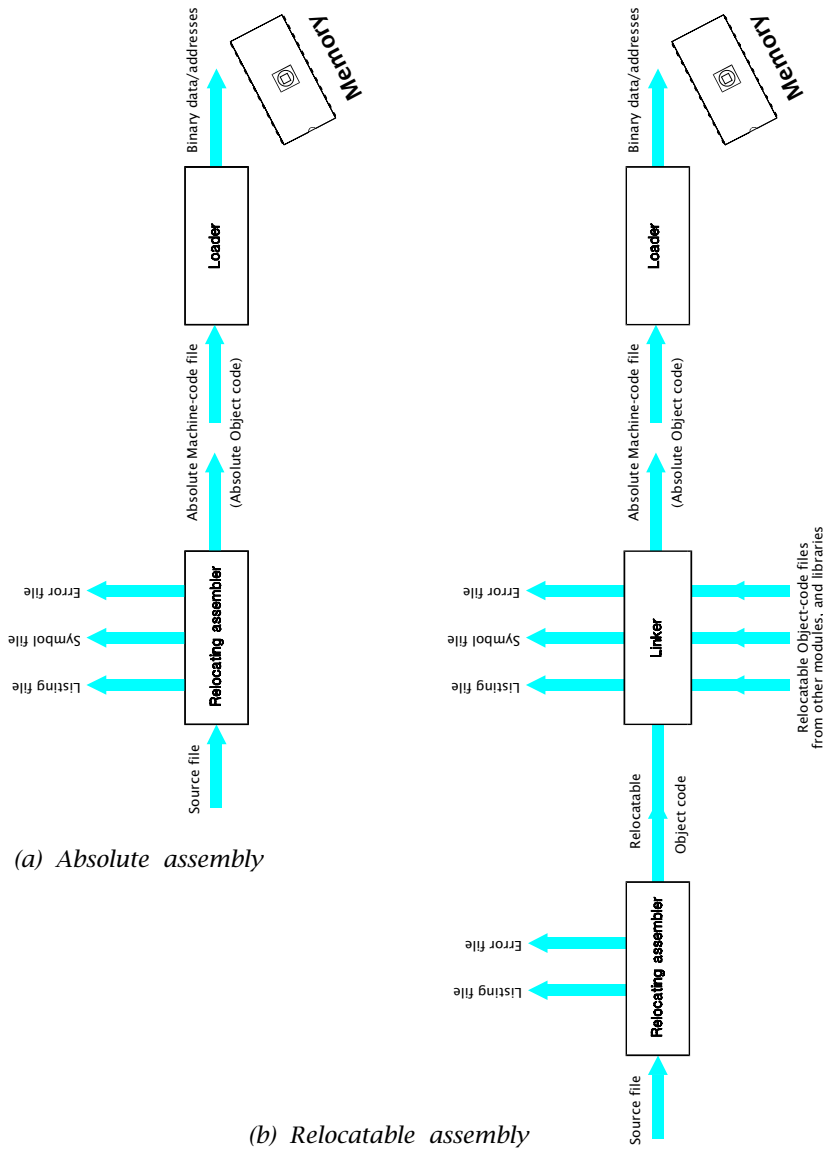
(a) Absolute assembly

(b) Relocatable assembly

Figure 7.2 *Assembly-level code translation.*

- The location of the instructions and data in the appropriate memory locations.

It is the second of these that is perhaps more difficult to understand.

Program 7.1 was designed to be processed by an **absolute assembler**. Here the programmer uses embedded directives (in this assembler distinguished by commands with a leading period) to tell the assembler to place the code in specified memory addresses. The use of the directive **.org** (for "ORiGin") means that the programmer needs to know where everything is to be placed. This absolute assembler process is shown in Fig. 7.2(a).

Absolute assembly is adequate where a program comprises a single self-contained file; which is the case in this text. However, real projects, often consisting of more than 10,000 lines of code, require team work. With many modules[5] being written by different people, perhaps also coming in from outside sources and commercial libraries, some means must be found to *link* the appropriate modules together to give the one executable machine-code file. For example, you may have to call up one of the modules that Fred has written some time ago. You will not know exactly where in memory this module will reside until the project has been completed. What can you do? Well a module should have its entry point labelled; say, FRED: in this case. Then you should be able to jump to FRED without knowing exactly what address this label represents.

The process used to facilitate this is shown in Fig. 7.2(b). Central to this modular tie-up is the **linker** program which satisfies such external cross-references between the modules. Each module's **source-code** file needs to have been translated into **relocatable object code** prior to the linkage. "Relocatable" means that its final location and various addresses of external labels have yet to be determined. This translation is done by a **relocatable assembler**. Unlike absolute assembly, it is the linker that determines where the machine code is to be located in memory, not the programmer.

In some products, the output of the linker may need some massaging to give the absolute machine-code file format that the loader program understands. The term "absolute" simply means that the machine-readable code is in its final form, and contains its fixed locations in memory.

---

[5]We will discuss modules in more detail in the next chapter.

In this book we will use an absolute assembler to translate source code to absolute machine code. As the 6800 family is a rather simple processor, this approach is adequate for the majority of projects where a MPU of this capacity is used. To clarify the process we will take a single module through from the creation of its source file to the final absolute machine-code file.

Editing

First the source file must be created using any **text editor**. Most operating systems come with a simple text editor; for example, `edit` for MS-DOS and `notepad` for Microsoft's Windows 95/NT. Third-party products are also available. A text editor differs from a wordprocessor in that no embedded "funny" codes are inserted, giving formatting and other information. For instance, if you want a new line then you hit the <RET> key; the text editor will not wrap around for you. However, most wordprocessors have a text mode and can be used to create program source files.[6] Traditionally assembly-level source files have an extension of `.s` or `.src`. The source file we will use as our model is given in Program 7.1.

The format of a line of source code in the assembler used in this text looks like:

Label (optional)   *Comment (optional)*

```
LOOP:  ldx   #ARRAY    ; Point IX to bottom of array
```

Instruction mnemonic   *Operand*

All lines, with the exception of comment-only lines, must contain an instruction (either executable by the MPU or an assembler directive) and any relevant operand or operands. If a line is labelled, then the label is delineated by a following colon.[7] A line label names the address of the first following executable instruction. This name must not start with a number and should be no more than 32 alphanumeric characters. The optional comment is delineated by a semicolon,[8] and whole-line comments are

---

[6]For example, programs for this book were created using Wordstar 2000 in its non-document format.

[7]With assemblers that do not use a colon as a delimiter, the first character in an unlabelled line has to be a space.

[8]Many assemblers use a *.

permitted (see line 11 of Program 7.1. Comments are ignored by the assembler, and are there solely for human-readable documentation. Notes should be copious and should explain what the program is doing, and not simply repeat the instruction. For example:

```
clra ; Clear A
```

is a waste of energy:

```
clra ; Zero the bit count
```

is rather more worthwhile. Not, or minimally, commenting source code is a frequent failing, not confined to students. A poorly documented program is difficult to debug and subsequently to alter or extend. The latter is sometimes known as program maintenance.

Assembling

If there are no syntax errors, then the assembler will translate your source code into absolute object code, which is basically machine code with information concerning the location in memory it is to be placed. Syntax errors include such things as referring to labels that don't exist or instructions that are not recognized. The output of the assembler will include an error file giving any such errors. If there are no syntax errors, a listing file, symbol file and machine-code file are generated.

Listing

The **listing file** of Table 7.1 reproduces the original source code together with the location in memory of each instruction and its code, all in hexadecimal. The listing file has only documentation value and is not executable by the processor.

Symbols

The **symbol file** shown in Table 7.2 gives a list of symbols together with their equivalent address. In this example there are six labels and one assignment of the value $24d$ ($18h$) to the label NUMBER, but in a large program there may be several hundred. Knowing the address to which a label refers and constant definations is useful in setting up breakpoints when debugging programs.

Absolute Code

The conclusive outcome of the translation process is the **machine-code file**. As can be seen in Table 7.3, such files essentially consist of lines of

```
 1                      .processor  m6800
 2                      .define NUMBER = 24 ; 1:  Number of elements n=24
 3                          .org 0C100h    ; 2:  Prog text begins @ C100h
 4 C100 CE0000 AVERAGE: ldx  #ARRAY    ; 3:  Point to start of array
 5 C103 4F             clra           ; 4:  Zero the double-byte sum total
 6 C104 5F             clrb           ; 5:  with the 2 accums. holding sum
 7 C105 E600    LOOP:   ldab 0,x       ; 6:  Add array byte element n
 8 C107 C900            adcb #0        ; 7:  Add carry into top byte
 9
10 C109 08              inx            ; 8:  Increment pointer
11 C10A 8C0018     cpx  #(ARRAY+NUMBER); 9:  Over the top yet?
12 C10D 26F6            bne  LOOP      ; 10: IF not THEN again
13            ; 11: Now divide by 24 by subtraction
14
15 C10F 7F0018          clr  AV        ; 12: Zero the average byte
16 C112 C018    DIV_LOOP:subb #24      ; 13: Take away 24 from lower byte
17 C114 8200            sbca #0        ; 14: & any borrow from the upper byte
18 C116 2507            bcs  EXIT      ; 15: IF borrow produced then finished
19 C118 7C0018          inc  AV        ; 16: ELSE note a successful subtraction
20 C11B 20F5            bra  DIV_LOOP  ; 17: and go again
21 C11D 9618            ldaa AV        ; 18: Get average into Acc. A
22 C11F 39     EXIT:    rts            ; 19: and return to caller
23
24                      .org   0h      ; 20: Data area
25 0000        ARRAY:   .byte  [NUMBER]; 21: Reserve 24 data bytes for array
26 0018        AV:      .byte  [1]     ; 22: and one byte for the average
27                      .end
```

Table 7.1 *The listing file* average.ls.

```
C112   DIV_LOOP
0018   AV
0018   NUMBER
C100   AVERAGE
C11F   EXIT
C105   LOOP
0000   ARRAY
```

Table 7.2 *The symbol file* average.sym.

hexadecimal digits representing the binary machine code, each preceded by the address of the first byte of the line. This file is ready to be loaded into memory, and subsequently run.

```
S123C100CE00004F5FE600C900088C001826F67F0018C018820025077C001820F59618397B
S1030000FC
S9
```

Table 7.3 *The absolute* S2-S8 *machine-code file* `average.hex`.

In the MPU world there are many different formats in common use. Although most of these de facto standards are manufacturer-specific, in the main they can be used for any brand of MPU. The format of the machine-code file shown here is known as Motorola S1-S9. Let us look at the first line, or record, which contains the code for the instructions in the program `average.s` in more detail:

The loader recognizes that a code record follows when the characters S1 are received. The characters S9 signify the end-of-file line. Code records begin with the tally in hexadecimal of all characters after S1, followed by the four-digit hexadecimal address of the first code byte. The core of the record is the machine code, with typically up to 32 bytes in each line. The final byte is known as a checksum. The checksum is calculated so that a total count of all record bytes, excluding the record start characters, will always give FF*h* (−1). This is used by the loader program to detect download errors.

S1-S9 files are suitable for processors with addresses that can be represented as a 16-bit code (four hexadecimal digits), such as the 6800 MPU. 68000 MPUs have 24-bit Address buses, and need six hexadecimal digit representations. 68020 and higher family members have 32-bit Address buses, and need eight hexadecimal representations. The S2-S8 Motorola format is similar to S1-S9 but with a six-digit address field. In a similar manner, S3-S7 format files support processors with an eight-digit address.

An assembler will be very particular that the syntax of the source code

is correct. If there are *syntax errors*[9] then an **error file** will be generated. For example, if line 6 of Program 7.1 is mistakenly entered as:

```
A_LOOP: ldb 0,y ; 6:  Add array byte element n
```

then the error file following is generated:

```
x6800 (1):
 a:average.s 7: unknown op-code ldb
 a:average.s 7: ldb not defined in file or include
 a:average.s 12: LOOP not defined in file or include
 a:average.s 7: y not defined in file or include
 a:average.s: 4 errors detected
```

Table 7.4 *The error file* average.er.

The unknown op-code `ldb` has been correctly picked up and also the attempt in line 10 to Branch to a non-existent line. It often happens that one syntax error causes a number of spurious alarms, as in this case. For example the assembler tried to find a label `ldb` elsewhere in the file as an alternative to an instruction mnemonic. Thus two errors were registered for the one syntax error.

Finally, we summarize some general information specific to this assembler as an aid to reading programs in the rest of the book:

- Number representation.

    - Hexadecimal: Denoted by a following h, e.g. 41h. Some assemblers use a $ prefix, e.g. $41.

    - Binary: Denoted by a following b, e.g. 01000001b. Some assemblers use a % prefix, e.g. %01000001.

    - Decimal: The default, but optionally followed by d, e.g. 65d.

    - Character: Denoted by surrounding single quotes, e.g. 'a'. Some assemblers use only a leading single quote, e.g. 'a.

---

[9]If the assembler announces that there are no errors then there is a tendency to think that the program will work. Unfortunately a lack of syntax errors in no way guarantees that the program will do anything of the sort!

- Label arithmetic.

  - Addition: +, e.g. LOOP+6.

  - Subtraction: –, e.g. LOOP–6.

  - Multiplication: *, e.g. NUM*6.

  - Division: /, e.g. NUM/6.

- Directives

  - .define: Associates a value with a symbol, e.g. .define NUM=3039. Some assemblers use the equate directive, e.g. NUM equ 3039.

  - .byte, .word, .double: Allocate, and optionally initialize, storage for one, two or four byte-sized objects respectively. For example, .byte 1,2,4,9,25 reserves five bytes with load-time variables as shown. The directives dc (Define Constant) and ds (Define Storage) are used by some assemblers together with a size extension, e.g. ds.w 10 to reserve ten words.

  - .org: Places the following code in memory starting from the specified address. Continues until a new .org is detected.

Examples

Example 7.1
Show how you could initialize a table of powers of 10 from 10^0 (1) to 10^4 (10,000) in program memory from C200h upwards. Each datum is to be stored as a word.

Solution
The table is shown in the source file of Program 7.2.

Program 7.2 *Table of powers of ten.*

```
POWER:   .org   0C200h    ; Table starts at C200h
         .word  1,10,100,1000,10000
```

The resulting listing file is shown in Program 7.3. Here the
`.list +.text` directive expands the listing file to show each data
element of the table in memory. Normally only the first element of
each table line is shown, as in Program 7.5. In large tables expansion
would lead to overlong listings.

Program 7.3 *The listing file output for the powers of ten array.*					
1				.list	+.text
2			POWER:	.org	0C200h ; Table starts at C200h
3	C200	0001		.word	1,10,100,1000,10000
		000A			
		0064			
		03E8			
		2710			

Notice that a decimal base was used in the source file but the
assembler has translated this into hexadecimal notation to reflect
the natural binary storage in memory. The programmer can use
the number base that is most convenient and understandable in
the source file, typically binary or decimal, and the assembler will
automatically translate as necessary.

Example 7.2
Figure 7.3 shows a typical electrocardiogram (ECG) trace that is to be
used as a reference for a subsequent analysis process. Show how you
would use the assembler to load in a 160 point byte-data array into
memory, located starting at $1000h$. This will be used by the analysis
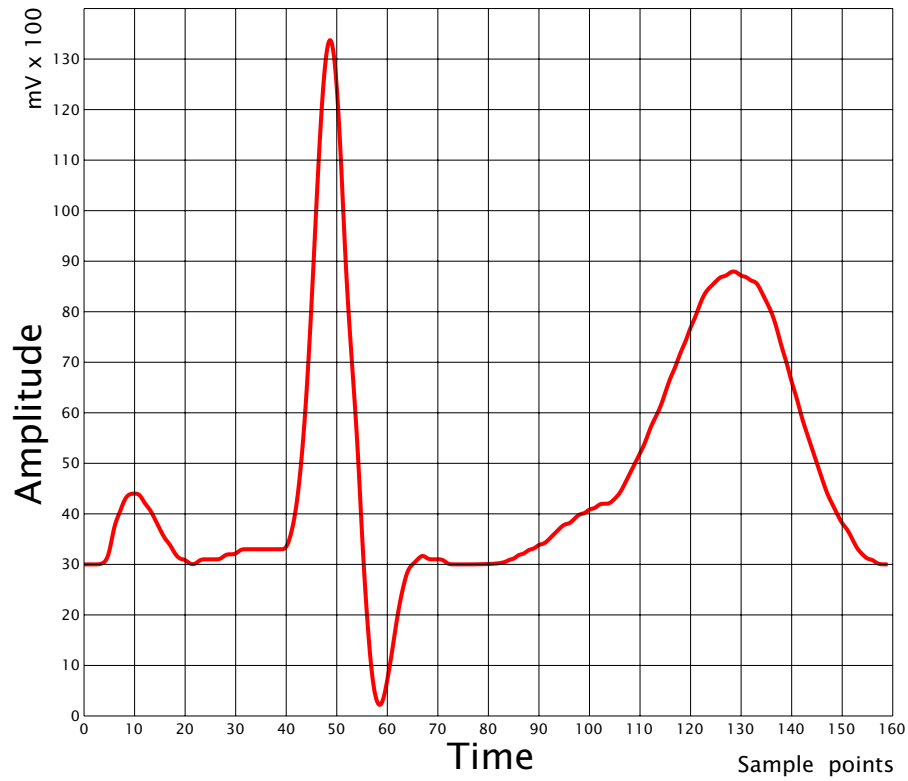software to indicate that the process can be terminated.

Figure 7.3 *A model ECG waveform.*

Solution

The solution to the problem is shown in Program 7.4. Here the directive `.byte` is used to specify that the following comma separated list of data bytes are to be placed in memory in sequence. This can be seen more clearly in the listing file of Table 7.5. A program to extract any byte from this table at random is given in Program 5.2 on page 101.

Program 7.4 *Source code for the ECG data table.*

```
; 160 magnitude samples of an ecg signal coded as bytes
.org  1000h ; Data memory
ARRAY: .byte 30,30,30,30,32,35,38,40
.byte 43,44,44,44,42,41,39,37,35
.byte 34,32,31,31,30,30,31,31
.byte 31,31,31,32,32,32,33,33
.byte 33,33,33,33,33,33,33,33
.byte 36,41,50,63,81,102,121,133
.byte 135,126,107,83,57,35,19,7
.byte 2,2,7,13,20,25,29,30
.byte 31,32,31,31,31,31,30,30
.byte 30,30,30,30,30,30,30,30
.byte 30,30,31,31,32,32,33,33
.byte 34,34,35,36,37,38,38,39
.byte 40,40,41,41,42,42,42,43
.byte 44,46,48,50,52,54,57,59
.byte 61,64,67,69,72,74,77,79
.byte 82,84,85,86,87,87,88,88
.byte 87,87,86,86,84,82,80,77
.byte 73,70,66,63,59,56,53,50
.byte 47,44,42,40,38,37,35,33
.byte 32,31,31,30,30,30,30
```

Example 7.3
Write a program in conjunction with the above table to scan the data array and determine the maximum (ECG peak) value.

Solution
Program 7.5 uses the Index register to move across the array from its initial value of ARRAY until 160 passes through the loop. Accumulator B, initially cleared, is used the hold the maximum value. Each array byte is compared with the contents of B and if it is higher (a Carry/borrow is generated) then that array value becomes the new maximum value. In this case the outcome will be $87h$ or $135d$. I have terminated the program in line 13 by a ReTurn from Subroutine (rts) instruction, thus turning the program into a subroutine.. Details relating to subroutines are the subject of the next chapter.

Program 7.5 *The listing file output for the determination of the peak of the ECG waveform.*

```
 1                            .processor   m6800
 2                            .org    0C100h      ; Program starts at C100h
 3 C100  CE1000 ECG_MAX:  ldx     #ARRAY      ; Point to the first table element
 4 C103  5F                clrb                ; Maximum value zero
 5
 6 C104  E100   ECG_LOOP: cmpb    0,x         ; Maximum - DATA[N]
 7 C106  2402              bcc     ECG_NEXT    ; IF Max Higher/Same THEN no update
 8 C108  E600              ldab    0,x         ; ELSE update maximum value
 9
10 C10A  08     ECG_NEXT: inx                 ; Point to the next array element
11 C10B  8C10A0            cpx     #ARRAY+160  ; Check to see if over the top?
12 C10E  26F4              bne     ECG_LOOP    ; IF not THEN go again
13 C110  39                rts                 ; End of program
14
15                            .org    1000h       ; Data memory
16
17 1000  1E     ARRAY:    .byte   30,30,30,30,32,35,38,40
18 1008  2B                .byte   43,44,44,44,42,41,39,37,35
19 1011  22                .byte   34,32,31,31,30,30,31,31
20 1019  1F                .byte   31,31,31,32,32,32,33,33
21 1021  21                .byte   33,33,33,33,33,33,33,33
22 1029  24                .byte   36,41,50,63,81,102,121,133
23 1031  87                .byte   135,126,107,83,57,35,19,7
24 1039  02                .byte   2,2,7,13,20,25,29,30
25 1041  1F                .byte   31,32,31,31,31,31,30,30
26 1049  1E                .byte   30,30,30,30,30,30,30,30
27 1051  1E                .byte   30,30,31,31,32,32,33,33
28 1059  22                .byte   34,34,35,36,37,38,38,39
29 1061  28                .byte   40,40,41,41,42,42,42,43
30 1069  2C                .byte   44,46,48,50,52,54,57,59
31 1071  3D                .byte   61,64,67,69,72,74,77,79
32 1079  52                .byte   82,84,85,86,87,87,88,88
33 1081  57                .byte   87,87,86,86,84,82,80,77
34 1089  49                .byte   73,70,66,63,59,56,53,50
35 1091  2F                .byte   47,44,42,40,38,37,35,33
36 1099  20                .byte   32,31,31,30,30,30,30
37                            .end
```

Example 7.4

Repeat Example 7.3 but this time calculating the average value. This average byte is to be in Accumulator B and two memory locations at $0000:1h$ can be used to hold the double-byte total sum of all data bytes. The average is $2Eh$ or $46d$.

Program 7.6 *Determining the average value for the ECG data array.*

```
        .processor  m6800
        .org    0C100h    ; Program starts at C100h
ECG_MAX: ldx    #ARRAY    ; Point to the first element of the table
        clr     SUM       ; Zero the grand total
        clr     SUM+1
ECG_LOOP: ldaa  0,x       ; Get DATA[N]
        adda    SUM+1     ; Add it to the LSB of the sum
        staa    SUM+1
        ldaa    SUM       ; Get the MSB of the sum total
        adca    #0        ; Add the previous carry to it
        staa    SUM
        inx               ; Point to the next array element
        cpx     #ARRAY+160 ; Check to see if over the top?
        bne     ECG_LOOP  ; IF not THEN go again
; Now divide by repetitive subtraction of 160 to get average
        clrb              ; Average value zeroed
AV_LOOP: ldaa   SUM+1     ; Get LSB of sum
        suba    #160      ; Take away 160
        staa    SUM+1
        ldaa    SUM       ; Get MSB of sum
        sbca    #0        ; Subtract the Carry/borrow from it
        staa    SUM
        bcs     FINISHED  ; Finish if produces a Carry/borrow
        incb              ; ELSE record a successful subtraction
        bra     AV_LOOP   ; and go do another one
FINISHED: rts            ; End of program
; ***********************************************************
        .org    1000h     ; Data memory
ARRAY:.byte    30,30,30,30,32,35,38,40
     .byte     43,44,44,44,42,41,39,37,35
     .byte     34,32,31,31,30,30,31,31
     .byte     31,31,31,32,32,32,33,33
     .byte     33,33,33,33,33,33,33,33
     .byte     36,41,50,63,81,102,121,133
     .byte     135,126,107,83,57,35,19,7
     .byte     2,2,7,13,20,25,29,30
     .byte     31,32,31,31,31,31,30,30
     .byte     30,30,30,30,30,30,30,30
     .byte     30,30,31,31,32,32,33,33
     .byte     34,34,35,36,37,38,38,39
     .byte     40,40,41,41,42,42,42,43
     .byte     44,46,48,50,52,54,57,59
     .byte     61,64,67,69,72,74,77,79
     .byte     82,84,85,86,87,87,88,88
     .byte     87,87,86,86,84,82,80,77
     .byte     73,70,66,63,59,56,53,50
     .byte     47,44,42,40,38,37,35,33
     .byte     32,31,31,30,30,30,30
SUM: .byte     [2]        ; Reserve two bytes for the sum
     .end
```

Solution
The coding in Program 7.6 is similar to that of Program 7.5 but this time the grand total of the 160 data bytes are summed. Once the array has been walked through the average is computed by dividing by 160. This is accomplished by subtracting 160 from the double-byte sum and incrementing the count until an underflow (Carry/borrow) occurs. This count is the quotient. See Program 6.1 on page 114.

Self-assessment questions

7.1   Repeat the program of Example 7.1 but with the assembler/linker that you are using for your course. Try printing out the various output files. Note the effect of deliberate syntax errors.

7.2   In Examples 7.3 and 7.4 the length of the data array was known in advance to be 160 bytes. Can you think of other ways in which an a priori knowledge could be avoided?

7.3   Repeat Example 7.3 but this time determining the minimum.

7.4   Write a program located at $C000h$ using the table of powers of ten that will convert a 2-byte natural binary number in $0030{:}1h$ to a string of BCD digits. Do this by:

1. Subtract 10,000 repetitively from the binary number incrementing the TEN_THOU memory location until underflow.

2. Restore the one 10,000 subtraction too many and then subtract 1000 repetitively from the binary number incrementing the THOU memory location until underflow.

3. Restore the one 1000 subtraction too many and then subtract 100 repetitively from the binary number incrementing the HUNDS memory location until underflow.

4. Restore the one 100 subtraction too many and then subtract 10 repetitively from the binary number incrementing the TENS memory location until underflow.

5. Restore the one 10 subtraction too many. The remainder is the units BCD digit.

Reserve five bytes in data memory for this BCD array, with the ten-thousand digit in $0000h$, thousand digit in $0001h$ etc.

Subroutines

Good software should be configured as a set of interacting modules rather than one large program working straight through from beginning to end. There are many advantages to modular programming, which is almost mandatory when code lengths exceed a few hundred lines or when a project is being developed by a team.

In the last chapter we referred to the need to link modules together in order to build up large programs. What form should should such modules take? In order to answer this question we will look at the use of program structures designed to facilitate this modular approach and the instructions associated with it.

After completing this chapter you will:

- *Appreciate the need for modular programming.*
- *Have an understanding of the structure of a stack and its use in the call–return subroutine mechanism.*
- *Understand the terms nested and recursive subroutine.*
- *Know how to use the Push and Pull instructions to move data on to and out of the stack.*
- *Understand how parameters can be passed to a subroutine, by copy or reference, and altered or returned to the caller.*
- *Be able to write a subroutine having a minimal impact on its environment.*

Take a look at the inside of your personal computer. It will probably look something like the photograph in Fig. 8.1, with a motherboard hosting the MPU, assorted memory and other support circuitry, and a variable number of expansion sockets. Into this will be plugged a disk controller

167

card and a video card. There may be others, such as a soundboard or modem. Each of these plug-in cards has a distinct and separate logical task and they interact via the services supplied by the main board — the motherboard.

There are many advantages to this **modular** construction.

- Flexibility; that is it is relatively easy to upgrade or reconfigure by adding or replacing plug-in cards.
- Can reuse from previous systems.
- Can buy in standard boards or design specialist boards in-house.
- Easy to maintain.

Of course there are a few disadvantages. A fully integrated motherboard is smaller and potentially cheaper than an equivalent mother/daughterboard configuration. It is also likely to be more reliable, as input

!!! Insert photograph 8.1 here !!!

Figure 8.1 *Modular hardware implementing a PC.*

and output signals do not have to traverse sockets/plugs. However, when they do occur, faults are often more difficult to track down and rectify.

Modular programming uses the same principle to construct "software circuits", i.e. programs. A formal definition of modular programming[1] is:

> An approach to programming in which separate logical tasks are programmed separately and joined later.

Thus to write a program in a modular fashion we need to decompose the specification into a number of stand-alone routines, each implementing a well-defined task. Such a module should be relatively short, be well documented and easy for a human, not necessarily the original programmer, to understand.

The advantages of a modular program are similar to those for modular hardware, but even more compelling:

- Each module can be tested, debugged and maintained on a stand-alone basis. This makes for overall reliability.
- Can be reused from previous projects or bought in from outside.
- Easier to update by changing modules.

Deciding how to segment a program into individual stand-alone tasks is where the real expertise lies. The actual coding of such tasks as subprograms is no different than the examples we have given in previous chapters, such as that shown in Program 7.1 on page 151. There are a few additional instructions associated with such sub-programs, and these are listed in Table 8.1. We will look at these and some useful techniques in constructing software in the remainder of the chapter.

Program modules at assembly level are universally known as **subroutines**, as they are in some high-level languages such as FORTRAN and BASIC.[2] Subroutines are the analog of hardware plug-in cards.

Consider the situation where a 1 second delay task is to be implemented. This may be needed to alert an aircraft pilot to look at the control panel warning lights for various scenarios (such as low fuel or overheating) by sounding a buzzer for a short time. In a modular program, this

---

[1]From *Chambers Science and Technology Dictionary*, Cambridge University Press, 1988.

[2]Other high-level languages use the terms function (**C** and Pascal) or procedure (Pascal).

Operation	Mnemonic	Description
Call		Transfer to subroutine
Jump to subroutine	jsr ea	Push PC on to stack, PC <- <ea>
Branch to subroutine	bsr offset	Push PC on to stack, PC <- PC+sex|offset
Return		Transfer back to caller
from subroutine	rts	Pull original PC back from Stack

Table 8.1 *Subroutine instructions.*

delay would be implemented by coding a 1 s subroutine which would be *called* by the main program as necessary. This is represented diagrammatically in Fig. 8.2.
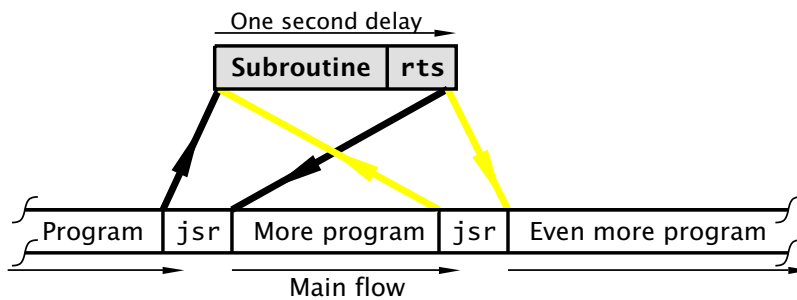
Figure 8.2 *Subroutine calling.*

In essence, calling up a subroutine involves nothing more than placing the address of the first instruction in the Program Counter (PC), that is doing a jump. Thus, if our delay subroutine were located at C100h, then jmp C100h would seem to do the trick. Of course, as we noted in the last chapter, the programmer should label the entry point, and assuming this has been done, as in Program 8.1, then we have jmp DELAY_1_S.

The problem really is how to get back again! Somehow the MPU has to remember from where in the caller program the subroutine was entered so that it can return to the *next* instruction in the caller sequence. This can be seen in the diagram, where the jumping-off point can be from *anywhere* in the main program, or indeed from another subroutine — the

latter process is called **nesting**; see Fig. 8.4.

One possibility is to place this address in a designated Address register or memory location prior to jumping off. This can then be moved back into the PC at the end of the subroutine as the return mechanism. This approach breaks down whenever one subroutine wishes to call another. Then the secondary subroutine will overwrite the return address of the first, and the main program can never be regained. To get around this problem, more than one register or memory location could be used to hold a stack of return addresses. This **last-in first-out stack** structure is shown in Fig. 8.3(a).

Consider an area of memory set aside by the programmer to store subroutine return addresses. This is called the **stack**. There is nothing special about this RAM except that the programmer must ensure that nothing else is likely to overwrite these memory locations. The address register called the **Stack Pointer** (SP) is used to point to the top of this
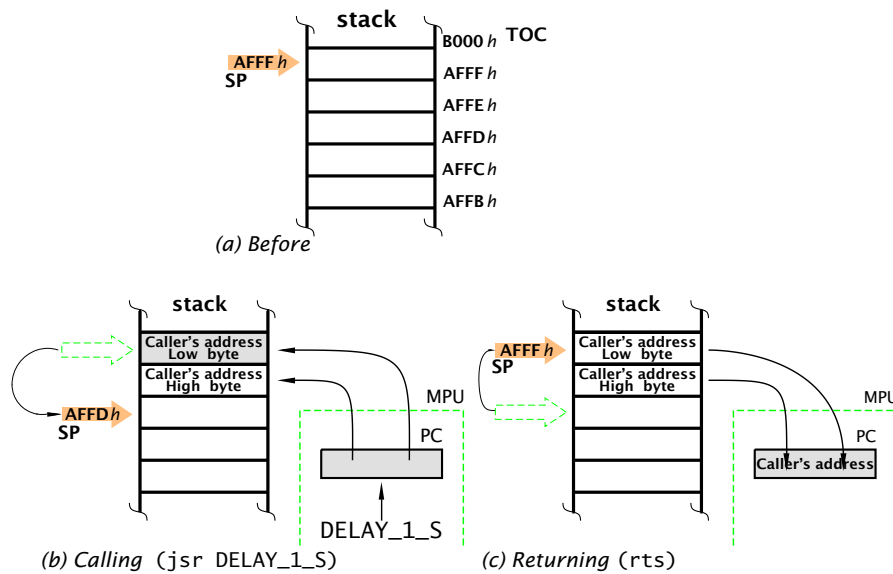
Figure 8.3 *Using a stack in memory to store return addresses.*

reserved area. In the diagram I have arbitrarily allocated RAM from AFFF*h* downwards as the stack. Thus the instruction:

```
lds #0AFFFh,a7    ; Point to top of stack
```

that is LoaD Stack pointer with the constant address AFFF*h*. will be placed somewhere near the beginning of the main program.

With the stack set up as shown in Fig. 8.3(a) the subroutine can be called using the special Jump instruction jsr (Jump to SubRoutine).[3] This instruction automatically moves the Stack Pointer down and then copies the two-byte address of the *next* instruction in the caller program (that is the contents of the Program Counter) into the stack. This process is called **pushing**. Control is transferred to the subroutine in the same way as an ordinary jmp, that is by copying the destination address into the PC. As addresses are two bytes long, two locations in the stack RAM are used for this storage, and SP will decrement by 2 during this push.

At the end of the subroutine the last instruction should be rts (ReTurn from Subroutine). This reverses the push action of jsr and **pulls** the return address back from the stack into the PC. The Stack Pointer is moved back up to the previous position automatically.

The beauty of the stack mechanism is its handling of **nested** subroutines. Consider the situation in Fig. 8.4 where the main program calls the first-level subroutine SR1 which in turn calls the second-level subroutine SR2. In order eventually to get back to the main program, the outward progression sequence must be exactly matched by the inward path. This pattern is matched by the **last-in first-out** (**LIFO**) structure of the stack mechanism, which can handle any arbitrary nesting sequence to any depth (within reason) automatically. It can even handle the (painful) situation where a subroutine calls itself! Such a subroutine is known as **recursive**. As we shall see in the next chapter, the stack mechanism is also used to handle interrupts. The technique is so useful that virtually all MPUs support subroutines in this manner.[4] Note that for clarity I have shown the stack organized as a word store, with each cell holding two bytes.

---

[3]bsr (Branch to SubRoutine) can be used if the subroutine is no more than ±128 bytes away.

[4]Some decrement the SP *before* pushing.

SR2

SR1

Main program

(a) Two-deep nesting

stack		stack		stack		stack
SR1		SR1		SR1		SR1
		SR2		SR2		SR2

SP   SP   SP   SP

(b) `jsr SR1`   *(c)* `jsr SR2`   *(d)* `rts`   *(e)* `rts`
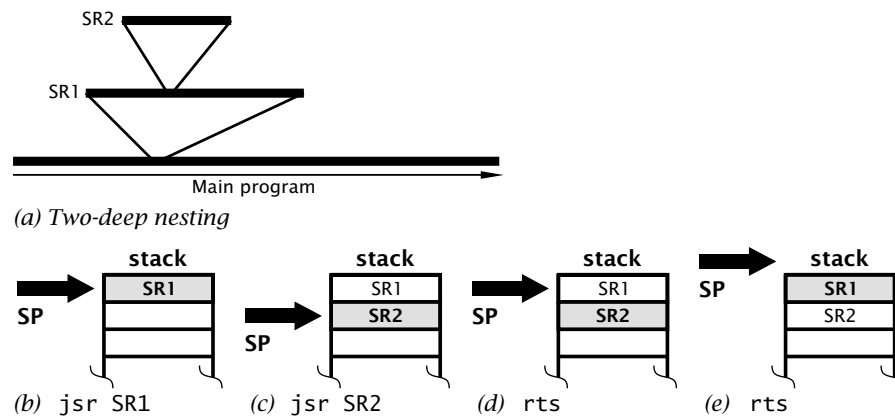
Figure 8.4 *Nested subroutines.*

If you are confused, then think of the stack mechanism as your diary, with the Stack Pointer as the bookmark. Every time you wish to use your diary to write down information, open it at the bookmark, which is set at the first blank page after your last entry, write down your entry (the return address) and close it with the bookmark advanced to the next clean page. Push data like this as many times as is necessary. Any time you wish to recall the entry, open the diary at the bookmark, move it back to the previous page and note your posting before closing it. Pull data out as many times as is necessary. You can intermingle pushes and pulls in any sequence. This is the last-in first-out structure.

Once a stack has been set up, from the programmer's perspective the following points are relevant:

- The subroutine should be called using the `jsr` (or `bsr`) instruction.
- The entry point to a subroutine should be labelled, and this label is then the name of that subroutine.
- The exit point from the subroutine should be the instruction `rts`. As a matter of style there should only be one way out (and one way in) from a subroutine.

As an example, let us code the 1 second delay subroutine. Creating a delay in software is simply a matter of doing nothing for the appropriate

duration. In Program 8.1 I have used a loop to count down from a constant n to zero. In order to calculate the value of n we need to know the processor's clock speed. Details of the number of clock cycles for each instruction/address mode combination is given in Appendix B under the columns marked with a tilde ($\tilde{}$). If the clock is, say, 1 MHz, then each cycle takes $1\,\mu$s.

Program 8.1 *A 1 second delay subroutine.*

```
;****************************************************************
; * FUNCTION: Delays by one second at 1 MHz                    *
; * ENTRY    : None                                            *
; * EXIT     : IX = 0                                          *
; ****************************************************************
;
            .define N=62499
DELAY_1_S:  ldx  #N        ; The start value,  3˜
D_LOOP:     dex            ; Decrement,        4*N˜
            nop            ; Do nothing,       2*N˜
            nop            ; Do nothing        2*N˜
            nop            ; Do nothing,       2*N˜
            nop            ; Do nothing,       2*N˜
            bne  D_LOOP    ; to zero,          4*N˜
            rts            ; and exit,         5˜
```

Each instruction in Program 8.1 is commented with the number of execution cycles, with K˜ denoting K cycles. The majority of the time is spent executing the six instructions in the loop dex:nop:nop:nop:nop:bne, which execute a total of N times. Given that the original jsr takes 9˜, then the total delay is:

Delay = (9(jsr) + 3(ldx) + 4N(dex) + 8N(nop × 4) + 4N(bne) + 5(rts)) cycles

Substituting Delay for 10^6 ($10^6\,\mu$s is 1 second) and 1 for cycles (1 MHz) gives:

$$10^6 = (9 + 3 + 5 + N \times (4 + 8 + 4))$$
$$10^6 = 16N + 17$$

$$16N = 10^6 - 17$$
$$N \approx 62,499$$

Obviously instructions outside the loop contribute very little to the overall delay in this case as N is so large and they can be ignored.[5] However, for short times the sandwich instruction delays should not be omitted from the calculation (see Program 8.6). In this case the approximate calculation is:

$$10^6 = N \times (4 + 8 + 4))$$
$$10^6 = 16N$$
$$16N = 10^6$$
$$N = 62,500$$

Notice the comment box at the head of the program. It is good practice to document your subroutine by giving a short description of what data is present at entry, at exit and any working registers or other location altered by the software. Where relevant an example should be given.

Most subroutines alter their environment to some extent. By environment is meant the state of the working registers, memory locations and flags. In our example, the value of IX will be changed to zero on exit. Although this is documented in the header comment box, the less a subroutine disturbs its environment the easier it is to use and the scope for error is correspondingly reduced.

Program 8.2 is a **transparent** version of our original program where IX is pushed into the stack on entry and pulled back on exit. The normal way to save the environment on entry to a subroutine is to push any internal register onto the stack and at the end, just before returning pull the original data back out into these registers. Newer microprocessors, such as the 6809 and the 68000 series have instructions that can push/pull any selection of registers to/from the stack in one single instruction. Unfortunately the 6800 only has `psha`, `pshb`, `pula` and `pulb` instructions, as detailed in Fig. 6.1 on page 112. There is no `pshx`/`pulx` instruction.

---

[5]The resulting factor of 1 parts in a million is rather better than the typical crystal tolerance of ±100 parts in a million!

Program 8.2 *A transparent 1 second delay subroutine.*

```
; ************************************************************
; * FUNCTION: Delays by one second at 1 MHz                 *
; * ENTRY   : None                                          *
; * EXIT    : None                                          *
; ************************************************************
            .define N=62496
            .define TEMPORARY = 0000h
; First save the old value of IX
DELAY_1_S:  psha                  ; Save Accumulator A,          4~
            stx     TEMPORARY  ; Put IX in memory,           5~
            ldaa    TEMPORARY  ; Get IXL,                    3~
            psha                  ; Push it out into the stack, 4~
            ldaa    TEMPORARY+1; Get IXH,                    3~
            psha                  ; Push it out into the stack, 4~
; Now delay
            ldx   #N             ; The start value,            3~
D_LOOP:     dex                   ; Decrement,                  4*N~
            nop                   ; Do nothing,                 2*N~
            nop                   ; Do nothing,                 2*N~
            nop                   ; Do nothing,                 2*N~
            nop                   ; Do nothing,                 2*N~
            bne   D_LOOP         ; to zero,                    4*N~
; Now retrieve the old value of IX
            pula                  ; Get IXH,                    4~
            staa    TEMPORARY    :                            4~
            pula                  ; Get IXL,                    4~
            staa    TEMPORARY+1 ;                             4~
            ldx     TEMPORARY  ; Put IXH:IXL into Index reg, 4~
            pula                  ; Finally get original A back, 4~

            rts                   ; and exit,                   5~
```

In Program 8.2 a temporary 2-byte memory location TEMPORARY is used to store the contents of the Index register on entry. After pushing the state of Accumulator A into the stack, both bytes of the original state of IX are loaded from memory and pushed into the stack. On exit the process is reversed and the original state of IX is preserved on return. *It is critically*

important that the Stack Pointer should be balanced in this way so that the return address can be picked up on exit. If the number of pushes does not equal the number of pulls, then garbage will be placed in the PC by rts and the system will die. This most often happens if the program is such that there are several pathways to the exit point, or even several exit points, and one of these omits balancing pulls. The same phenomenon can happen if there are several entry points. Thus good programming structure dictates that there should be only one way into and one way out of a subroutine.

In this case saving the state of IX in memory directly to TEMPORARY and retrieving it in the same way from absolute memory would be sufficient to give transparency. However, this method is not very flexible in general where subroutines are to be nested. Unless each subroutine uses a different memory word as a temporary store then one subroutine may overwrite another's preserved value of IX. So although in the 6800 the process using the stack is rather clumsy it is recommended.

Calculating the value N to put into IX as the countdown constant is similar to the previous case:

$$10^6 = (9 + 4 + 5 + 3 + 4 + 3 + 4 + 3 + (6 \times 4) + 5 + N \times (4 + 8 + 4))$$
$$10^6 = 16N + 64$$
$$16N = 10^6 - 64$$
$$N = 62,496$$

Once again we could ignore the sandwich around the meat (the actual loop) and use the approximate value $N = 62,500$.

Our delay program is an example of a double-void subroutine, in that no parameters (cf. signals in the hardware analog) are sent to it and nothing is returned — just the side effect of a delay.

A slightly more exciting version of our subroutine is given in Program 8.3. Here the caller determines the number of delay seconds K by 'sending' a parameter via B. Thus to call up a 1 minute delay, the caller will use the sequence:

```
ldab  #60       ; Sixty seconds in a minute = K
jsr   DELAY_K_S ; Go to it!
```

Program 8.3 *Delaying for K seconds.*

```
;*************************************************************
; * FUNCTION: Delays by K seconds at 1 MHz                  *
; * ENTRY    : K in B                                       *
; * EXIT     : None                                         *
; *************************************************************
            .define N=62496
            .define TEMPORARY = 0000h
; First save the old value of IX and B
DELAY_1_S:  psha                   ; Save Accumulator A
            stx     TEMPORARY  ; Put IX in memory
            ldaa    TEMPORARY  ; Get IXL
            psha                   ; Push it out into the stack
            ldaa    TEMPORARY+1; Get IXH
            psha                   ; Push it out into the stack
            pshb                   ; Save the seconds parameter

; Now delay
D_LOOP1:    tstb                   ; Is the parameter zero?
            beq  EXIT              ; IF it is THEN delay finished
; ELSE do one second delay
            ldx  #N                ; The start value
D_LOOP2:    dex                    ; Decrement
            nop                    ; Do nothing
            nop                    ; Do nothing
            nop                    ; Do nothing
            bne  D_LOOP            ; to zero

;           decb                   ; Decrement seconds count
            bra  D_LOOP1           ; and repeat

; Now retrieve the old value of B & IX
            pulb                   ; Retrieve old value of parameter
            pula                   ; Get IXH
            staa    TEMPORARY
            pula                   ; Get IXL
            staa    TEMPORARY+1
            ldx     TEMPORARY  ; Put IXH:IXL into Index reg
            pula                   ; Finally get original A back

            rts                    ; and exit
```

The actual routine itself is similar to the previous program, but each time the second is counted out, K in B is decremented. Thus the total delay is $K \times 1$ s.[6]

In order to make the subroutine transparent, both the Index register and Accumulator B have to be saved. Again Accumulator A is used as an intermediary and this is the first register to be pushed. In this case IX has been saved first followed by the Accumulator B byte. The state of the stack during execution of Program 8.3 illustrated in Fig. 8.5 shows this succession, with IX located first. In order to balance the stack, three pulls must be made at the end of the subroutine. The first is to restore B (as it was last in) and the next two reassembles the double-byte Index register in memory. The last pull (balancing the first push) restored the state of Accumulator A.

(a) On arrival	(b) Saving environment	(c) Retrieving environment	(d) On return
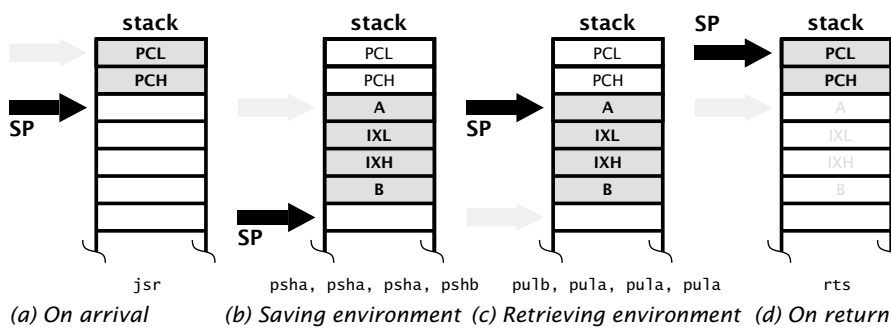
Figure 8.5 *The state of the stack in subroutine* DELAY_K_S.

In all our examples we have used accumulator registers to **pass parameters**. The Index registers can also be used to pass the address of a data structure, such as an array (see Example 8.??). The 6800 family have few internal registers that can be used to pass parameters to a subroutine, unlike MPUs such as the 68000 which has eight Data and eight Address registers. In the majority of cases where the 6800 is concerned other approaches using external memory must be used.

One possibility is to use a fixed area of RAM as a passing ground (some-

---

[6]Ignoring the small fixed time overheads.

times called a heap), where data can be copied for transmission. This can also be used as a working area, where intermediate data generated by the subroutine can be temporarily stored. The problem here is that if all subroutines share the same block, then unintended interactions can occur. This is contrary to our stipulation that subroutines should be stand-alone with a minimal interaction with the general environment.

To avoid such interaction, each subroutine could be given its own private block of RAM, from which other software is banned. If there are many subroutines, then this technique is rather extravagant in memory usage.

A alternative approach is to use the stack to pass data to and fro, which can be accessed using the Index register to point to the appropriate data. Furthermore, once in a subroutine, the Stack Pointer can be moved down to open a "hole" (known as a frame) in memory for local storage. On exit this frame can be closed up, and this conforms to the privacy stipulation. Newer MPUs, such as the 6809, can use a form of Indexed addressing mode where the Stack Pointer is the Index register. This makes accessing data on a random basis from the stack a relatively efficient process. High-level languages, such as **C** make extensive use of this technique. The 6800 is deficient in this area and is rarely used to execute high-level languages.

The previous example was still void in that no data was returned to the caller on exit. For our next example we will code a subroutine that will evaluate the square root of an integer n passed to the subroutine in A which is returned in B.

The crudest way of doing this is to try every possible integer k from 1 upwards, generating k^2 by multiplication and checking that the outcome is no more than n. A slightly more sophisticated approach is based on the relationship:

$$k^2 = \sum_{i=0}^{k} (2 \times i) + 1$$

On this basis a possible structure for this function is:

1. Zero the loop count k
2. Set variable i to 1
3. DO forever:
    - Take i from $number$
    - IF the outcome is under zero THEN BREAK out

- ELSE add 2 to i
- Increment the loop count k
- REPEAT loop

4. RETURN loop count k as $\sqrt{number}$

That is sequentially subtract the series 1,3,5,7,9,11…from $number$ until underflow occurs; with the tally of successful passes being the square root. An example giving $\sqrt{65} = 8$ is given in Fig. 8.6(a) using this series approach. A flowchart visualizing the task list is also given in Fig. 8.6(b).

The software listed in Program 8.4 closely tracks the flowchart. B is used as a working register to hold the 'magic' number i (which is incremented by 2 in line 15). The original value of the number in Accumulator A is saved in line 10 and restored from the stack on exit in line 19.

(a) An example   (b) Flowchart of the process
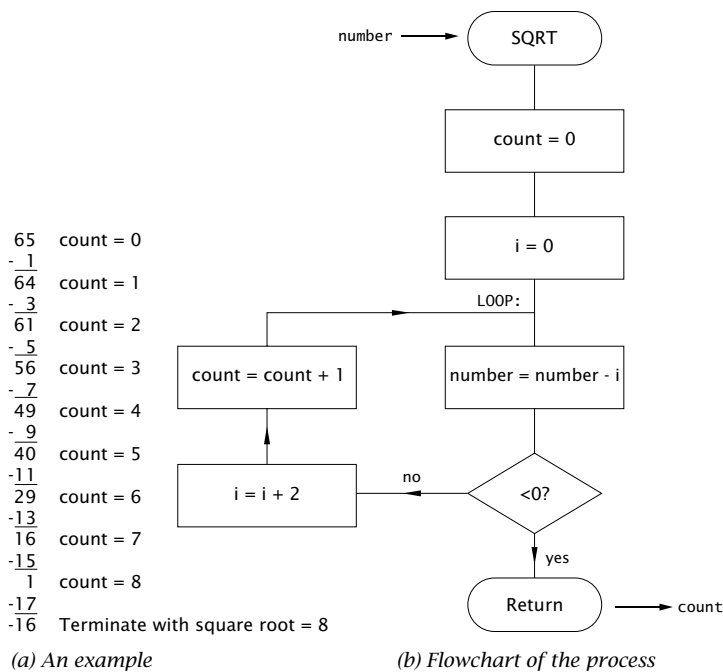
Figure 8.6 *Finding the square root of an integer.*

Program 8.4 *Coding the square root subroutine.*

```
        .processor m6800
; ************************************************************
; * FUNCTION: Evaluates the square root of an integer number  *
; * EXAMPLE : 65 on entry returns 8                           *
; * ENTRY   : number is passed in A                           *
; * EXIT    : No other register is altered                    *
; * EXIT    : Root is returned in B                           *
; ************************************************************
;
        .define  COUNT = 0030h  ; Use memory @ 0030h for count
        .org     0C100h
SQRT:   psha                    ; Protect the environment
        clr      COUNT          ; count = 00
        ldab     #1             ; i = 1

S_LOOP: sba                     ; number = number - i
        bcs      S_EXIT         ; Breakout if underflow (borrow)
        addb     #2             ; ELSE i = i + 2
        inc      COUNT          ; count = count + 1
        bra      S_LOOP         ; and repeat

S_EXIT: ldab     COUNT          ; Get COUNT, which is the sqr root
        pula                    ; Retrieve old value of A
        rts                     ; Return with root = count in B
        .end
```

The core of the program is unexceptional. Notice how the `bcs` instruction — which some assemblers allow the alternative more meaningful mnemonic `blo` — is used after the $number - i$ operation to detect underflow to below zero (which sets the Carry/borrow flag) and exit the loop.

Memory location $0030h$ is used as a straight counter to indicate the number of successful subtractions, and as such gives the answer ready to return to the caller. The largest possible outcome is $\sqrt{255} = 15$, which can easily be accommodated in the byte-sized Accumulator B.

Incidentally, it really is not necessary to keep a count of the number of successful subtractions as $i = (2 \times count) + 1$. Thus the square root can be deduced by keeping i in B and on exit shifting right once. This

divides by 2 and by throwing away the 1 that pops out into the carry, effectively predecrements by 1 (i is always odd and so its least significant bit is always 1). Try coding this alternative arrangement.

Examples

Example 8.1
Code a subroutine that will calculate the 8-bit checksum of the 256 bytes in memory between $0000h$ and $00FFh$. The checksum here is defined as the complement of the modulo-256 sum of all the bytes.

Solution
Firstly you must be able to figure out what exactly you are being asked to do. In essence you have to add together all 256 bytes in memory located between the two specified memory locations. This addition is to be with a resolution of eight bits. That means that any carry from an addition is to be ignored. This is rather like adding several 2-digit decimal numbers and ignoring any hundreds that may be generated after each add.

   The fact that you must deal with an array of data should alert you to the fact that you will have to use the Index register as a pointer, initialised to $0000h$ and incrementing until the byte in memory at $00FFh$ has been dealt with. You can use Accumulator B to hold the sum as this process walks through the array. This gives us the task list:

   1. Clear the sum.
   2. Initialise the Index register to point to the first byte.
   3. WHILE the pointer is not over the top:
       - Add byte n to the byte sum.
       - Increment the pointer, that is increment n.
   4. Complement the byte total sum.

   The coding shown in Program 8.5 follows the task fairly closely. As Accumulator B is eight bits wide, continually adding the pointed-to bytes will automatically give a modulo-256 ($2^8 = 256$) sum.

Program 8.5 *Generating a checksum.*

```
C_SUM:   clrb          ; Zero the byte sum
         ldx  #0000   ; Point to the start of the array @ 0000h

LOOP:    addb 0,x      ; Add the byte in memory @ where IX points to
         inx           ; Advance the pointer
         cpx  #0100h  ; Over the top? (00FF + 1 = 0100h)
         bne  LOOP     ; IF not THEN add the next byte

         comb          ; Complement the byte total sum
         rts           ; and return from subroutine
```

Incrementing IX on each pass through the loop will eventually give a value above the address of the top array byte. At this point the loop is exited and the subroutine exited after the sum is complemented as specified.

Example 8.2
Write a subroutine to give a fixed 208 μs delay. You may assume a 1 MHz processor clock rate which will give an execution cycle time of 1 μs.

Solution
The solution shown in Program 8.6 is similar to that in Program 8.2, but as the delay is so short, the surrounding instructions and jsr need to be accounted for. In addition Accumulator A is used as the delay counter and no nop instructions used to pad out the execution time. The delay calculation is:

Total delay is $9 + 4 + 2 + 2N + 4N + 4 + 5 \, \mu s$

$$24 + 6N = 208$$
$$6N = 184$$
$$N \approx 30$$

Program 8.6 *A transparent 208 μs delay subroutine.*

```
; ************************************************************
; * FUNCTION: Delays by 208 microseconds at 1 MHz           *
; * ENTRY    : None                                         *
; * EXIT     : None                                         *
; ************************************************************
            .define N=31
DELAY_1_S:  psha                ; Save Accumulator A,         4˜

; Now delay
            ldaa  #N            ; The start value,            2˜
D_LOOP:     deca                ; Decrement,                  2*N˜
            bne  D_LOOP         ; to zero,                    4*N˜
; Now retrieve the old value of A
            pula                ;                             4˜

            rts                 ; and exit,                   5˜
```

This program will be $4\,\mu$s too short using a value of $N = 30$. Adding two nops, each with a delay of $2\,\mu$s, outside the loop would tune the delay to the desired value of $208\,\mu$s.

Example 8.3

The majority of digital electronic displays are based on a selective activation of seven segments[7] in the manner shown in Fig. 8.7. Write a transparent subroutine that will accept a four-bit binary coded decimal nybble n ($0000 - 1001b$) in Accumulator A and exit with the listed seven-segment code. No other registers are to be altered on exit. Interface details for seven-segment displays are given in Fig. ?? on page ??.

Solution

The easiest way of implementing this task is to store the ten seven-segment codes as a look-up table, in the manner of the electrocardio-gram of Program 7.4 on page 162. As the table comprises ten bytes,

---

[7] Just look at your digital watch.

11000000 11111001 10100100 10110000 10011001 10010010 10000010 11111000 10000000 10010000
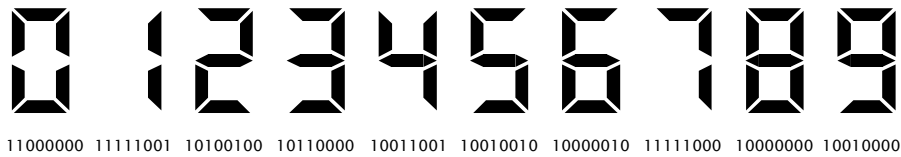
Figure 8.7 *The seven-segment BCD font.*

the `.byte` directive is used in Program 8.7 following the executable code.

The program itself follows the following coding task list.

1. Save the IX and A registers.

2. Add the byte number n to the base address of the table to point to entry n.

3. Extract table entry n into Accumulator B.

4. Restore the original value of IX and Accumulator A.

5. Return.

as delimited by the appropriate comments.

In order to add n to the value of the Index register a word in memory is used as a temporary 2-byte store, TEMP2:TEMP2+1 (at 0032:3h. With the table base address put into IX (`ldx #TABLE`) it is transferred into memory (`stx TEMP2`) and then the contents of Accumulator A added to the double-byte number in memory in the usual way (see Program 3.7 on page 74). With this done the address TABLE + n is moved back into IX which then points to entry n in the table. As we are using temporary memory storage for this addition I have also used a double-byte location to temporarily store the original value of IX for later retrieval.[8]

Example 8.4

Write a program to flash a light-emitting diode (LED) five times at 1-second intervals. This will involve:

---

[8]Random table retrieval is horribly complicated with the 6800 MPU. The 6809 MPU tidied up this function by allowing an accumulator to be an offset for Indexed addressing. The program then would be `pshs X` — `ldx #TABLE` — `ldab a,x` — `puls x` — `rts`.

Program 8.7 *The seven-segment decoder.*

```
; *************************************************************
; * FUNCTION: Converts from BCD binary n to                   *
; * FUNCTION: BCD active-low 7-segment code                   *
; * ENTRY   : Binary in A, Table of ten 7-segments following  *
; * EXIT    : 7-segment code in B, A and IX unchanged         *
; *************************************************************
          .define TEMP1 = 0030h, TEMP2 = 0032h
; Task 1: Save the IX and A registers
SVN_SEG: psha             ; Save original contents of A
         stx   TEMP1       ; Put original value away for safekeeping
; Task 2: Add the BCD number n to the table base address
         ldx   #TABLE      ; Point IX to table below
         stx   TEMP2       ; Put into memory
         ldab  TEMP2+1     ; Get LSB of table bottom
         aba               ; Add the value n
         staa  TEMP2+1     ; and put in memory
         ldab  TEMP2       ; Get MSB of table bottom
         adcb  #0          ; Add the carry bit
         stab  TEMP2       ; put away in memory
         ldx   TEMP2       ; and back into IX
; The value of IX is now #TABLE + n
; Task 3: Extract table entry n
         ldab  0,x         ; Copy entry at TABLE+n to B
; Task 4: Restore the original values of IX and A
         ldx   TEMP1       ; Restore original value of IX
         pula              ; and the original value of A
; Task 5: Return
         rts               ; Return with the goodies in B

; This is the table of seven-segment codes
TABLE:   .byte 11000000b, 11111001b, 10100100b, 10110000b,
               10011001b, 10010010b, 10000010b, 11111000b,
               10000000b, 10010000b
```

1.Writing a 0.5-second delay subroutine.

2.Writing a main routine calling up the delay routine the appro-

priate number of times and controlling the LED.

You may assume that the LED is turned on by accessing memory location 9006*h* and off by accessing 9007*h*. The microprocessor is running at a clock rate of 1 MHz.

Solution

The first part of this specification is similar to that of Program 8.1 but as the delay is somewhat shorter the nops are not required. Given that the original jsr takes 9˜, then the total delay is:

Program 8.8 *A 0.5 second delay subroutine.*

```
;*************************************************************
; * FUNCTION: Delays by 0.5 second at 1 MHz                 *
; * ENTRY   : None                                          *
; * EXIT    : IX = 0                                        *
; *************************************************************
              .define N=62499
DELAY_500_MS: ldx  #N        ; The start value,   3˜
D_LOOP:       dex            ; Decrement,         4*N˜
              bne  D_LOOP    ; to zero,           4*N˜
              rts            ; and exit,          5˜
```

Delay = (9(jsr) + 3(ldx) + 4N(dex) + 4N(bne) + 5(rts)) cycles

Substituting Delay for 0.5×10^6 (0.5 second) and 1 for cycles (1 MHz) gives:

$$
\begin{aligned}
0.5 \times 10^6 &= (9 + 3 + 5 + N \times (4 + 4)) \\
0.5 \times 10^6 &= 8N + 17 \\
8N &= 0.5 \times 10^6 - 17 \\
N &\approx 62,498
\end{aligned}
$$

The second part of the specification involves writing the main routine that calls this subroutine ten times in all. Five of these calls

follow when the LED is illuminated and five when the LED is turned off. The coding in Program 8.9 uses a loop, with Accumulator B holding the flash count from five down to zero.

Program 8.9 *Five 1-second flashes.*

```
      .org  0C000h

MAIN: ldab  #5           ; Hold the flash count
FLOOP: clr  9006h        ; LED on
      jsr   DELAY_500_MS ; for 0.5 seconds
      clr   9007h        ; LED off
      jsr   DELAY_500_MS ; for 0.5 seconds

      decb               ; Decrement count
      bne   FLOOP        ; and repeat five times
      ....  ....
```

Example 8.5
The binary approximation to the fraction $\frac{1}{3}$ is:

$$\frac{1}{3} = \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} - \frac{1}{64} + \frac{1}{128} \cdots$$

Using this series, write a subroutine that will divide a byte in Accumulator B by three with the quotient being returned in Accumulator A. The outcome up to $\frac{1}{128}$ is 0.3359375, which is within 0.78% of the exact value. With an 8-bit datum there is no point in including any further elements in the series.

Solution
The fractions $\frac{1}{2}$, $\frac{1}{4}$ etc. can be easily generated by shifting right. The coding listed in Program 8.10 simply repetitively shifts the number in situ in Accumulator B right. With each shift the outcome is either added to or subtracted from the sum kept in an initially cleared Accumulator A.

```
                    Program 8.10 Dividing by three
DIV_3: clra       ; Zero the outcome
       lsrb       ; N/2
       aba        ; Q = N*(1/2)
       lsrb       ; N/4
       sba        ; Q = N*(1/2-1/4)
       lsrb       ; N/8
       aba        ; Q = N*(1/2-1/4+1/8)
       lsrb       ; N/16
       sba        ; Q = N*(1/2-1/4+1/8-1/16)
       lsrb       ; N/32
       aba        ; Q = N*(1/2-1/4+1/8-1/16+1/32)
       lsrb       ; N/64
       sba        ; Q = N*(1/2-1/4+1/8-1/16+1/32-1/64)
       lsrb       ; N/128
       aba        ; Q = N*(1/2-1/4+1/8-1/16+1/32-1/64+1/128)
       rts
```

Self-assessment questions

8.1 A frequent mistake made by students in code such as Program 8.1, is to write bne DELAY_1_S. What would happen in this situation?

8.2 Alter Program 8.2 to give a variable delay of $\frac{1}{n}$ s, where n is a parameter passed in B.

8.3 Using the routine of Program 6.7 on page 142, write a subroutine to update a pseudo-random number located in RAM at a location which is pointed to by IX on entry, with the next in the sequence being placed there on exit. The subroutine should be transparent.

8.4 Modify the your solution to the previous problem to return a random number in B between 1 and 6, for a game of dice. Note: dividing a number by six gives a remainder of between 0 and 5. Adding one moves the range to 1 – 6. Thus update the random number and divide a *copy* of the number by six. Increment to give the required

range.

8.5 As part of an operating system a subroutine is to be written to test memory from $0000h$ through $1FFFh$. The technique is to store the code $01010101b$ ($55h$) in each memory location and check that the contents are in fact as they should be. If the test is successful then the value $00h$ should be in Accumulator A on return otherwise the test should abort with FFh returned in A and with the Index register pointed to the memory location which is faulty. Assume that the subroutine starts at $C200h$.

8.6 Parity is a method of error protection whereby each byte of data has the most significant bit set in such a way to ensure that the overall number of bits in the byte is odd or even. Write a subroutine that will convert all bytes between $0030h$ and $00FFh$ to odd 1's parity. You may assume that the existing patterns have a 0 in the most significant bit position 7.

   The task list is:

   1. Set pointer to $0030h$.

   2. WHILE pointer is less than $1000h$ DO:

      (a) Count up the number of bits in the pointed-to byte (see Program 6.3 on page 124).

      (b) Determine if this number is odd or even. The weight of the least significant bit is one (2^0), whilst all other bits are even (eg. 2, 4, 8…). Thus an odd binary number always has bit 0 = 1.

      (c) IF even THEN set bit 7 of the pointed-to number to 1 ELSE leave bit 7 at 0.

   3. Increment pointer.

8.7 Write a transparent subroutine to generate the appropriate pattern on an array of active-low light-emitting diodes to simulate an electronic game die (although I have shown seven bits corresponding to the seven 'pips', a close look at the patterns shows that this could be reduced to only four). The 'throw' enters the subroutine as a number between 1 and 6 (which is the outcome of a random-number

generator such as outlined in Example 8.4) in A and the die pattern is available on exit in B. Hint: see Example 8.3.

Throw	gfedcba		
1	0111111	⚀	3F h
2	1110110	⚁	76 h
3	0110110	⚂	36 h
4	1100100	⚃	64 h
5	0100100	⚄	24 h
6	1000000	⚅	40 h

Throw (D0.B) → Subroutine DIE → Die pattern (D7.B)
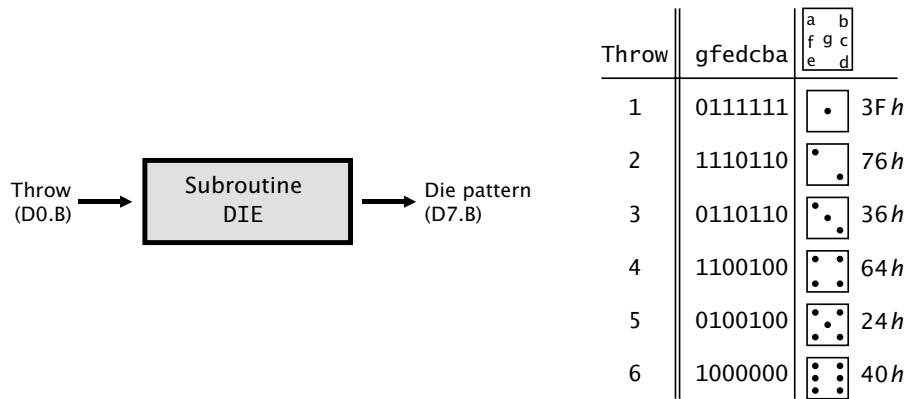
Figure 8.8 *The active-low die patterns.*

8.8 Interfacing digital electronics to the analog world invariably introduces noise into the signal, even if there was none there before. One of the simplest filtering algorithms to enhance the signal to noise ratio is digital smoothing. This technique involves generating a composite value in which each point is replaced by an average of itself with its nearest neighbours, i.e. pre samples.

The waveform shown to the left of Fig. 8.9 has a noise blip situated at its 5th sample ($n = 5$). The smoothed version is the equivalent with each point generated according to the formula:

$$F(n) = (0.25)A_{n-2} + (0.5)A_{n-1} + (0.25)A_n$$

Write a subroutine reading one sample from a A/D converter located at $9004h$ and returning with the composite value. You may use memory locations $0030h$, $0031h$ and $0032h$ to store the current and two last samples. These memory bytes will have to be updated on each call to the subroutine. The filtered value is to be returned in

Accumulator A. Hint: Use the Index register to point to the bottom
of the 3-byte data array.
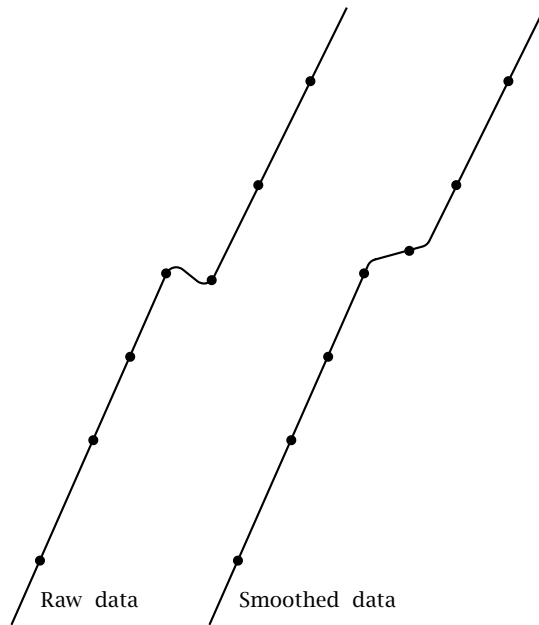
Figure 8.9 *Three-point smoothing.*

CHAPTER 10
The real world
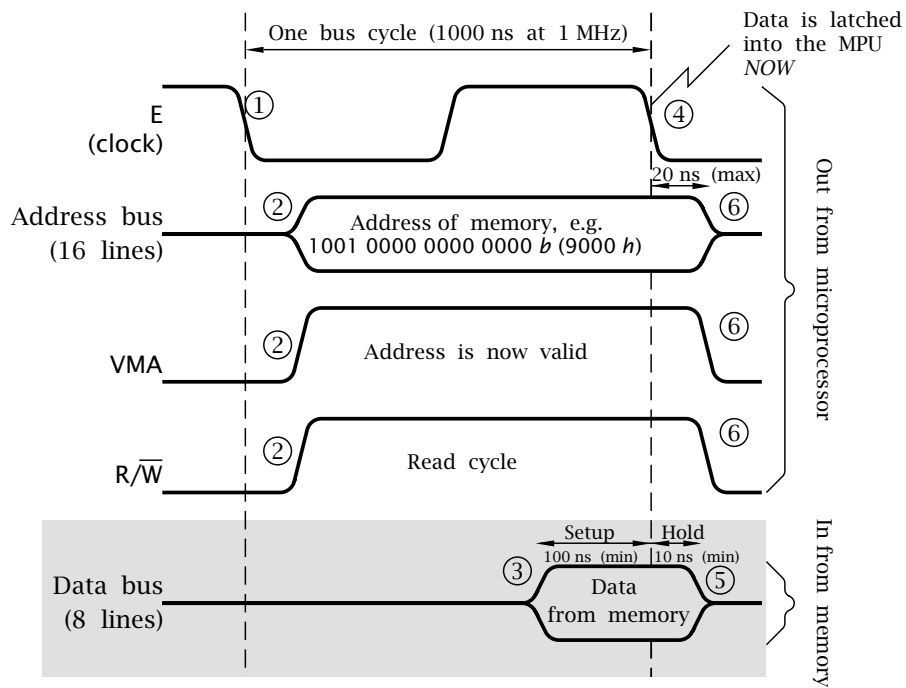
Figure 10.1 *The Read cycle during execution of the instruction ldaa 9000h.*
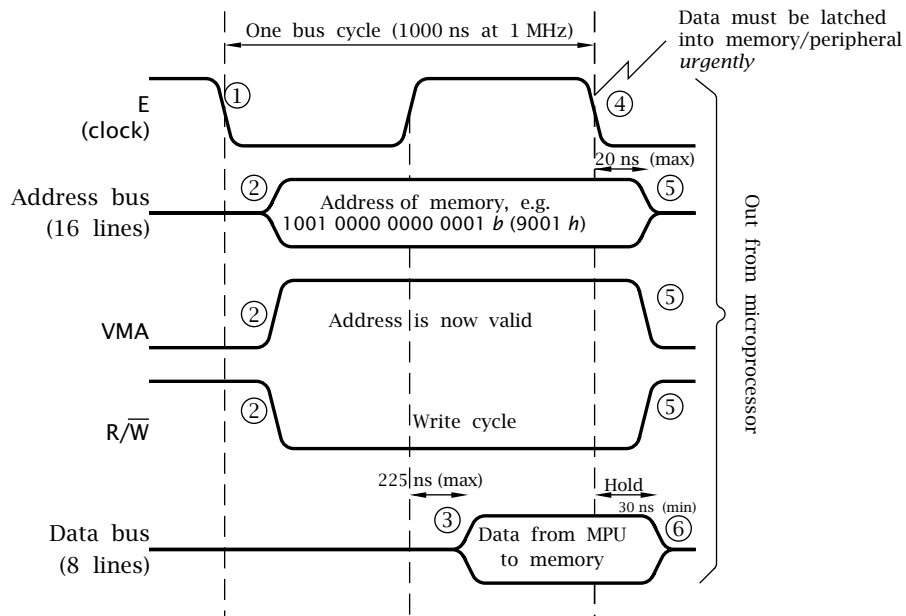
Figure 10.2 *The Write cycle during execution of the instruction* `staa 9001h`.

Glossary
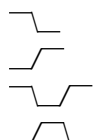
⌐_	A negative-going edge.
_⌐	A positive-going edge.
⌐_⌐	An active-low pulse.
_⌐_	An active-high pulse.
2's complement	A method of representing negative numbers. The number is changed to the opposite sign by inverting all bits and adding 1.
A	This 8-bit Accumulator register is primarily used as one of two working register for the ALU.
Address	A reference of the location of data in memory or within I/O space.
a$_n$	Address bus line n. The 68000 family mainly have 32 address lines and thus can access $2^{32} = 4,294,967,296$ or 4 Gbytes of memory/peripheral input/output.
A/D converter	Analog to Digital converter. Converts an analog signal, continually variable between an upper and lower level, to an n-bit digital equivalent.
Address mode	The technique an instruction uses to pin-point where in memory an operand lies.
ALU	Arithmetic Logic Unit. The digital circuitry that implements the fundamental operations, such as add, subtract, AND, OR, NOT.
ANSI	American National Standard Institute.
ASCII	American Standard Code for Information Interchange. An early and nearly universal standard equating a range of letters, numbers, punctuation and control character mapped

	on to a seven-bit binary code. It has been extended to various eight-bit supersets and to 16-bit Unicode as used in Microsoft's Windows 95.
B	This 8-bit Accumulator register is primarily used as one of two a working registers for the ALU.
BCD	Binary Coded Decimal. A hybrid decimal/binary coding technique whereby each digit of a decimal number is replaced one of ten binary patterns. Where this code is the normal 8-4-2-1 arrangement, the term natural BCD is sometimes used.
Binary	A number system using a base of 2.
Bit	Binary digIT. A physical variable, such as voltage or light, having two states.
Byte	Eight-bit binary word, giving $2^8 = 256$ unique combinations.
C	Carry flag. Doubles as a borrow indicator for Subtraction and Comparison operations.
CCR	Code Condition Register. Holds the flags and Interrupt mask bit in the order **H**, **I**, **N**, **Z**, **V**, **C**.
CMOS	Complimentary Metal-Oxide Semiconductor. A fabrication technique using both N- and P-channel field-effect transistors.
CPU	Central Processing Unit. The component of a computer that controls the interpretation and execution of instructions.
$\overline{\text{CS}}$	Active-low Chip Select signal. The standard designation on memory and peripheral devices indicating input(s) which must be active to enable that chip.
D/A converter	Digital to Analog converter. Converts an n-bit digital word to its analog equivalent.
d_n	Data bus line n. The main data flow between the MPU and the outside world is along this common 8-bit Data highway.
ea	Effective Address. The calculated source or/and destination address according to the address mode used.
EPROM	Erasable Programmable Read-Only Memory. A PROM that can be erased under high-intensity ultra-violet light, then reprogrammed. One-time programmable (OTP) versions without the quartz window are available.

G	Giga, a prefix indicating a billion. Specifically in binary systems $2^{30} = 1,073,741,824$.
H	Half-carry flag. The carry between bit 3 and bit 4 of a byte. This is useful where the byte is representing two BCD digits, and then stands for the carry from the lower to the upper BCD digit. Only the Add instructions activate this flag.
Handshake	The protocol used to set up, sequence and terminate a flow of data between two or more peripheral devices and a controller.
Hexadecimal	Pertaining to a number system with a base of 16. Usually used a shorthand representation for binary numbers grouped in four digits.
I	The Interrupt mask bit in the CCR. When 1 $\overline{\text{IRQ}}$ requests are ignored.
IC	Integrated Circuit. An electronic circuit fabricated on a semiconductor material, typically silicon.
IEC	International Electrotechnical Commission
$\overline{\text{IRQ}}$	Interrupt ReQuest line used to request a maskable interrupt service.
IX	The 16-bit Index register is primarily used to hold an address to point to a datum byte in memory using the Indexed address mode.
Interrupt	A signal that when activated causes the MPU to transfer program control to a particular software module called an ISR.
I/O port	An input or/and output connection providing for data communication between MPU and a peripheral device.
ISR	Interrupt Service Routine. The subroutine entered via an interrupt request or other exception. It must be terminated with an `rti` instruction rather than an `rts`.
LIFO	Last-In First-Out store, known as a push-down stack.
K	Kilo, a prefix indicating a thousand. Specifically in binary systems $2^{10} = 1,024$.
$\overline{\text{LDS}}$	Active-low Lower Data Strobe status signal. Indicates whenever the data on the lower byte of the Data bus is valid (see also $\overline{\text{UDS}}$).

LED	Light-Emitting Diode.
LSB	Least Significant (rightmost) Bit or Byte.
LSI	Large-Scale Integration. Describing an IC with between 100 and 1000 gate complexity (see also SSI, MSI and VLSI).
LSD	Least Significant Digit (typically of a decimal or BCD number).
M	Mega, a prefix indicating a million. Specifically in binary systems $2^{20} = 1,048,576$.
MCU	MicroController Unit. A microprocessor integrated on the same chip as support circuitry such as memory, I/O ports and timers.
MPU	MicroProcessor Unit. The ALU and control elements of a computer-like processor integrated on the one IC.
ms	Millisecond (10^{-3} s).
MSB	Most Significant (leftmost) Bit or Byte.
MSD	Most Significant Digit (typically of a decimal or BCD number).
MSI	Medium-Scale Integration. Describing an IC with between 12 and 100 gate complexity, e.g. a decoder (see also SSI, LSI and VLSI).
N	Negative flag. Reflects the state of the most significant bit after an instruction (see also Sign bit).
$\overline{\text{NMI}}$	Non-Maskable Interrupt line used to request a non-maskable interrupt service.
ns	Nanosecond (10^{-9} s).
Nybble	Four-bit binary word, giving $2^4 = 16$ unique combinations.
$\overline{\text{OE}}$	Active-low Output Enable signal, usually pertaining to the three-state output buffers in a memory or other peripheral input port.
OS	Operating System. Software that controls the execution of a computer system that links the hardware environment to the user program and may provide facilities such as debugging and multitasking.
PC	Program Counter. Instruction pointer to the instruction being fetched from memory.

PC	Personal Computer.
PROM	Programmable Read-Only Memory. Generic term for a memory chip that can be programmed once or relatively few times, usually before insertion into the circuit. typically holds program and fixed data in embedded microprocessor systems (see also EPROM).
RAM	Random Access Memory. Memory that is written into and read from in circuit, in which any location may be accessed with the same time delay. Typically holds temporary data and the stack.
Register	An array of flip flops or latches normally holding a single word in the CPU.
RTL	Register Transfer Language. A notation describing the operation of an instruction viewed from the perspective of moving data between registers and/or memory.
R/$\overline{\text{W}}$	The Read/Write status signal from the microprocessor to memory and other circuitry, giving the direction of transfer of data along the Data bus.
Sign bit	The MSB of a signed word, usually 1 for negative.
Stack	A last-in first-out data structure in memory used in conjunction with the Stack Pointer to hold the return address for subroutines, the processor state for interrupts and to hold accumulator register data temporarily using the `psh` and `pul` instructions to push and pull data into and out of the structure.
SP	Stack Pointer. The address register automatically used to point to the current byte in the stack.
SSI	Small-Scale Integration. Describing a simple IC with typically a few gates' complexity (see also MSI, LSI and VLSI).
TTL	Transistor Transistor Logic family. A common bi-polar circuit implementation largely coonfined to SSI and MSI logic circuits. The voltage and current levels are a *de facto* standard in logic circuits of any implementation type.
μ s	Microsecond (10^{-6} s).
VLSI	Very Large-Scale Integration. Describing an IC with a complexity of over 1000 gates, such as a memory (see also SSI, MSI and LSI).

VMA Valid Memory Access status line indicating that the pattern on the Address bus is valid.

Z The Zero flag. Set when the oucome of an instruction execution is zero.

Appendix B

6800 instruction set

Acc. & memory Operation	Mnemonic	Immed OP	~	#	Direct OP	~	#	Index OP	~	#	Extend OP	~	#	Inher OP	~	#	H	N	Z	V	C	Description
ADD	adda	8B	2	2	9B	3	2	AB	5	2	BB	4	3				√	√	√	√	√	[A] <- [A] + [M]
	addb	CB	2	2	DB	3	2	EB	5	2	FB	4	3				√	√	√	√	√	[B] <- [B] + [M]
Add accums	aba													1B	2	1	√	√	√	√	√	[A] <- [A] + [B]
ADd with Carry	adca	89	2	2	99	3	2	A9	5	2	B9	4	3				√	√	√	√	√	[A] <- [A] + [M] + C
	adcb	C9	2	2	D9	3	2	E9	5	2	F9	4	3				√	√	√	√	√	[B] <- [B] + [M] + C
AND	anda	84	2	2	94	3	2	A4	5	2	B4	4	3				•	√	√	0	•	[A] <- [A] · [M]
	andb	C4	2	2	D4	3	2	E4	5	2	F4	4	3				•	√	√	0	•	[B] <- [B] · [M]
BIT test	bita	85	2	2	95	3	2	A5	5	2	B5	4	3				•	√	√	0	•	[A] · [M]
	bitb	C4	2	2	D4	3	2	E4	5	2	F4	4	3				•	√	√	0	•	[B] · [M]
CLeaR	clr							6F	7	2	7F	6	3				•	0	1	0	0	[M] <- #00
	clra													4F	2	1	•	0	1	0	0	[A] <- #00
	clrb													5F	2	1	•	0	1	0	0	[M] <- #00
CoMPare	cmpa	81	2	2	91	3	2	A1	5	2	B1	4	3				•	√	√	√	√	[A] - [M]
	cmpb	C1	2	2	D1	3	2	E1	5	2	F1	4	3				•	√	√	√	√	[B] - [M]
Compare accums	cba													11	2	1	•	√	√	√	√	[A] - [B]
COMplement (1's)	com							63	7	2	73	6	3				•	√	√	0	1	[M] <- [M̄]
	coma													43	2	1	•	√	√	0	1	[A] <- [Ā]
	comb													53	2	1	•	√	√	0	1	[B] <- [B̄]
Complement (2's)	neg							60	7	2	70	6	3				•	√	√	1	[2]	[M] <- #00 - [M]
	nega													40	2	1	•	√	√	1	[2]	[A] <- #00 - [A]
	negb													50	2	1	•	√	√	1	[2]	[B] <- #00 - [B]
Decimal Adjust A	daa													19	2	1	•	√	√	√	[3]	Adjusts sum of BCD bytes to BCD formatted byte
DECrement	dec							6A	7	2	7A	6	3				•	√	√	[4]	•	[M] <- [M] - #1
	deca													4A	2	1	•	√	√	[4]	•	[A] <- [A] - #1
	decb													5A	2	1	•	√	√	[4]	•	[B] <- [B] - #1
Exclusive OR	eora	88	2	2	98	3	2	AB	5	2	BB	4	3				•	√	√	0	•	[A] <- [A] ⊕ [M]
	eorb	C8	2	2	D8	3	2	E8	5	2	F8	4	3				•	√	√	0	•	[B] <- [B] ⊕ [M]
INCrement	inc							6C	7	2	7C	6	3				•	√	√	[5]	•	[M] <- [M] + #1
	inca													4C	2	1	•	√	√	[5]	•	[A] <- [A] + #1
	incb													5C	2	1	•	√	√	[5]	•	[B] <- [B] + #1
LoaD Accum	ldaa	86	2	2	96	3	2	A6	5	2	B6	4	3				•	√	√	0	•	[A] <- [M]
	ldab	C6	2	2	D6	3	2	E6	5	2	F6	4	3				•	√	√	0	•	[B] <- [M]
OR	oraa	8A	2	2	9A	3	2	AA	5	2	BA	4	3				•	√	√	0	•	[A] <- [A] + [M]
	orab	CA	2	2	DA	3	2	EA	5	2	FA	4	3				•	√	√	0	•	[B] <- [B] + [M]
PuSH accum.	psha													36	4	1	•	•	•	•	•	[Mstack] <- [A]; SP--
	pshb													37	4	1	•	•	•	•	•	[Mstack] <- [B]; SP--
PULl to accum.	pula													32	4	1	•	•	•	•	•	SP++; [A] <- [Mstack]
	pulb													33	4	1	•	•	•	•	•	SP++; [B] <- [Mstack]
ROtate Left	rol							69	7	2	79	6	3				•	√	√	[6]	√	M
	rola													49	2	1	•	√	√	[6]	√	A
	rolb													59	2	1	•	√	√	[6]	√	B
ROtate Right	ror							66	7	2	76	6	3				•	√	√	[6]	√	M
	rora													46	2	1	•	√	√	[6]	√	A
	rorb													56	2	1	•	√	√	[6]	√	B
Shift Left, Arith.	asl							68	7	2	78	6	3				•	√	√	[6]	√	M
	asla													48	2	1	•	√	√	[6]	√	A
	aslb													58	2	1	•	√	√	[6]	√	B
Shift Right, Arith.	asr							67	7	2	77	6	3				•	√	√	[6]	√	M
	asra													47	2	1	•	√	√	[6]	√	A
	asrb													57	2	1	•	√	√	[6]	√	B
Shift Right, Logic	lsr							64	7	2	74	6	3				•	√	√	[6]	√	M
	lsra													44	2	1	•	√	√	[6]	√	A
	lsrb													54	2	1	•	√	√	[6]	√	B
STore Accum.	staa				97	4	2	A7	6	2	B7	5	3				•	√	√	0	•	[M] <- [A]
	stab				D7	4	2	E7	6	2	F7	5	3				•	√	√	0	•	[M] <- [B]
SUB	suba	80	2	2	90	3	2	A0	5	2	B0	4	3				•	√	√	√	√	[A] <- [A] - [M]
	subb	C0	2	2	D0	3	2	E0	5	2	F0	4	3				•	√	√	√	√	[B] <- [B] - [M]
Sub accums	sba													10	2	1	•	√	√	√	√	[A] <- [A] - [B]
SuB with Carry	sbca	82	2	2	92	3	2	A2	5	2	B2	4	3				•	√	√	√	√	[A] <- [A] - [M] - C
	sbcb	C2	2	2	D2	3	2	E2	5	2	F2	4	3				•	√	√	√	√	[B] <- [B] - [M] - C
Transfer accums.	tab													16	2	1	•	√	√	0	•	[B] <- [A]
	tba													17	2	1	•	√	√	0	•	[A] <- [B]
TeST zero or minus	tst							6D	7	2	7D	6	3				•	√	√	0	0	[M] - #00
	tsta													4D	2	1	•	√	√	0	0	[A] - #00
	tstb													5F	2	1	•	√	√	0	0	[B] - #00
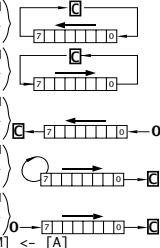
Index reg. & Stack ptr.

Operation/Mnemonic		Immed OP ~ #	Direct OP ~ #	Index OP ~ #	Extend OP ~ #	Inher OP ~ #	CCR H N Z V C	Description
ComPare indeX reg.	cpx	8C 3 3	9C 4 2	AC 6 2	BC 5 3		• 7 √ 8 •	[X] – [M:M+1]
DEcrement indeX reg.	dex					09 4 1	• • √ • •	[X] <- [X] – #1
DEcrement Stack ptr.	des					34 4 1	• • • • •	[SP] <- [SP] – #1
INcrement indeX reg.	inx					08 4 1	• • √ • •	[X] <- [X] + #1
INcrement Stack ptr.	ins					31 4 1	• • • • •	[SP] <- [SP] + #1
LoaD indeX reg.	ldx	CE 3 3	DE 4 2	EE 6 2	FE 5 3		• 9 √ 0 •	[X] <- [M:M+1]
LoaD Stack pntr.	lds	8E 3 3	9E 4 2	AE 6 2	BE 5 3		• 9 √ 0 •	[SP] <- [M:M+1]
STore indeX reg.	stx		DF 5 2	EF 7 2	FF 6 3		• 9 √ 0 •	[M:M+1] <- [X]
STore Stack pntr.	sts		9F 5 2	AF 7 2	BF 6 3		• 9 √ 0 •	[M:M+1] <- [SP]
Transfer indeX to Stack pntr.	txs					35 4 1	• • • • •	[SP] <- [X]
Transfer Stack pntr. to indeX	tsx					30 4 1	• • • • •	[X] <- [SP]

Branch & Jump

Operation/Mnemonic		Rel OP ~ #	Index OP ~ #	Extend OP ~ #	Inher OP ~ #	CCR N Z V C	Description
Branch Always	bra	20 4 2				• • • •	Skip always
Branch if:							
Carry Clear	bcc	24 4 2				• • • •	Skip if **C** is 0
memory Higher or Same[10] as accum.	bcc[11]	24 4 2				• • • •	Skip if **C** is 0
Carry Set	bcs	25 4 2				• • • •	Skip if **C** is 1
memory LOwer[10] than accum.	bcs[12]	25 4 2				• • • •	Skip if **C** is 1
Not Equal to zero	bne	26 4 2				• • • •	Skip if **Z** is 0
EQual to zero	beq	27 4 2				• • • •	Skip if **Z** is 1
memory HIgher[10] than accum.	bhi[12]	22 4 2				• • • •	Skip if **C** + **Z** = 0
memory LOwer[10] or same as accum.	bls	23 4 2				• • • •	Skip if **C** + **Z** = 1
oVerflow Clear	bvc	28 4 2				• • • •	Skip if **V** is 0
oVerflow Set	bvs	29 4 2				• • • •	Skip if **V** is 1
Plus	bpl	2A 4 2				• • • •	Skip if **N** is 0
MInus	bmi	2B 4 2				• • • •	Skip if **N** is 1
memory Greater or Equal[12] to accum.	bge	2C 4 2				• • • •	Skip if $\mathbf{N}\oplus\mathbf{V} = 0$
memory Less Than[12] accum.	blt	2D 4 2				• • • •	Skip if $\mathbf{N}\oplus\mathbf{V} = 1$
memory Greater Than[12] accum.	bgt	2E 4 2				• • • •	Skip if $\overline{\mathbf{N}\oplus\mathbf{V}}\cdot\overline{\mathbf{Z}} = 1$
memory Less or Equal[12] to accum.	ble	2F 4 2				• • • •	Skip if $\overline{\mathbf{N}\oplus\mathbf{V}}\cdot\overline{\mathbf{Z}} = 0$
Branch to SubRoutine	bsr	8D 8 2				• • • •	Skip always after pushing [PC] onto stack
JuMP to ea	jmp		6E 4 2	7E 3 3		• • • •	Goto ea
Jump to SubRoutine at ea	jsr		AD 8 2	BD 9 3		• • • •	Push [PC] onto stack then goto ea
No OPeration	nop				01 2 1	• • • •	[PC] <- [PC] + 1
ReTurn from Interrupt	rti				38 10 1	<—13—>	Restore last state from stack
ReTurn from Subroutine	rts				39 5 1	• • • •	Pull PC from stack
SoftWare Interrupt	swi				3F 12 1	• • • •	Save state on stack then goto [FFFA:Bh]
WAit for Interrupt	wai				3E 9 1	• • • •	Idle until interrupt

Code Condition Register

Operation	Mnemonic	Inher OP ~ #	CCR H I N Z V C	Description
CLear Carry	clc	0C 2 1	• • • • • 0	[**C**] <- #0
SEt Carry	sec	0D 2 1	• • • • • 1	[**C**] <- #1
CLear oVerflow	clv	0A 2 1	• • • • 0 •	[**V**] <- #0
SEt oVerflow	sev	0B 2 1	• • • • 1 •	[**V**] <- #1
CLear Interrupt mask	cli	0E 2 1	• 0 • • • •	[**I**] <- #0
SEt Carry	sec	0F 2 1	• 1 • • • •	[**I**] <- #1
Transfer A to CCR	tap	06 2 1	A5 A4 A3 A2 A1 A0	[CCR] <- [A]
Transfer CCR to A	tpa	07 2 1	• • • • • •	[A] <- [CCR]

Index