DR. WATSON
Computer Learning Series

Beginners Assembly Language Programming

VIC 20

Commodore Reg. Product

by Dr. P. Holmes

June 1982

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publisher for any errors or omissions contained herein.

1st Edition February 1982
2nd Edition March 1982
3rd Edition June 1982 (Revised format)

OTHER BOOKS IN THIS SERIES:

The Dr Watson Book of Assembly Language Programming for the Commodore PET 2/3/4/8000. (2nd Edition)
ISBN 0 907792 02 2

COPYRIGHT © 1982 P HOLMES World rights reserved.

No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including but not limited to photography, photocopy, magnetic or other recording, without the prior written permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

ISBN 0 907792 05 7

Published by: Glentop Publishers Ltd
21 Colin Drive
London
NW9 6ES UK

* Dr Watson is a Trademark of Glentop Publishers Ltd.

Printed By Image Publicity, 2 Northington Street, London WC1N 2JJ Tel: 01-405 5333

C O N T E N T S

INTRODUCTION	Loading, debugging and running the ASSEMBLER
CHAPTER 1	Understanding assembly and machine code. Writing and understanding assembly language instructions and simple programs. 6502 architecture.
CHAPTER 2	Unconditional jumps, calculating addresses. The program counter. Conditional jumps (branches). The 6502 flags.
CHAPTER 3	Screen displays; Timing of programs, Modes of addressing.
CHAPTER 4	Mathematical operations: Hexadecimal inputs. Multiplication, division, binary-coded decimal arithmetic. Logical operators. Binary multiplication.
CHAPTER 5	Enhancing the basic ASSEMBLER, adding labels, memory labels and Macros.
CHAPTER 6	Machine-code without the assembler. The monitor, protecting programs in memory. Machine-code programs with colour and sound.
CHAPTER 7	Using the VIC's built in routines. The Stack.
CHAPTER 8	Interrupts. The overflow flag. Numerical screen output, the USR command, signed and floating-point numbers and built-in sub-routines.
CHAPTER 9	Solutions to Exercises.
APPENDIX I	Binary, Binary-coded decimal and hexadecimal notation.
APPENDIX II	Table 1. The 6502 instruction set. Table 2. VIC Character set. Table 3. Hex to decimal conversion. Table 4. ASCII Character set. Table 5. 6502 Flag Guide.
APPENDIX III	Floating-point number sub-routine addresses.
APPENDIX IV	VIC Memory allocation.

I N T R O D U C T I O N

This book is a self-paced course on machine-code/ Assembly-language programming based on the 6502A microprocessor. In order to overcome the usual difficulty associated with assembly languages, the Mnemonics used are totally uniform throughout the book and the instruction set used is that employed by the assembler that was written especially to accompany this text. Most of the examples are designed to be loaded via this assembler and all are designed to run on the Commodore VIC-20.

In order to ensure that the vital first few machine code programs run, the opening chapter goes in great detail through the stages of entering, assembling, listing and running these programs. Those who bring to the book pre-existing skill are asked to bear with the text in these early stages. Notwithstanding the gentle pace of the opening, the reader should find himself/herself accelerating along with the text. By the end of chapter eight he/she will have covered the whole of 6502A instruction set. This is identical to that of the 6502, the 6502A chip simply running faster, thus when the "6502" is referred to in the text it means the whole chip family - including the 6502A.

THE 6502A ASSEMBLER

The listing contained in this introduction is of an assembler designed to run on a Commodore VIC-20 Computer and is written in BASIC V2. In this introduction it is listed in its simplest form that is used in the first four chapters of the book ASSEMBLER 1. In chapter five some refinements are added, principally those to allow the use of labels and these may be added if you have the Super-expander. If you have bought the cassette this full version is the one you have. The listing of ASSEMBLER 1 is numbered to allow space for the addition of this additional code later so when you convert ASSEMBLER 1 to ASSEMBLER 2 you can simply type in the extra lines.

The program should be entered as for any other listing but BEFORE you attempt to run it, it should be SAVEd. This is particularly true of any program that uses any machine-code element as a POKE into the wrong part of memory will bring the whole system about your ears! Then there is no alternative but to turn off and reload.

* Assembler available on tape/disk from your dealer or direct from the publisher.

Don't worry about crashes harming your machine - they won't - a RUN/STOP and RESTORE should return you to BASIC - if not just turn off and reload.
When you try to run your program, you will, no doubt, have the usual clutch of syntax errors etc., but the hardest errors to spot will be in the DATA statements. In order to help in clearing any errors that may have crept in, add the following short program to the front of your assembler:-

```
1  A=0:C=0
2  READ A$,B:PRINT A;A$;B
3  REM IF A$="FRED" THEN END
4  A=A+1:IF A-C = 17 THEN 7
5  IF A=255 THEN END
6  GOTO 2
7  GET A$:IF A$="" THEN 7
8  C=C+17:GOTO 2
```

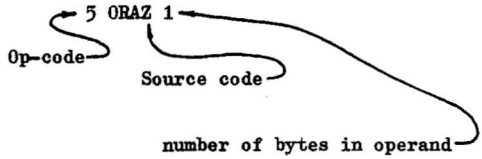
When you RUN this it will read through the DATA statements and print these out as below:-

```
0 BRK 0
1 ORAIX 1
2 0
3 0
4 0
5 ORAZ 1

etc.
```

The zero's you can ignore as long as they are in the right place, but the rest must be correct, the easiest way to check them being to take one from each block of 17 and check in this way.

Take the entry



Look up the entry for ORAZ in Appendix 2 under "Mnemonic". Read across to "Op-code in decimal" and this should be the same as the op-code, in this case 5. Then check the number of bytes in the operand against the listing.
After checking the 17 entries press the space bar to continue.
When entering program code such as this, the biggest danger is in omitting a comma or two which will either

put the whole block out by one or more pairs of entries or - if you're lucky will give a mismatch error when the debugging program tries to read a string as a numeric variable i.e. B. If this happens it should point you to where the error lies.

Once safely loaded, SAVE the program BEFORE running any machine-code programs!

When entering machine-code/assembly language programs, the assembler will trap many syntax errors and inform you of their occurrence. Most of these will be obvious when you look back at what you have entered. However, in order to assist in your diagnosis, the assembler will give a guide to the error by reference to the error table below.

ERROR #1	Source code entered was invalid.
ERROR #2	Operand of wrong size - too big or too small.
ERROR #3	No source code found after label.
ERROR #4	Label and label references not in Assembler 1.
ERROR #5	Hexadecimal operand too small - format should be $XX XX.
ERROR #6	NO OPERAND found where expected.
ERROR #7	Separating space missing.
ERROR #8	LOAD/SAVE format incorrect.

Once the assembler is taped you may wish to type in and SAVE the Bin/Hex tutor and exercises. These programs demonstrate binary, hexadecimal and binary-coded decimal notations and gives a series of exercises in converting one to the other. It is written to be used in conjunction with Appendix 1. If you are totally conversant with these notations then you may feel it to be unnecessary to enter this program. However, machine code programming requires a good knowledge of binary and hexadecimal in particular, so if you have any doubts about your ability in this field the Bin/Hex program might help.

```
100 POKE36879,41
110 PRINT"J≡QQQ     ⌒ COPYRIGHT
120 PRINT"     |C| P.HOLMES
130 PRINT"     ‿     1982
140 CLR:D$="0123456789ABCDEF"
150 PRINTTAB(9)"QQQQMENU
160 PRINTTAB(9)"‾‾‾‾Q
170 PRINTTAB(7);"ENTER..1Q
180 PRINTTAB(7);"LIST...2Q
190 PRINTTAB(7);"RUN....3Q
210 I$="ERROR #"
220 GOSUB940
230 ONVAL(Z$)GOTO250,640,920
240 GOTO220
250 INPUT"JSTART ADD";L
260 INPUTX$:D=LEN(X$):J=0
270 IFX$="END"THEN1780
290 F=1:GOTO1300
300 IFD=3ORD=4THENX1$=X$:G=0:G1$="":GOTO480
310 IFD<3THENPRINTI$;"1":GOTO260
320 FORX=1TOD
330 IFMID$(X$,X,1)=" "ANDJ=0THENJ=X
340 NEXT:IFJ=0ANDD>4THENPRINTI$;"7":GOTO260
350 X1$=LEFT$(X$,J-1)
360 G1$=RIGHT$(X$,D-J):G=VAL(G1$)
370 IFMID$(X$,J+1,1)<>"$"THEN480
380 IFLEN(G1$)<6THENPRINTI$;"5":GOTO260
390 N=5:GOSUB410:GOTO480

410 G=0:FORX=NTO1STEP-1
420 IFX=3THENNEXTX
430 S$(X)=MID$(X$,X+J+1,1):FORZ=0TO15
440 IFS$(X)=MID$(D$,Z+1,1)THENT=Z:Z=15
450 NEXTZ:X1=N-X+1:IFX<3THENX1=N-X
460 G=G+T*16↑(X1-1):NEXTX
470 RETURN
480 FORX=0TO255:READE$,B
490 IFX1$=E$THENN=X:X=256
500 NEXTX:RESTORE
510 IFX1$=X$THEN530
520 IFG=0ANDF=1ANDG1$<>"0"THENGOSUB1740
530 IFB>3THENPRINTI$;"1"
540 IFB=0THENGOSUB570
550 ONBGOSUB580,600
560 GOTO260
570 POKEL,N:L=L+1:RETURN
580 IFG<0ORG>255THENPRINTI$;"2":RETURN
590 POKEL,N:POKEL+1,G:L=L+2:RETURN
```

continued.......

```
600 IFG<0THENPRINTI$;"2":RETURN
610 POKEL,N
620 G2=INT(G/256):G1=G-(G2*256)
630 POKEL+1,G1:POKEL+2,G2:L=L+3:RETURN
640 INPUT"□START ADD";L:Y=0:PRINT"□";
650 Y=Y+1:IFY<21THEN690
660 GOSUB940
670 IFZ$<>"C"THEN110
680 Y=0:PRINT"□";:GOTO650
690 C=PEEK(L):FORX=0TOC:READE$,B:NEXT:RESTORE
700 IFE$=""THEN890
710 P=3:GOSUB800:P$=G$
720 ONBGOTO760,830
730 C=L:P=1:GOSUB800
740 PRINTL;G$;TAB(9)"!";E$TAB(15)" ■"P$"■"
750 L=L+1:GOTO650
760 H1$=P$:C=PEEK(L+1):C1=C:P=3:GOSUB800:C=L:P$=G$
770 H2$=P$:P=1:GOSUB800
780 PRINTL;G$;TAB(9)"!";E$;" ";C1:PRINTTAB(9);
790 PRINT"!■";H1$;" ";H2$"■":Y=Y+1:L=L+2:GOTO650
800 G$="":FORX=PTO4:S=16↑(4-X):H(X)=INT(C/S)
810 C=C-(H(X)*S):H$(X)=MID$(D$,H(X)+1,1):G$=G$+H$(X)
820 NEXT:RETURN
830 H1$=P$:C=PEEK(L+1):P=3:GOSUB800:H2$=G$
840 C=PEEK(L+2):P=3:GOSUB800:H3$=G$
850 G=PEEK(L+1)+PEEK(L+2)*256
860 C=L:P=1:GOSUB800
870 PRINTL;G$;TAB(9)"!";E$;G:PRINTTAB(9);
880 PRINT"!■";H1$;" ";H2$;" ";H3$"■":Y=Y+1:L=L+3:GOTO650
890 P=3:GOSUB800:P$=G$:C=L:P=1:GOSUB800
900 PRINTL;G$;TAB(9)"!?NO OP:■"P$"■"
910 L=L+1:GOTO650
920 INPUT"□START ADD";L:PRINT"□":SYS(L):GOSUB930:GOTO100
930 PRINT"■■■■■■■■■■■■■■■■■■■■■PAK
940 GETZ$:IFZ$=""THEN940
950 RETURN
960 DATABRK,0,ORAIX,1,,,,,,ORAZ,1,ASLZ,1,,
970 DATAPHP,0,ORAIM,1,ASLA,0,,,,,ORA,2,ASL,2
980 DATA,,BPL,1,ORAIY,1,,,,,,,ORAZX,1,ASLZX,1
990 DATA,,CLC,0,ORAY,2,,,,,,,ORAX,2,ASLX,2
1000 DATA,,JSR,2,ANDIX,1,,,,,BITZ,1,ANDZ,1,ROLZ,1
1010 DATA,,PLP,0,ANDIM,1,ROLA,0,,,BIT,2,AND,2
1020 DATAROL,2,,,BMI,1,ANDIY,1,,,,,,,ANDZX,1
1030 DATAROLZX,1,,,SEC,0,ANDY,2,,,,,,,ANDX,2
1040 DATAROLX,2,,,RTI,0,EORIX,1,,,,,,,EORZ,1
1050 DATALSRZ,1,,,PHA,0,EORIM,1,LSRA,0,,
1060 DATAJMP,2,EOR,2,LSR,2,,,BVC,1,EORIY,1
1070 DATA,,,,,,EORZX,1,LSRZX,1,,,CLI,0,EORY,2
```

continued.....

```
1080 DATA,,,,,EORX,2,LSRX,2,,,RTS,0,ADCIX,1
1090 DATA,,,,,ADCZ,1,RORZ,1,,,PLA,0,ADCIM,1
1100 DATARORA,0,,,JMPIA,2,ADC,2,ROR,2,,,BVS,1
1110 DATAADCIY,1,,,,,,,ADCZX,1,RORZX,1,,,SEI,0
1120 DATAADCY,2,,,,,,,ADCX,2,RORX,2,,,,
1130 DATASTAIX,1,,,,,STYZ,1,STAZ,1,STXZ,1,,
1140 DATADEY,0,,,TXA,0,,,STY,2,STA,2,STX,2,,
1150 DATABCC,1,STAIY,1,,,,,STYZX,1,STAZX,1
1160 DATASTXZY,1,,,TYA,0,STAY,2,TXS,0,,,,
1170 DATASTAX,2,,,,,LDYIM,1,LDAIX,1,LDXIM,1,,
1180 DATALDYZ,1,LDAZ,1,LDXZ,1,,,TAY,0,LDAIM,1
1190 DATATAX,0,,,LDY,2,LDA,2,LDX,2,,,BCS,1
1200 DATALDAIY,1,,,,,LDYZX,1,LDAZX,1,LDXZY,1
1210 DATA,,CLV,0,LDAY,2,TSX,0,,,LDYX,2,LDAX,2
1220 DATALDXY,2,,,CPYIM,1,CMPIX,1,,,,,CPYZ,1
1230 DATACMPZ,1,DECZ,1,,,INY,0,CMPIM,1,DEX,0,,
1240 DATACPY,2,CMP,2,DEC,2,,,BNE,1,CMPIY,1,,,,,,
1250 DATACMPZX,1,DECZX,1,,,CLD,0,CMPY,2,,,,,,,CMPX,2
1260 DATADECX,2,,,CPXIM,1,SBCIX,1,,,,,CPXZ,1,SBCZ,1
1270 DATAINCZ,1,,,INX,0,SBCIM,1,NOP,0,,,CPX,2,SBC,2
1280 DATAINC,2,,,BEQ,1,SBCIY,1,,,,,,,SBCZX,1,INCZX,1
1290 DATA,,SED,0,SBCY,2,,,,,,,SBCX,2,INCX,2,,8
1300 GOTO300
1740 PRINTI$;"4":GOTO260
1780 GOTO100
```

C H A P T E R 1

Having driven a computer before you are probably well aware that a 'something' exists called "machine-code" and that other thing "assembler". Quite simply, machine-code is the language that the micro-processor chip in your computer understands. As an example take a simple addition sum — adding 1 and 2.

In English you would say:-

Add one to two, what's the answer?

In Basic you might say something like:-

```
10  LET A = 1
20  LET B = 2
30  C = A + B
40  PRINT C
```

In 6502Amachine-code you could say:-

```
A2  01
8E  84 03
A9  02
6D  84 03
8D  00 1E
```

This is pretty-well unintelligible isn't it? Well, that's why we use assembly language. The same program is given below in assembly language along with a brief comment on each line.

```
LDXIM  1          Load X register with a '1'.
STX    900        Store the content of X in 900.
LDAIM  2          Load the Accumulator with a '2'.
ADC    900        Add to the accumulator the contents
                  of memory location 900.
STA    7680       Store contents of accumulator in
                  7680  (i.e. on screen).
RTS               Return from machine-code sub-routine.
```

It's much easier to read that than the machine-code, isn't it? With an assembler you can enter your program in assembly language and be able to read through it and understand it readily. All the assembler does is to change the assembly language into machine-code. Thus, when it sees "LDXIM", it changes this command into an 'A2' and puts this into memory in the right place.

It is possible to enter machine-code directly into memory and this will be demonstrated later on in this book. However, until then we will concentrate mainly on assembly language programs.

The heart of the 6502Ais the accumulator (A) through which almost all your data has to flow. It is basically an eight bit store that can store a number up to 255. 6502A instructions allow you to write directly into this store using the instruction "LoaD a number into the Accumulator using Immediate Mode". The Mnemonic for this is LDAIM.

Another instruction allows you to transfer a number from this store to any specified memory location. If this memory location is between 7680 and 8186 then the number taken from the store will be displayed on the screen. The instruction is:-

STA	STore contents of Accumulator in the address specified.

You should note that all the TRANSFER commands you will meet should really be thought of as COPY commands as they create a SECOND copy of that data, leaving the source of this unaltered.

Let's have a go then at running a machine-code program!

This program will put a number into the accumulator and then transfer it to the top left position on the screen, i.e. location 7680.

One word about the assembler. It needs to be told when you start a program and where you want it to be put in the memory. For short programs the 192 bytes of the cassette buffer. from memory location 828, are sufficient so the early programs will start there. Also, initially all numbers should be put in in a decimal format.

Similarly it needs to be told that the program has ended. Thus, the first and last lines of the assembly program are:-

```
START ADD? 828
..... Program .....
END
```

N. B. The underlining of START ADD ? means that the computer types this. This convention is used throughout this book. These lines are nothing to do with the machine-code program but just for the assembler, in the jargon they are called PSEUDO-CODES.

When you have put in your machine-code program you may run it by typing in SYS 828 or by using the facility on the assembler. Either way you must tell the program to return from machine-code to BASIC or monitor. The command that does this is ReTurn

from machine-code Subroutine or RTS

Right, to put that into the program we must:

1. Tell the assembler that the START ADD? is 828.

2. LoaD number '0' into accumulator A using Immediate Mode. The Mnemonic for this is
   LDAIM + number to be entered
   i.e. LDAIM 0

3. STore in a specific Address the contents of the Accumulator. Mnemonic = STA. After this we must tell the 6502 what the address is, i.e. STA 7680.

4. ReTurn To the BASIC Subroutine, RTS

5. Tell the assembler to END

OR

PROGRAM 1.1

```
START ADD? 828
LDAIM 0
STA 7680
RTS
END
```

Now to enter this:-

a) Load the ASSEMBLER program into your machine.
b) Type in RUN <return>
c) Screen shows MENU
d) Select '1' to ENTER program.
   Screen tells you to put in the assembly code and prompts with START ADDRESS?
e) Type in "828" and press <return> After entering the code, the assembler will type a "?" then:
f) Type in "LDAIM" <space> "0" press <return>
g) Type in "STA" <space> "7680" <return>
h) Type in "RTS" <return>
i) Type in "END"

At this stage your program should appear as overleaf:-

START ADD? 828
? LDAIM 0
? STA 7680
? RTS
? END

If it does, then O.K. press <return> and carry on to j). If it doesn't, press <return> and then go back to d).

Program returns to MENU.

j) Select '3' for RUN.
Screen asks for address of program start.

k) Type in "828" <return>
After this input the program first clears the screen and then runs the machine-code program which types an "@" in the top left-hand corner of the screen.

l) Press any key and continue. Program returns to MENU.

Now select '2' to list the program.

The program then asks you for the START ADD?

Type in "828" <return>

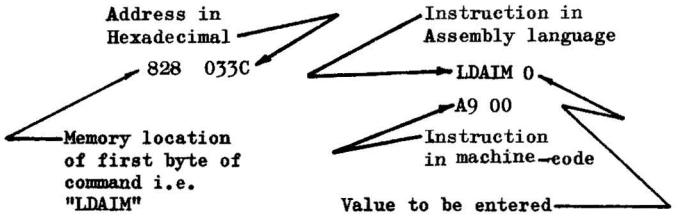
The screen displays:-

```
828 033C | LDAIM  0
         | A9 00
830 033E | STA 7680
         | 8D 00 1E
833 0341 | RTS   60

   etc.       etc.
```

Your machine will show a whole screen of data but anything below the line beginning 833 will be ignored as the 6502 reads the RTS (ReTurn from machine-code Sub-routine) and returns to BASIC.

What the above shows is, taking the first line of program 1.1.



Thus the listing is a check on what you entered and also gives you the full machine-code program or the object program. Reading this off from below the assembly language listing this is:-

```
A9  00
8D  00  1E
60
```

This object code could be entered directly into memory and would yield the same results as the program you typed in. The assembly language only helps you to compile the program in the first place.

Now to look at another instruction and use this in a program. As stated earlier, the accumulator is the repository of most "answers" and this ADC "does a sum" and loads the answer into the accumulator.

ADC	ADd with Carry contents of specified memory location to the accumulator.

To do this, however, we must first add two lines to the program that get the 6502 ready for adding. Don't worry what they mean for now — just type 'em in! - and follow the instructions.

One other point about the jargon! The term INSTRUCTION is used to describe an executable machine code statement. Thus it could consist of LDAIM 0 or just RTS. However, the term is also used to refer to the mnemonic alone, as when one says the 6502 instruction set. In this book, the term COMMAND is used to refer to the mnemonic part of an instruction when this precision is required. For instance, in the instruction LDAIM 0, the LDAIM part may be referred to as the command.

Let's look at the stages:-

PROGRAM 1.2

START ADD? 828	Gives address for beginning of program.
CLD) CLC)	Gets the 6502 ready for adding.
LDAIM 1	LoaD "1" into the Accumulator in IMmediate mode.
STA 7680	Store the contents of accumulator (1) in 7680.
LDAIM 2	LoaD "2" into the Accumulator in IMmediate mode.
ADC 7680	ADd Contents of 7680 (1) to the contents of the accumulator (2).
STA 7682	STore the contents of the Accumulator (3) in 7682.
RTS	ReTurn from machine-code Subroutine.
END	End assembly

Right then, let's type it in!

If you make a mistake before pressing return, correct normally. If you press return and then make a mistake - type in "END" and start again.

1. Run the Assembler program if not already running.
2. Select '1' to begin entering assembly program.
3. Tell assembler where to start, i.e. type in "828" <return>
4. Type in "CLD" <return> (press <return> after each entry).
5. Type in "CLC"
6. Type in "LDAIM 1"
7. Type in "STA 7680"
8. Type in "LDAIM 2"
9. Type in "ADC 7680"
10. Type in "STA 7682"
11. Type in "RTS"
12. Type in "END"
13. Select '3' to run program and then give start address - 828.
    Screen displays "A" in 7680 and "C" in 7682.
    Press any key to return to menu.
    If you wish to list, select a '2' and then give the location "828".

- - - - - - - -

Another way of looking at the two lines

```
LDAIM 1
STA 7682
```

is as a way of putting a "1" into memory or of printing a VIC ASCII "1" or an "A" on the screen.

- - - - - - - - - -

The 6502 has two index registers in addition to its accumulator and these are referred to as Index registers X and Y and can each store one 8 bit number. The arrangement of these or, as the jargon has, the ARCHITECTURE of the 6502 is shown below (in part) in Fig. 1.1.

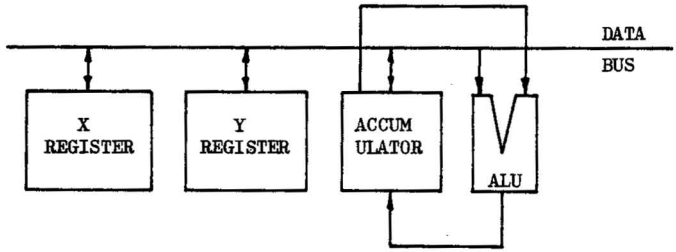


Fig. 1.1

In this figure the X and Y registers are shown identically, although they do differ slightly. Nevertheless, they are both indexed registers and can thus have the value stored in them incremented or decremented (increased or decreased in steps of 1). To the right of the figure is the ALU or Arithmetic logic unit in which arithmetic and logical operations are carried out. It has two inputs for the data that it manipulates and the output from the operation is fed into the accumulator. For this reason,almost all data flows through the accumulator making this a key feature of the 6502. Data flows between the various registers along the DATA BUS which is a common trackway for communication within the 6502. For talking to devices beyond the chip this data bus is extended to access memory also.

In the remainder of this chapter, we will look at these registers and the ways that data can be fed in, out and between these.

First of all we'll have a go at using the X-register -- so to load this we use the instruction

LDX

LoaD index register X with data from the specified address, i.e.

LDX 900 means LoaD index register X with the data in memory location 900.

LDX differs from the earlier "LDAIM" (apart from one loading the Accumulator and one X register) in that the LDAIM command is an IMmediate load command. When the 6502 sees this it looks for what's immediately following the instruction and loads that - as data - into the Accumulator. With the new command above"LDX"the 6502 looks for what follows and this specifies the ADDRESS of the data. Thus with the instruction:-

LDX 900

the 6502 goes to memory location 900 to find the data which it loads into the X-register. This instruction (as are all the register instructions) is really a COPY as the data put into the X-register is really COPIED from it as the same data remains in that memory location addressed.

To recover the data we may use the instruction:-

STX

STore the contents of the specified address in register X, i.e.

STX 7680 means Transfer contents of X register into memory location 7680.

Right here goes. Here's the program!

PROGRAM 1.3

START ADD? 828

LDAIM 1	Load '1' into the accumulator.
STA 7680	Store the contents of accumulator in
LDX 7680	Load into X-register, contents of memory location 7680 (i.e."1").
STX 7682	Store contents of X-register in 7682 (i.e. 1)
RTS	Return from machine-code subroutine.
END	End assembly.

At Menu select 'R' to run the program.
The screen should display "A", "space", "A" at the top left-hand position.

By now you should be able to write simple programs so, as an exercise try the following:-

Exercise 1.1

Load the Accumulator directly with a '1', display this (a screen'A') in 7680. Answer on page 9.1.

Don't forget to put in the "RTS" at the end. If you do forget then the 6502 will run on to see what it can find and try to execute this. If you're unlucky it will find something that crashes the system. It may be a recoverable crash and RUN/STOP pressed along with RESTORE may return the VIC to BASIC. If it is not recoverable then it will be necessary to re-load the assembler and start again.

Exercise 1.2

Write your name in the top left hand corner of the screen. One possible answer on page 9.1.

Exercise 1.3

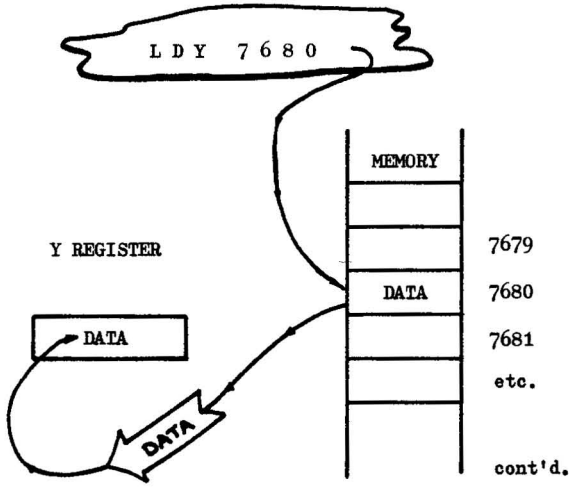
Put an 'X' in each of the four corners of the screen. Answer on page 9.1.

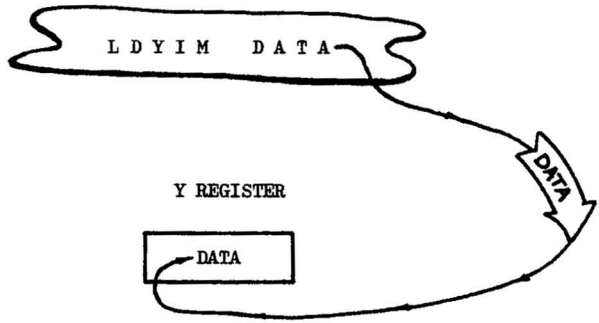
****

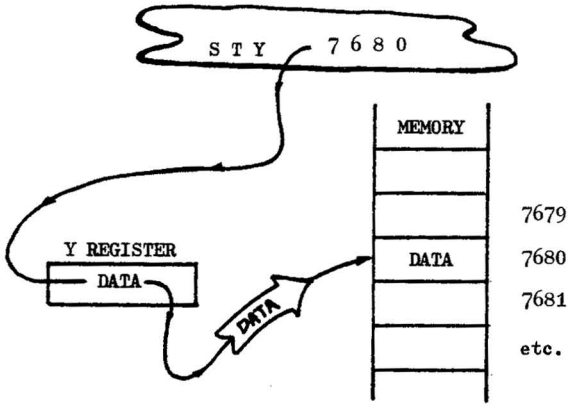
The Load and Store instructions that we have met so far are complemented by the corresponding Y-register instructions.

LDY	LoaD register Y with data at specified address.
LDYIM	LoaD register Y with data specified.
STY	STore the data in the Y-register at specified address.

Diagrammatically, these instructions are shown below:-







You should now know, or be able to interpret the following:-

LDA	LDX	LDY	RTS
LDAIM	LDXIM	LDYIM	
STA	STX	STY	
ADC			

For many operations, BUT NOT ALL, the X and Y registers can be treated interchangeably for instance program 1.3 could be written:-

PROGRAM 1.3

```
START ADD ? 828
LDAIM 1
STA 7680
LDX 7680
STX 7682
RTS
END
```

OR

PROGRAM 1.3a

```
START ADD ? 828
LDAIM 1
STA 7680
LDY 7680
STY 7682
RTS
END
```

Because of this interchangeability and the need to swap data rapidly between registers during the run of a program several instructions exist to do this automatically. They are typified by:-



Transfer the contents of the Accumulator into the index register X.

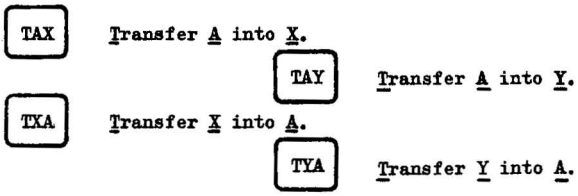
Using this command in Program 1.3, (to produce program 1.4) shortens this while maintaining the same function.

PROGRAM 1.4

```
START ADD? 828
LDAIM 1
STA 7680
TAX
STX 7682
RTS
END
```

When run the screen should display two "A"s, one in 7680 and one in 7682. Descriptions given of the codes so far have been spelled out in detail. However as you are getting more used to the jargon, it is

reasonable to begin to abbreviate. From now on, instead of "the contents of the X register", we will just refer to X and similarly so with the Y-register (Y) and accumulator (A). Thus, a summary of the transfer instructions is:-



Exercise 1.4

Write a program that loads a "Z" into the accumulator and an "A" into the X-register, then swaps these over without using IM (immediate entry) commands, prints the "Z" on the first screen memory location and the "A" on the last.

A possible answer on page 9.1.

Exercise 1.5

Write a program that: Loads a diamond into the accumulator, an asterisk into X and an "E" into Y. Then, without using any immediate entry commands move the "E" into A, the diamond into X and the asterisk into Y. Print the diamond in the screen top-left, the asterisk in the top-right and twe "E"s in bottom two corners of the screen.

A possible answer on page 9.2.

****

CHAPTER 2

Jumping, branching , the Program counter.

Few real life programs proceed along a smooth uninterrupted path without jumping or branching at some stage. This chapter looks at those commands and their uses and then examines the flags that enable the branches to be controlled.

Unconditional Jumps

These tell the program to jump willy-nilly - no conditions. Only two such 6502 instructions exist; the first to be considered is:-

JMP — JuMP to the specified address.

For instance, JMP 834 means jump to memory location 834.

Put in a program and it will look like this:-

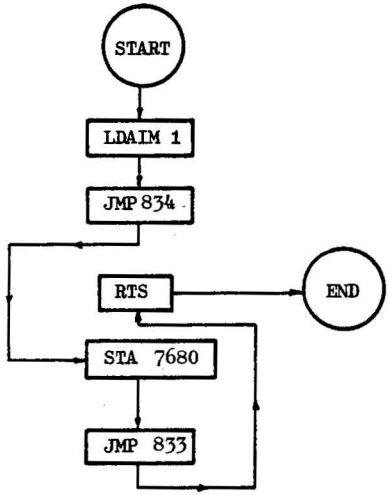


Fig. 2.1

Such a jump routine doesn't really achieve a lot but it could, for instance, be used to patch a piece of code into a program. In figure 2.1, for instance, the command STA 7680 has effectively been inserted into the program.

Now this can be typed in thus:-

PROGRAM 2.1

```
START ADD ? 828
LDAIM 1
JMP 834
RTS
STA 7680
JMP 833
END
```

Once again, it can be run by selecting 'R' on the menu and then starting the program at 828. When run,it should give an 'A' in the top left-hand corner.

When the jumps are used in this way it's necessary to tell the program exactly where to jump to, i.e. to give an ADDRESS, hence JMP 834. Calculating these addresses is straight forward as long as it is done systematically. For instance, all the commands or "SOURCE-CODES": RTS, LDAIM, JMP etc. take up one byte of memory, thus, to jump over RTS in Fig. 2.2a we jump from 830 to 832, i.e. over 831 which contains RTS.

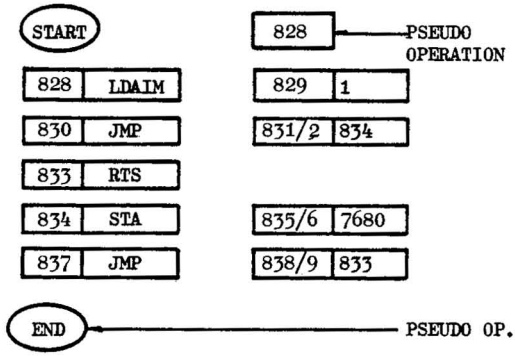


Fig. 2.2a

The part following the source-code is known as the OPERAND and calculating its length is a bit more complicated!

One easy way for you, as the proud owner of a Dr Watson assembler, is to look at the DATA statements! There you will see the first entry is BRK,0 and it is the "0" that tells the assembler not to look for an operand. Looking further along that line you will see ORAIX,1 and, again the "1" tells the assembler how big the operand is on this occasion - 1 byte. If you continue reading you will come to ASLX,2, a command with a 2 byte operand.

The different lengths come from the different sizes of operands. For instance the instruction RTS has no operand as the address to which it points is determined by the point from which the sub-routine came originally. LDAIM instructions have a 1 byte operand as the accumulator can only hold values up to 255, or 1 byte, in its eight bit register. Other instructions that you have met need two bytes as their operands are greater than 255. For instance, when printing out an accumulator value onto the screen the command STA 7680 needs two bytes. (See Appendix II)
Figure 2.2a shows the location of the various instructions and operands for program 2.1.

LDAIM 1 takes up two bytes - one for its object code A9 and one for the number to be loaded into the accumulator. Remember the accumulator is only one byte long, so it can only hold a number up to 255.

JMP 833 takes three bytes - one for JMP and two for the address - here 833.

RTS takes only one byte - it has no operand as do TAX, TXA, etc.

STA 7680 This takes three bytes - one for STA(8D) and two for 7680.

This can also be seen by using the "LIST" command on the assembler MENU. Return to MENU and type "L" for LIST, then tell the assembler where to start listing i.e. type in "828".

The screen will display:-

```
828 033C | LDAIM 1
         | A9 01
830 033E | JMP 834
         | 4C 42 03
833 0341 | RTS 60
834 0342 | STA 7680
         | 8D 00 1E
837 0345 | JMP 833
         | 4C 41 03
```

cont'd.

Fig. 2.2b

As Fig. 2.2b shows, the assembler breaks the instructions and operands down into one byte chunks. We can calculate the total length of the program by counting the one byte pieces of the machine code. Thus, program 2.1 is 12 bytes long.
(A9 01 4C 42 03 60 8D 00 1E 4C 41 03)

JSR — Jump to Sub-Routine.

This is another jump command which is used along with RTS and together these are like GOSUB.....RETURN in BASIC.

	BASIC	ASSEMBLER
10	GOSUB 200	830 JSR 834
	"	
	"	
	"	
	"	
200	REM***SUB-ROUTINE	834 STA 7680
300	RETURN	837 RTS

We can modify program 2.1 to use this instruction instead of the straight JMP used there. The program then becomes:-

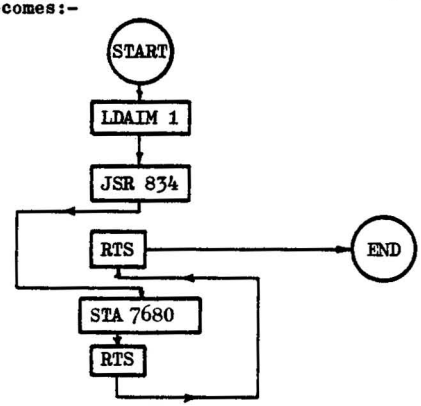


Fig. 2.3

PROGRAM 2.2

```
START ADD? 828          Pseudo-code.
LDAIM 1                 Load Accumulator in
                        immediate mode with
                        a '1'.
JSR 834                 Jump to the subroutine
                        at 834.
RTS                     Return from machine-code
                        to subroutine i.e. back
                        to BASIC.
STA 7680                Store contents of accu-
                        mulator in 7680.
RTS                     Return from subroutine.
END                     Pseudo-code.
```

The advantage of RTS over JMP is shown in this program as, with RTS it is not necessary to calculate the address for the jump. In program 2.1 we had to put in the JMP 833 to have the same effect as RTS in this program. The 6502 does this trick by use of a :

PROGRAM COUNTER (PC)

This is a 16 bit register which contains the address of the next command which is to be executed. In reality all it is is two 8-bit memories, one for each byte, which is built into the 6502 chip. When you select 'R' at the assembler MENU and then type in 828, this generates a command that sets the PC to 828 and starts execution from there. As the PC fetches each byte from memory it is incremented by 1, thus always pointing to the next memory location containing the required data.

Take the first three lines of program 2.2 for instance:

```
START 828
LDAIM 1
JSR   834
```

A summary of the PC counter contents during execution of this is given in fig. 2.3.

PROGRAM	PC BEFORE EXECUTION	PC AFTER EXECUTION
START 828	?	828
LDAIM 2	828	830
JSR 834	830	834

Fig. 2.3

This figure illustrates how the PC steps through the program until it comes to the JSR command. It then takes the jump command and sets the PC to the address specified, i.e. 834. As it is only two bytes long and can thus only store one address, the PC uses some external memory when it needs to remember more than one. This area of memory, the stack is discussed below (page 7.7).

Exercise 2.1

Write a program to put a 3 in the accumulator starting at 828. Then jump to a sub-routine at 900 which adds 3 to this 3 already in A, return to original routine and print the accumulator sum onto the top left hand corner of the screen.

Answer on page 9.2.

Conditional Jumps

We have already looked at unconditional jumps but any program that needs to test for conditions needs CONDITIONAL JUMPS. In BASIC, the analogy is with the IF.....THEN command,

i.e. 10 IF X = Y THEN 500

In this line the values X and Y, which have been stored in memory, are compared.

The 6502 carries out this operation in several different ways one of these is by using a group of single bit registers. These are memory locations within the 6502 chip itself that can store only a '0' or a '1' and are known as the processor STATUS FLAGS . One example of these is the Z flag or the ZERO flag. This is set to '1' when an arithmetic process is carried out that gives a result of zero.

Several different instructions can set this flag one of these being:-

DEX	DEcrement the contents of the X register.

The segment of Program 2.3 below demonstrates this in use.

PROGRAM 2.3

(In Part)

```
START ADD ? 828
LDXIM 100
DEX
```

It loads X with '100' and then decrements this one down to 99. This facility for indexing both the X and Y registers accounts for their name - index registers. When the contents of X are zero, the zero flag is set to 1. If we wish to use the setting of this flag to control the program then we must use an instruction that tests the flag and brings about a branch dependent upon whether or not it is set. Such an instruction is:

BEQ	Branch if result was EQual to zero. (i.e. if the ZERO flag is set.)

This checks on the status of the Z flag and if it is set (i.e. to 1), branches as specified. The operand in this case is only one byte long, so only 0 to 255 can be accommodated. As these 256 numbers are needed for branching in both directions 0 to 127 are assigned to forward jumps, for example '60' giving a forward jump of 60 steps while 128 to 256 are used for backward steps. In the case of the latter a branch instruction of, say 200 gives a backward step of 256 - 200 or 56 steps.

The BEQ instruction is illustrated in Program 2.3 below, where it checks the condition of the Z flag and, if set branches forwards 3 bytes.

PROGRAM 2.3

```
START ADD? 828
LDXIM 100
DEX
BEQ 3
JMP 830
STX 7680
RTS
END
```

When run this program prints an "@" in 7680.

As with many X-register instructions, DEX has a corresponding Y register instruction:-

DEY	DEcrement the contents of the Y register.

Exercise 2.2

Write a program to carry out the same operation as Program 2.2 but utilising the Y register.
Answer on page 9.2.

A second instruction that also checks the Z flag is:

BNE	Branch if Not Equal.

This does the reverse of the BEQ command and branches if the Z flag is NOT set. Program 2.4 below, a modification of program 2.3 is shortened considerably by the use of BNE rather than BEQ.

PROGRAM 2.4

```
START ADD ? 828
LDXIM 100
DEX
BNE 253
STX 7680
RTS
END
```

When run, the program is identical in effect to program 2.3 and puts an "@" in 7680.

Both index registers can be indexed upwards as well as down by DEX and DEY commands. This is done by means of:-

INX	INcrement the contents of X by one.
INY	INcrement the contents of Y by one.

Instructions to compare values

Naturally, when incrementing a straight check for zero is not possible so the registers must be compared against a value previously set somewhere and the 6502 possesses three instructions to do this. First of these instructions to be examined is:-

CPX

ComPare the contents of the specified memory address with the X register.

This is actually done by subtracting the memory contents from X and can, thus, give a positive, negative or zero value. Thus, the instruction CPX 900 does the following:-

1. Read contents of memory location 900.
2. Subtract these contents from those of the X register.
3. Set Z flag if answer = 0. (Also sets other flags not yet considered.)

NOTE:

Neither the contents of the memory location nor the X register are changed during this operation.

At the moment we are interested in the zero condition. To utilise this instruction we can set the X register at zero and store a value for comparison in memory somewhere. The flow diagram for this is given below, in fig. 2.5.

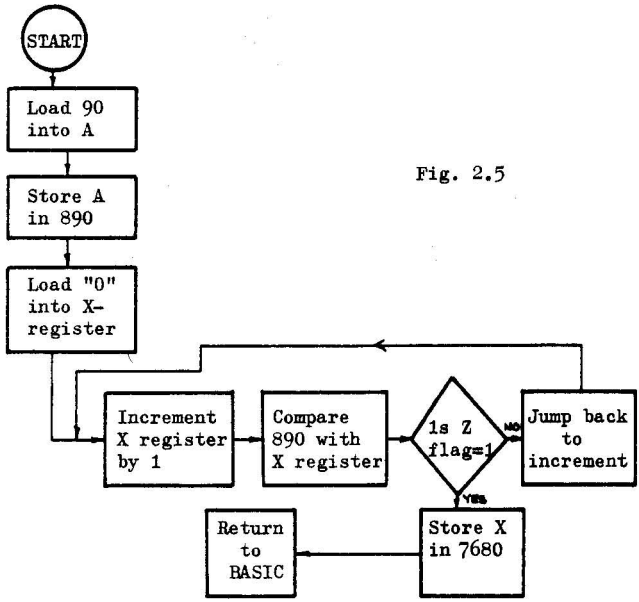


Fig. 2.5

Written into a program it looks like this:-

PROGRAM 2.5

```
START ADDRESS? 828
LDAIM 90        Load 90 (a diamond) into Accumulator.
STA 890         Store contents of accumulator in 890.
LDXIM 0         Load '0' into X register.
INX             Increment X register.
CPX 890         Compare value in X register with that in 890 (i.e. 90).
BEQ 3           Branch forward three bytes if CPX answer = 0.
JMP 835         Jump to memory location 835.
STX 7680        Store contents of X in 7680.
RTS             Return from machine-code to BASIC.
END
```

Just to make that clearer, we'll step through some of the stages. Figure 2.6 below is numbered in stages where stage 1 represents the first time that the program steps through INX, stage 2 the second time and so on.

Note that although X is loaded with a '0' this is immediately incremented to a '1' at INX.

Loop Number	Accumulator contents	X-register contents	Z-flag
1	90	1	0
2	90	2	0
3	90	3	0
etc.			
88	90	88	0
89	90	89	0
90	90	90	1

Fig. 2.6

At stage 90, BEQ is activated and program jumps 3 bytes to STX 7680 and then RTS.

Now type in the program and run it. It should display a diamond in 7680.

The compare instruction CPX has a corresponding instruction for the Y register:-

CPY — ComPare contents of specified memory location with those in the Y register.

Its operation corresponds exactly with that for CPX.

Exercise 2.3

Re-write program 2.5 to use the Y-register rather than the X and on completion of the loop print out a heart at 7780.
Answer on page 9.3.

The third compare instruction is:-

CMP — CoMPare the contents of the specified memory with the Accumulator.

This is particularly useful as the results of all arithmetic operations are deposited in the accumulator and CMP allows a direct comparison between a specified value and an answer.

An example of its use is given in program 2.6.

PROGRAM 2.6

```
START ADD? 828
LDXIM 0          Load a '0' into X.
LDAIM 83         Load an '83' (a heart)
                 into A.
INX              Increment X.
STX 900          Store X in 900.
CMP 900          Compare A with 900.
BNE 247          Branch if Not equal.
STX 7680         Store X in 7680.
RTS              Return from Subroutine.
END
```

Actually, using CPX to put a '1' into the Z flag was rather using a sledge hammer, as this flag is very

readily set. In fact, it can be set to '1' by zero data when such data is passed from memory to the accumulator, when it is passed to X or Y registers or when a Zero answer is obtained to an arithmetic process. Thus, the program 2.5 could have been written omitting the CPX 890 instruction.

To demonstrate this, program 2.5 is re-written using the Y register and DEY and testing with the command BNE, as shown in figure 2.7 below:-

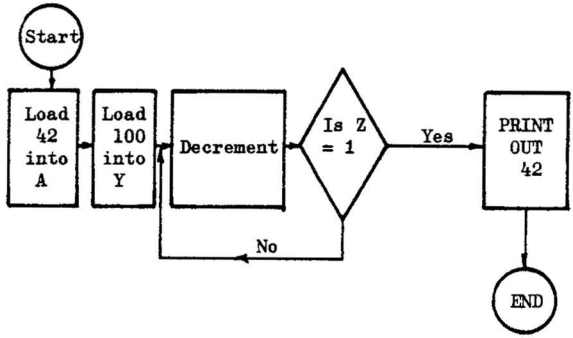


Fig. 2.7

And the program:-

PROGRAM 2.7

```
START ADD?828
LDAIM 42
LDYIM 100
DEY
BNE 253
STA 7680
RTS
END
```

The instruction BNE 253, in this program gives a branch of 256 - 253, i.e. a backward branch of 3 taking the program back to DEY, as the step is counted from the beginning of the next instruction whether the branch is forward or backward.

If you haven't run program 2.7 by now, have a go, it should put an asterisk in 7680!

Exercise 2.4

Now write the program for the flow diagram below, figure 2.8:-

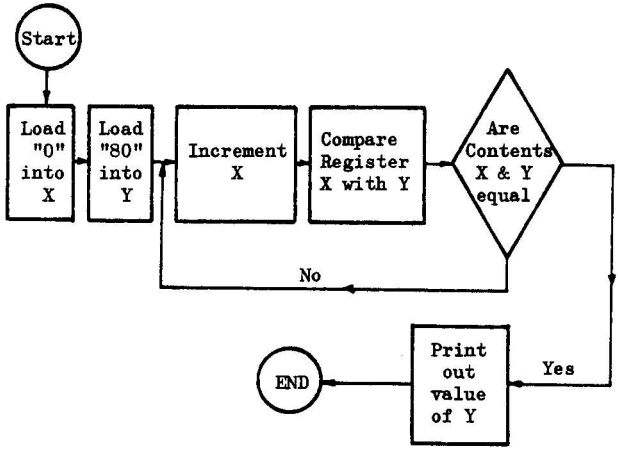


Fig. 2.8

If you run into difficulty with this, the more detailed flow-diagram on page 9.3 should help. PLEASE don't look at this until you've had a try though.

The answer is given on page 9.3.

- - - - - - - -

THE 6502 FLAGS

Program 2.7 used the Z-flag which is only one of 7 flags available on the 6502. As these each only contain one BIT of data, i.e. a '0' or a '1' they can all be stored in one byte of memory - the Processor Status Word. Thus the flags are contained in the PSW as shown below in Figure 2.8.

Bit number	7	6	5	4	3	2	1	0
Flag	N	V	-	B	D	I	Z	C

Fig. 2.8

It is not proposed to give a full description of all these now, they will, however, be described and illustrated when they are encountered. The Z-flag - the Zero-flag - has, of course already been met and used in programs and exercises.

The function of the flags are summarised below:-

N — Negative flag. Set when an arithmetic operation results in a negative result. The flag is controlled by the instruction ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TXA, TYA.

V — Overflow flag. Set when an arithmetic operation results in an overflow from bit 6 to bit 7, tells that result will be wrong unless overflow allowed for. The flag is controlled by the instruction ADC, BIT, CLV, PLP, RTI, SBC.

B — Break flag. Set when a programmed interrupt is brought about by a BRK instruction.

D — Decimal flag. Set when arithmetic operations are to be carried out in decimal. The flag is controlled by the instructions CLD, PLP, RTI, SED.

I — Interrupt flag. Set when an interrupt sequence is in operation. The flag is controlled by the instructions BRK, CLI, PLP, RTI, SEI.

Z — Zero flag. Set when an arithmetic operation results in a zero answer. The flag is controlled by the instructions ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TXA, TYA.

C — The Carry flag. Indicates the presence of a 'carry' or a 'borrow' during arithmetic operations also set during shift or rotate operations to indicate possible loss of a bit. The flag is controlled by ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

The N-Flag

The N-flag, bit 7 on fig. 2.8 above, is the Negative flag which is set to '1' when the answer to an operation is negative. It can be tested by two instructions, one of which is:-

BMI — Branch on MInus.

An instruction such as BMI 27 would test the N flag and if set branch the program forward 27 bytes.

An example of the use of BMI is given in program 2.8, where the contents of Y are incremented until a CPY command gives a minus and triggers a BMI command.

PROGRAM 2.8

```
START ADD? 828
LDAIM 90          Load 90 into A.
STA 900           Store accumulator
                  contents in 900.
LDYIM 0           Load '0' into Y.
INY               Increment Y.
CPY 900           Compare contents
                  of 900 with con-
                  tents of Y.
BMI 250           Branch on minus,
                  i.e. test N flag.
STY 7680          Store contents Y
                  in 7680.
RTS               Return from sub-
                  routine.
END
```

On running, this program should put a diamond in 7680.

- - - - - - - - -

In addition to the BMI command, the N flag is also tested by the command:-

BPL — Branch on PLus.

A command such as BPL 12 would test the N flag and if not set branch the program forward 12 bytes. Program 2.9 uses BPL to test the N flag and to branch when not set. In this program memory location 900 is loaded with 90 and Y loaded with a higher number in this case, 100. Thus after a CPY instruction the negative flag is not set,

i.e. a plus had occurred and the BPL instruction brought about a branch. As the Y register is decremented, however, the CPY test yields a minus at 90 - 91 and the Branch on PLus (BPL) test fails allowing the program to run through to the end.

PROGRAM 2.9

```
START ADD? 828
LDAIM 91
STA 900
LDYIM 100
DEY
CPY 900
BPL 250
STY 7680
RTS
END
```

When run this program will print 90 (a diamond) in 7680.

Exercise 2.5

Write a program using BPL to branch when the X register reaches 0 having been decremented from 90, then print out the current value of X.

A possible answer on page 9.3.

CHAPTER 3

More instructions, addressing, screen outputting.....

One of the great advantages of machine code programs is their speed of operation and this naturally facilitates screen displays. Animation, for instance, can be achieved by means of commands such as:-

STAX — STore the contents of the Accumulator in the specified address indexed with the X register.

Thus, if X contains 100 and the accumulator 90, the instruction

```
STAX 7680
```

will put a diamond into (7680 + 100). When used along with the increment instruction this enables the location on the screen to be indexed. Program 3.1 demonstrates this.

PROGRAM 3.1

```
START ADD? 828
LDXIM       100     Load 100 into X.
LDAIM       90      Load 90 (a diamond)
                    into A.
STAX        7679    Output diamond at
                    ( 7679 + X).
DEX                 Decrement X value.
BNE         250     Branch on not equal.
RTS
END
```

When run, this program puts a diamond in the first 100 locations on the screen. Once again the 'X' command has a corresponding 'Y' command:

STAY — STore the contents of the Accumulator in the specified address indexed with the Y register.

Exercise 3.1

Modify program 3.1 to use the Y register rather than the X register, using only direct POKE commands.
Answer on page 9.5.

Instead of the decrement instruction being used, the program could have used an increment but would then have had to do a compare at 100 in order to set the zero flag to a '1'. Try this as an exercise!

Exercise 3.2

Print an asterisk in the first 100 screen locations using an INX command to increment.
A possible answer on page 9.4.

In exercise 3.2 the branch instruction was activated by zero generated by a compare command. However, if the X or Y register is incremented from 255, it clocks back to zero and resets the Z flag. It can thus be used to branch without a compare if it is initially set to the appropriate value. Program 3.2 performs a similar function to 3.1 but uses INX rather than DEX. It increments X from 216 to 255 and on the 39 loops through, plus the next loop, prints out an asterisk. Once at 255 the 8 bit register is full of '1's and the addition of one more '1' ripples through each bit of the store re-setting them all to '0's. The 6502 does notice this switching over and sets a flag to remember the event (discussed above, page 2.13).

This program offers no advantage over 3.1 on its own but may, in a particular context be advantageous.

PROGRAM 3.2

```
START ADD? 828
LDXIM 216
LDAIM 42
STAX 7464
INX
BNE 250
RTS
END
```

As with BASIC programs a character can be moved across the screen by filling the screen while POKING a blank one space behind it. Program 3.3 demonstrates this type of routine.

PROGRAM 3.3

```
START ADD? 828
LDXIM 0
LDYIM 32
STY 900
LDAIM 90
STA 901
STAX 7680
TYA
STAX 7679
LDA 901
INX
BNE 243
RTS
END
```

When run, the program runs a diamond across the screen to 7934. You may be quite at home with this general technique of animation, if so please ignore Figure 3.1. If not, you should step through this stage by stage.

Command	Acc	X-Reg	Y-Reg	900	901	Screen mem.	Content	Z-flag
LDXIM 0	?	0	?	?	?	?	?	0
LDYIM 32	?	0	32	?	?	?	?	0
STY 900	?	0	32	32	?	?	?	0
LDAIM 90	90	0	32	32	?	?	?	0
STA 901	90	0	32	32	90	?	?	0
STAX 7680	90	0	32	32	90	7680	90	0
TYA	32	0	32	32	90	7680	90	0
STAX 7679	32	0	32	32	90	7679	32	0
LDA 901	90	0	32	32	90	7679	32	0
INX	90	1	32	32	90	7679	32	0
BNE 243	90	1	32	32	90	7679	32	0
STAX 7680	90	1	32	32	90	7681	90	0
TYA	32	1	32	32	90	7681	90	0
STAX7679	32	1	32	32	90	7680	32	0
etc.	until.....							
STAX 7680	90	255	32	32	90	7934	90	0
TYA	32	255	32	32	90	7934	90	0
STAX 7679	32	255	32	32	90	7933	32	0
LDA 901	90	255	32	32	90	7933	32	0
INX	90	0	32	32	90	7933	32	1
BNE 243	90	0	32	32	90	7933	32	1
RTS	90	0	32	32	90	7933	32	1

Fig. 3.1

As written, program 3.3 is by no means the only way of doing the job and most probably is not the best, nevertheless it does flash the diamond to where it's required.

THE TIMING OF PROGRAMS

Program 3.3 highlights one of the problems of machine code - a few pages ago it was an advantage! - SPEED. Whereas in BASIC it's not often necessary to slow things down, that's not so in machine code. The 6502 A chip takes its operating speed from an internal clock driven from a crystal oscillator which, in the case of the VIC runs at 2 MHz (two megahertz) or two million cycles per second. Thus each cycle takes $\frac{1}{2}$ millionths of a second and speeds of operation of the various instructions will be referred to by the number of cycles that it takes for them to be carried out or EXECUTED. Some of these operations take place entirely within the 6502 and are carried out much quicker than thosewhich are required to retrieve data from memory. The instruction TAX for instance takes two cycles to execute while STAX takes six.

Clearly, a knowledge of the time taken for the instructions to be executed is important as it is this that determines the speed of operation of the program and also allows use to be made of the 2 MHz clock for timing loops and delays.

Looking back at Program 3.3, the time that a character remains on the screen prior to "moving on" can be calculated. Thus, the diamond appears at STAX 7680, and the next few stages are tabulated below:-

Command	Execution time (cycles)	Elapsed time (cycles)
STAX 7680	-	0
TYA	2	2
STAX 7679	6	8
LDA 901	4	12
INX	2	14
BNE 243	2	16
STAX 7680	6	22
TYA	2	24
STAX 7679	6	30

So, from a diamond appearing to its being overwritten by a blank is 15 cycles or 15 micro-seconds. Overall, the 256 diamonds are written in about 3850 micro seconds or 3.8 milli-seconds much faster than the eye can follow.

To attain a more leisurely progress across the screen a delay could be programmed in to allow the diamond to stay in view longer. Such a device is simple in principle but may need some care when implementing in an actual application. Program 3.4 below shows a simple delay loop.

PROGRAM 3.4 (Part)

```
START ADD? 828
LDXIM 250          2 cycles.
DEX                2 cycles
BNE 253            3 cycles
RTS
```

This gives (ignoring LDXIM's 2 cycles) a delay of 5 cycles per loop or 250 x 5 = 1250 cycles per execution. Even when run right through, a delay of only 625 micro-seconds is obtained and this must be augmented by nesting this loop within another one as program 3.5 shows.

PROGRAM 3.5 (Part)

```
START ADD? 828
LDYIM 200          2 cycles
LDXIM 250          2 cycles
DEX                2 cycles
BNE 253            3 cycles
DEY                2 cycles
BNE 248            3 cycles
RTS
```

A complete run of this program would run the DEX sub-routine 200 times, i.e. achieving an overall delay of 200 x 625 micro-seconds or $\frac{1}{8}$ of a second.

If we wish to use the computer for precision timing, then we can clearly not ignore the odd two micro-seconds here and there and we must step very carefully through the program to ensure that we account for all parts of the program. In particular we must watch the branch instructions. For instance, the BNE in program 3.5 normally takes three cycles, i.e. when the branch succeeds. However, when the branch fails and the program runs past the instruction it takes only two cycles. Under other conditions, if the branch takes the program into another section of memory (i.e. another "page", see page 3.7

for discussion) the instruction takes an extra two cycles!

However, we were principally interested in delays in order to slow down our animation so let's try putting some delay into a program (3.3) to check that they really do work!

Program 3.6 uses 3.3 as a basis and inserts the delay loop illustrated in 3.4 putting 0.6 milliseconds between the appearance and disappearance of a diamond.

PROGRAM 3.6

START ADD? 828		
LDYIM	0	
LDXIM	90	Setting up.
STX	900	Diamond
LDXIM	32	and blank
STX	901	
LDA	900	Load and Display
STAY	7680	diamond
LDXIM	250	Inserted
DEX		delay
BNE	253	
LDA	901	Load blank into accumulator
STAY	7680	Display blank
INY		Go to next
BNE	236	location
RTS		

In program 3.6, a 250 times loop was used as a longer delay, as that in program 3.5 requires two registers to index. In order to use one register for indexing the character across the screen and in the loop, the index contents could be successively dumped into and then retrieved from memory. When used in this way, the lack of more than two registers in the 6502 can be overcome quite readily - at the expence of a little more coding.

- - - - - - - -

MODES OF ADDRESSING

When moving a character across the screen we have used the instructions STAX and STAY as these are able to index along with the relevant register. In earlier work we used the STA instructions on its own and this is clearly a relative of STAX, STAY, etc. The difference between the two types of instruction lies in their modes of addressing and clearly the X and Y are part of this.

In fact, the STA instruction has seven varieties depending on the mode of addressing used. Thus, the address is a modifier of the command, designed to modify its function in a particular way.

The address essentially points the 6502 to a location in memory either directly or indirectly, the way it does this being determined by the particular mode of addressing used. Addressing is uniform throughout the 64K of possible memory except for the 256 locations from 0 to 255. To address these locations, only one byte is needed whereas all other locations need two. Hence, this area of memory is given a special name - ZERO PAGE - and a special mode of addressing. In fact, the whole of the memory that the 6502 can address is divided up into pages of 256 bytes and an instruction that causes operation over the boundary between pages takes an extra cycle to be executed.

In this book, addresses in Zero page memory are indicated by the use of a 'Z' on the instruction, e.g. STAZ. However, this is only a convention chosen here and is by no means a universal one. All that really matters is that the zero page STA command has the op-code of 85_{16} or 133_{10}. This is achieved with the assembler provided, by the position of STAZ in the Data statement of 133 (x 2).

IMPLIED ADDRESSING

This mode, sometimes also known as inherent addressing is probably the easiest to use, as the 6502 does all the work for you!

Several instructions have already been used such as TYA, TXA, RTS, in which the 6502 itself calculates the address. Basically they form two separate groups, one in which the whole instruction is executed within the 6502 itself, i.e. TYA, transfer Y to A and the other group where an external reference is necessary, e.g. RTS return from subroutine.

The members of the first group already considered are; DEX, DEY INX, INY, TAX, TAY, TXA, TYA while those yet to be discussed are; CLC, CLD, CLI, CLV, NOP, SEC, SED, SEI Those in the second group are RTS (already used) and BRK, PHA, PHP, PLA, PLP and RTI.

ABSOLUTE ADDRESSING

Instructions using this mode are among the easiest to understand as their operand is a two byte number that

defines the address absolutely. Thus in program 3.6, page 3.6, the instruction STX 901 tells the X register exactly where to store its contents. Later in the same program LDA follows the same pattern.

Instructions that utilise this form of addressing are: those already met; ADC, CMP, CPX, CPY, JMP, JSR, LDA, LDX, LDY, STA, STX, STY, while those yet to be discussed AND, EOR, ORA and SBC.

ZERO-PAGE ADDRESSING

This form of addressing is really a sub-set of absolute addressing but is restricted in size of its operand to 255. The major advantage of this mode is that it executes in only three cycles compared with the four required for the absolute modes. In this book those instructions using zero-page modes are suffixed with a 'Z'. Because of the faster execution times when using page zero addresses, page zero of the VIC is used by the BASIC interpreter and is, thus, not readily available for machine code programs. It is possible to use this page, however, by relocating this page to another location in RAM but this is felt to be a procedure beyond the scope of this book.

Three of the locations in page zero that are useful are 160 to 162 which contain the jiffie clock that increments every 1/60th of a second. Program 3.7 is a very simple program that loads one byte of this into the accumulator and then prints it onto the screen.

PROGRAM 3.7

```
START ADD? 828
LDAZ 160        Load accumulator in
                Zero-page mode with
                contents of 160.
STA  7680       Store contents of A
                in 7680.
RTS             Return from machine
                code subroutine.
```

Because of their limited range of use, no zero-page instructions have been used to date in this book. Those that are available for use are: ADCZ, ANDZ, ASLZ, BITZ, CMPZ, CPXZ, CPYZ, DECZ, EORZ, INCZ, LDAZ, LDXZ, LDYZ, LSRZ, ORAZ, ROLZ, RORZ, SBCZ, STAZ, STXZ, STYZ.

IMMEDIATE ADDRESSING

This mode of addressing allows a number to be loaded immediately, i.e. directly as specified, into a register, or to be used directly as a means of comparison. All the immediate commands in this book are recognisable by the 'IM' in their source code. This format is not universal and some other assemblers use the format:

BASIC SOURCE CODE # OPERAND

i.e. instead of LDAIM 90
LDA #90

Many examples of immediate mode addresses have been seen already, for example in program 3.6, page 3.6. In this program, the X register was loaded directly using LDXIM 32, and in other programs both the Y-register and accumulator have been loaded in the same way. However, many other instructions can be used in the immediate mode, resulting in neater programs. Program 3.8 demonstrates the use of:-

CPYIM	ComPare Y with value specified in Immediate Mode.

PROGRAM 3.8

```
START ADD? 828
LDYIM 0          Load Y with '0'.
TYA              Transfer Y to A.
INY              Increment Y.
STAY 7679        Store contents of
                 A in 7679 + Y.
CPYIM 100        Compare Y with 100.
BNE 247          Branch if Z flag
                 not set.
RTS
```

On running, the program prints the first 100 characters of the character set in the first 100 screen locations.

INDEXED ADDRESSING

In this mode an address is calculated using the contents of a register added to a specified address. It has been used frequently to print characters across the screen in the form STAX, STAY. In program 3.8 STAY was used in

this way with the instruction STAY 7680. In some assemblers, the register which indexes the command is appended after the operand thus the command

```
STAX 7680      would be written
STA  7680,X
```

When using indexed addressing the X and Y registers behave differently and care must be taken over their use.

Both registers can be used with absolute indexed instructions, i.e. operands with two bytes.

Those indexed with X are: ADCX, ANDX, ASLX, CMPX, DECX, EORX, INCX, LDAX, LDYX, LSRX, ORAX, ROLX, RORX, SBCX, STAX.

Those indexed with Y are: ADCY, ANDY, CMPY, EORY, LDAY, LDXY, ORAY, SBCY, STAY.

N.B. Exceptions to the generality are that STY CANNOT be indexed with X.

ASL, DEC, LSR, ROL, ROR CANNOT be indexed with Y.

The Zero-page CANNOT be indexed with Y.

The source codes that can be used in zero-page mode have the suffix 'Z' added to this to identify the register used. Thus the instruction ADC becomes ADCZX. The total list of zero-page, X-indexed commands is: ADCZX, ANDZX, ASLZX, CMPZX, DECZX, EORZX, INCZX, LDAZX, LDYZX, LSRZX, ORAZX, ROLZX, RORZX, SBCZX, STAZX, STYZX.

It is,of course, NOT possible to use the two X register index commands STX and LDX with reference to X itself!

RELATIVE ADDRESSING

Many programs used so far, have utilised relative addressing in which a branch has been defined relative to the current position of the program, i.e. the operand expresses the desired displacement. In program 3.8 the instruction BNE 247 was used to test the setting of the Z flag and to branch when it was not set. All the branch instructions used so far

utilise relative addressing and it is this group as a whole that uses this mode. The group consists of:- BCC, BCS BEQ, BMI, BNE, BPL, BVC, BVS.

INDIRECT ADDRESSING

This is by far the most complicated - and the most versatile of all the addressing modes. It gets the "indirect" in the name from the fact that the operand is a pointer and not an address. It is this pointer that directs the 6502 to-wards the memory location that contains the address.

However, once again, the X and Y indexing mechanisms differ considerably in operation and give rise to different sub-modes of addressing. All the instructions that utilise this kind of addressing are recognisable in the assembler as they contain either an 'IX' or an 'IY' and have a one byte operand. Because of this they can only point to locations in the zero-page and hence suffer from the same restrictions as the other zero-page commands.

Using the X-Register

With indirect addressing using the X register, the operand is indexed with (i.e. added to) the contents of the X register to produce a pointer. This location and the one after it are then examined and their contents provide addresses for the required data in the order least significant byte, most significant byte.

The technique is useful for examining one particular element in a table, the actual position in the table being set by the value in the X register. With the lack of availability of spare zero-page space on the VIC the addressing mode is of little use. However, just to demonstrate one instruction in use, program 3.9 uses:-

LDAIX LoaD A Indirectly indexed with X.

This is used to retrieve four bytes stored in page zero from 84 - 88. This is an area of RAM used for numeric storage in program 3.9.

PROGRAM 3.9

```
START ADD? 828
LDXIM 0          Load X immediate with
                 '0'.
LDAIX 87         Load A indirect 84 + X.
STAX  7680       Store A in 7680 + X.
INX              Increment X.
CPXIM 4          Compare X immediate with
                 '4'.
BNE 246          Branch if not equal.
RTS
```

When run, the program will print four characters in the first four screen locations. What these are will differ every time depending upon what BASIC had been doing earlier. When used in a program these would form two addresses in the order

character 1	Address 1	least significant byte
character 2	Address 1	most significant byte
character 3	Address 2	least significant byte
character 4	Address 2	most significant byte.

This type of addressing is known as Indexed Indirect Addressing or perhaps more clearly as PRE-INDEXED INDIRECT ADDRESSING. As the latter name implies the addressing is pre-indexed as the X value is added on before the 6502 picks up the address.

Instructions that use this type of addressing are; ADCIX, ANDIX, CMPIX, EORIX, LDAIX, ORAIX, SBCIX, STAIX.

Using the Y-Register

Indirect addressing using the Y register operates somewhat differently,as the operand of the instruction points directly to a zero-page memory location. This contains the least significant byte of the address, the next memory location containing the most significant byte. Finally, the index register contents are added to this address to form the final indexed address. Not surprisingly this form of addressing is referred to as POST-INDEXED INDIRECT ADDRESSING as the indexing is carried out after the address is retrieved.

Commands that use this type of addressing are; ADCIY, CMPIY, EORIY, LDAIY, ORAIY, SBCIY, STAIY.

INDIRECT ABSOLUTE ADDRESSING

This mode of addressing is used by one instruction only:



JuMP Indirectly Addressed.

It is an absolute instruction in that the operand is a two byte address and can thus address any location in memory. However, it is indirect in that, at that location and the subsequent one it finds the address (LSB first then MSB) for the jump instruction. In the Dr Watson assembler, it is recognised by the mnemonic JMPIA but some other assemblers may represent this with JMP followed by the operand in brackets i.e. JMP(844), i.e. Jump to 844 to find LSB and 845 to find MSB of jump address.

Putting this into a program using JMPIA 844 yields:-

PROGRAM 3.10 (Part)

```
DEC HEX
840 0348 JMPIA 844
843 034B NOP
844 034C 51(81₁₀)
845 034D 03(3₁₀)
846 034E NOP
847 034F NOP
848 0350 NOP
849 0351 STA 7680
852 0354 RTS
```

This program, on meeting JMPIA 844 jumps to 844 to pick up 51 and then to 845 to retrieve 03 and re-assembles the address 0351_{16}(849). This is loaded into the program counter and the program jumps to 849 to execute the program stored there. As program 3.10 (Part) stands it is not possible to program in the 51 and 03 into 844 and 845 as these hex numbers do not represent valid op-codes. The assembler will simply reject them. They can be most easily put in by loading them into a register and then transferring into memory - as is shown in program 3.10(Whole).

PROGRAM 3.10 (Whole)

```
828 LDAIM 90
830 LDXIM $00 51
832 STX 844
835 LDXIM $00 03
837 STX 845
840 JMPIA 844
843 NOP
844 NOP
845 NOP
846 NOP
847 NOP
848 NOP
849 STA 7680
852 RTS
```

When run, this first loads the address $0351 into locations 844 and 845. The JMPIA then retrieves these and jumps to $0351 where it executes the routine stored there. This however, is a routine and not a sub-routine as it was entered with a JMPIA and not a JSR, so the RTS at 852 returns the program to the BASIC program. Once run, this program will have modified itself and put the 03 and 51 into 844 and 845.

ADDRESSING GENERALLY

The whole subject of addressing is clearly a complex one and one to be approached only with care. A basic rule must be to check carefully in Appendix II before using any addressing of which you are not absolutely certain. To some extent, the assembler will assist in weeding out instructions that don't exist but it can't write your programs!

CHAPTER 4

Adding it up, Logical operators .....

On page 1.1, as early as program 1.1, a program was written to add together two simple numbers and displayed these. The "simplicity" of the numbers arises from the fact that they were only single digits and that their answer gave no carries. When larger numbers are added and carries arise, the 6502 handles these by use of its carry or C flag.

This is set automatically when an addition operation is carried out that brings about a carry between the two bytes of a two byte number — clear eh? If not, please read on!

Using one byte, it's only possible to count up to 255, thus if we wish to count beyond this we have to use two bytes. These 16 bits then allow us the count up to 65536. If you have not yet read the section on binary and hexadecimal (page A1.1) then it would be well to do it now! Naturally it would be possible to hold as large a number as there are bytes free, however, we will now consider the use of two bytes which is rather fancily described as DOUBLE PRECISION operation.

If two or more bytes are to be successfully utilised to represent a single number then they must be linked from the first to the second byte by some mechanism. This is the function that the carry flag performs and its operation is tested by the instruction:-

BCC	Branch on Carry Clear.

This tests for the carry flag being clear, i.e. set to '0' and executes a branch if this is so. One precaution is always best observed when testing for this flag, however, that is to ensure that the flag is in the expected state prior to the operation that may modify it. To perform this task is the instruction:-

CLC	CLear the Carry flag.

This sets the carry flag to 'clear' or '0' and is used prior to the process that may reset this in program 4.1:-

PROGRAM 4.1

```
        CLC
        LDAIM 0
        ADCIM 1
        BCC   252
        STA   7680
        RTS
```

NOTE:
As you are well aware by now, the assembler must be given a start address, i.e. "START ADD? From this chapter onwards this first line is omitted in the listing but must be added in.

When run, this program progressively increases the accumulator contents by 1 until at 255 the ADCIM 1 instruction flips the eight '1''s over to eight '0''s and sets the carry flag to 1. Thus when the accumulator is displayed with the instruction STA 7680, it is seen to contain '0' (i.e. an @ on the screen).

- - - - - -

The 6502 has a second test instruction for the carry flag, this being:-

BCS — Branch on Carry Set.

This tests for the carry flag being 'set' i.e. containing a '1', and if the test is positive executes a branch. Program 4.2 illustrates this instruction in use:-

PROGRAM 4.2

```
        CLC
        LDAIM 0
        ADCIM 1
        BCS   3
        JMP   831
        STA   7680
        RTS
```

Once again this program progressively fills all eight bits of the accumulator with '1''s and finally, on flipping these over to '0''s sets the carry flag and

terminates the program. At the end the accumulator contains all '0''s and therefore an '@' is displayed on the screen.

Let's have a go at adding together two numbers larger than 256, we'll take 1257. Firstly we have to calculate the MSB and LSB and to do this, of course, we have to convert the number to a Hexdecimal format. Thus:-

```
INT (1257/4096)      = 0  therefore Right-most
                          character              = 0
INT (1257/256)       = 4  therefore 2nd character = 4
INT((1257-4x256)/16) = 8  therefore 3rd character = 8
(1257-4x256-8x16)    = 5  therefore 4th character = 5
```

Thus

Hex 1257_{10} = 0485_{16}

and its two parts

MOST SIGNIFICANT BYTE	LEAST SIGNIFICANT BYTE
04_{16}	85_{16}

To add together two 1257_{10}'s we must first add the LSB's, check if there is a carry and then add the MSB's taking into account the necessity (or otherwise) of a carry, i.e. add LSB's

```
          85
          85
plus carry 0A
          --
```

```
85
85
16,10 = carry + 0A
```

Then add the MSB's

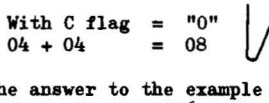
```
          04
        + 04
          --
          08
          --
```

Next add in the carry

08 + carry = 09

In the explanation we have glibly said "+carry" and it is this operation that the C-flag does for the programmer. The flag is set to a '1' when an operation is carried out that leads to a "carry". The next operation that is carried out then takes account of this carry and adds 1

on to the next "add" carried out.



Thus the answer to the example is, in Hex 090A
or, in decimal $9 \times 256 + 10 = 2314_{10}$

Now let's try doing that the long way - using the computer!

We can rely on the 6502 handling the carry but we can't rely on it knowing when to execute a carry! All double precision work is carried out least significant byte (LSB) first as it is in this addition operation that the carry arises and it is then stored up ready for the most significant byte (MSB) addition when this is done. This is why the 6502 uses the notation, LSB first, MSB second - this is the order that they are used.

In order to ensure that the LSB addition is not upset by a carry we must preface an addition by the clear carry (CLC) instruction.

Firstly we must work out the value of MSB and LSB in decimal as both methods of putting data into memory that we have discussed to date require this. For the LSB, its decimal value is

$$8 \times 16 + 5 = 133_{10}$$

and for MSB it is

$$0 \times 16 + 4 = 4_{10}$$

Now to write the program, but - before we do that let's just introduce a new instruction:-

NOP No OPeration

When the 6502 meets it, it does nothing for two cycles of its operation.

We'll see why we put it in shortly, for now type in the program.

PROGRAM 4.3

```
CLC             Clear carry flag.
CLD             Explained later.
LDAIM 133       Load A with LSB.
ADCIM 133       Add with carry 2nd LSB.
STA   7890      Store sum of LSB's in 7890.
NOP
LDAIM 4         Load A with MSB.
ADCIM 4         Add with carry 2nd MSB.
STA   7888      Store sum of MSB's in 7888.
RTS             Return from machine code to
                subroutine.
END
```

Having done this you may run the program, it should put I J on the screen.

The output means: I or 9_{10} and J or 10_{10}

i.e. $90A_{16} = 2314_{10}$

Stepping through that program, the various stages are:

(? signifies Random value.)

Stage	Accumulator	7890	7888	C Flag
START828	?	0	0	?
CLC	?	0	0	0
CLD	?	0	0	0
LDAIM 133	133	0	0	0
ADCIM 133	133	0	0	1
STA 7890	133	10	0	1
NOP	133	10	0	1
LDAIM 4	4	10	0	1
ADCIM 4	9	10	0	1
STA 7888	9	10	9	1
RTS	9	10	9	1

Fig. 4.1

As the above chart shows, at the ADCIM 133, when a carry is generated, the C flag is set at '1' and affects the subsequent ADC. If you want to check this replace, the NOP command with CLC and clear the flag after it has been set.

This can be done by POKEing into 837, the code for CLC i.e. 24.

PROGRAM 4.4

```
POKE 837,24
```

Now, when run, program 4.3 modified by 4.4 will give a display of:-

H J

In this run, the value of J has been computed and when its value of 266 overflowed, 256 was carried the carry bit set and 10 was stored in the accumulator. However, this 256 was lost when the carry bit was later cleared by CLC (Program 4.4)

USING HEXADECIMAL INPUTS

When preparing to write program 4.3 it was necessary to carry out the messy conversions into and out of hexadecimal in order to evaluate the LSB and MSB. However, the assembler possesses a few lines that saves at least one of these tasks, the calculation from hexadecimal back to decimal. Thus, any operand can be entered in hexadecimal, providing it is preceded by a '$' sign. Utilising this in program 4.3, this becomes

PROGRAM 4.3a

```
CLC
CLD
LDAIM $00 85
ADCIM $00 85
STA 7912
NOP
LDAIM $00 04
ADCIM $00 04
STA 7910
RTS
END
```

When run, this gives exactly the same result as program 4.3, i.e. I J. If you select the list option you will find that the assembler has converted the $85 into a decimal value and POKEd this into memory.

Exercise 4.1

Using hex inputs, add together 1807_{16} and $2AFA_{16}$. Verify your program afterwards in base 10. Answer page 9.4.

To re-cap the use of the C flag; it stores the fact that an addition operation has occurred that yielded a carry. If the C flag is left set by mistake then this carry will be added on to the next ADC operation whether you want it or not. For the carry to be carried from LSB to the MSB you have to arrange the data in the appropriate order.

The 6502, of course, has an instruction that carries out subtraction with carry, this is

SBC	SuBtract from the accumulater with Carry, the data at the specified memory location.

e.g. SBC 891

means look in memory location 891 and subtract the number that you find there from the number in the accumulator.

However, in the same way as it was necessary to set the carry flag for addition, it is also necessary to set it for subtraction. Not unexpectedly, though it is necessary to set it the other way, i.e. to '1' rather than to '0'. This instruction is:-

SEC	SEt the Carry bit to '1'.

Let's look at that in a simple program to take 2 from 4. For a change we will use the immediate mode to load the date.

PROGRAM 4.5

```
SEC
LDAIM 4
SBCIM 2
STA 7680
RTS
END
```

By now you should be well enough equipped to try a double precision subtraction on your own so have a go at the exercises below - in case you have a problem exercise 4.1 is explained in fair detail to demonstrate the carry operation. (page 9.4)

Exercise 4.2

Write a program to subtract 600 from 800 using ABSOLUTE ADDRESSING. Store the data in memory locations 890 onwards. Display the answer in 7888.
Answer on page 9.5.

Exercise 4.3

Write a program to subtract 500_{10} from the sum of 300_{10} and 400_{10}.

Display the answer in 7910/1 in the order LSB / MSB.

Answer on page 9.6.

MULTIPLICATION

The arithmetic instructions available for the 6502 allow additions and subtractions to be carried out but no provision is made for multiplication. This has to be carried out, therefore, by a series of repeated additions. For example, the sum 2 x 3 can be expressed as 2 + 2 + 2 and is, thus relatively simple to evaluate. The process is thus one of adding 2 to 0 three times and requires the three to be set up in a loop to define the number of passes through this.

The process is illustrated in Fig. 4.2 below where Y = number of passes through the loop, in this case 3, and A = current sum.

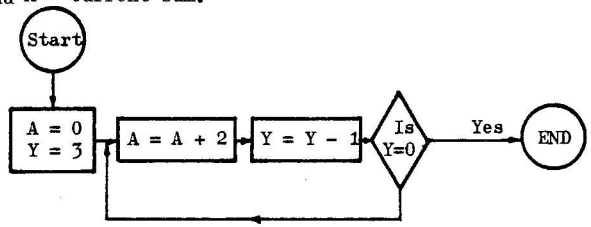


Fig. 4.2

It is put into program 4.5 below:-

PROGRAM 4.5

```
LDYIM 3
LDAIM 0
ADCIM 3
DEY
BNE 251
STA 7680
RTS
```

When run, the program gives an 'I' (a VIC '9') in 7680. Within this program, they key segment is

PROGRAM 4.5a

```
┌→ADCIM 3
│  DEY
└─BNE  251
```

This small loop that does the work is known as an ALGORITHM. One limitation of this simple algorithm is that it can only handle an answer up to 255, after that, it generates a carry and clocks the accumulator back to zero. If no account is taken of this carry, then the answer has lost 256 for each loop! When elaborated somewhat, this algorithm can handle double precision multiplication. This can be achieved by checking for a carry after each addition and if a carry is generated, adding one onto the MSB. One way of carrying out this incrementing is by means of the instruction.

INC	INCrement the contents of the specified memory location.

Program 4.6 shows the algorithm 4.5a elaborated to record the number of carries generated and to increment the MSB to record these.

PROGRAM 4.6

```
        LDYIM 0   )     Set up 905 to receive carry bits.
        STY   905 )
        LDYIM 17        Set up number of times around loop.
        LDAIM 0         Clear accumulator to receive sum
                        of LSB's.
    ┌──►CLC           Clear carry prior to adding.
    │   ADCIM 16      Add 16 to A once.
    │ ┌►BCC   3
    │ │ INC   905     Increment carry record.
    │ └─DEY
    └───BNE   245     Check if Y decremented to zero.
        STA   7682    Display LSB sum.
        LDA   905 )
                      Display MSB.
        STA   7680 )
        RTS
```

When run, this should display A P or 256 + 16 i.e. 16 x 17.

In the example used - 16 x 17 - it made little difference whether the algorithm was repeated 16 or 17 times, however, if the sum 2 x 1000 was attempted then one possible way would be much quicker than the other. This could be tackled by writing a short sub-routine that ensures that the multiplier assumes the smallest of the two values and, thus, the algorithm is cycled the least possible number of times. However, on page 4.21 another method of multiplication will be introduced - binary multiplication - and this keeps the number of iterations down to the minimum.

DIVISION Division using the 6502 is a process of repeated subtraction in the same way that multiplication is one of repeated addition. It is illustrated in program 4.6a in which 30 is divided by 2. In this the accumulator is used to store the running remainder i.e. starts with 30 and progressively declines to zero (30, 28, 26.....4,2,0). The X register is used to load the divisor into memory while the Y register counts the number of times that the subtraction can be made.

PROGRAM 4.6a

```
LDYIM 0          INY
LDXIM 2          CMP 900
STX   900        BCS 247
LDAIM 30         STY 7680
SEC              STA 7682
SBCIM 2          RTS
```

When run, this displays the quotient 15 (as an O) in 7680 and the remainder 1 (as an A) in 7682.

BINARY CODED DECIMAL ARITHMETIC

In addition to numbers being represented in binary and decimal notation, a hybrid or mixed notation, binary coded decimal or BCD, also exists. The usage and format of this is described in Appendix 1, page A1.7. BCD forms a bridge between the two notations and, in many cases greatly facilitates output. Fortunately for the 6502 programmer, the 6502 itself can handle BCD arithmetic and is turned on to the BCD handling mode by the instruction

SED	SEt Decimal mode of operation.

This instruction sets the D flag automatically to a '1'and thereafter arithmetic is done in BDC. When the decimal mode of operation is no longer required it is cleared with the instruction

CLD	CLear the Decimal flag.

This sets the flag back to a '0' and thereafter arithmetic is done in binary.

A simple program to add together 1 and 2 using BCD is given in program 4.7.

When run, program 4.7 puts a 'C' in 7680. One thing to note when using BCD arithmetic is that for safety's sake the decimal flag should be cleared after use, this was done with a CLD just before the RTS.

The example given in program 4.7 is identical in effect to the arithmetic that we have done to date with the exception that in BCD, the carry occurs after each half-byte exceeds 9. This is demonstrated below in Program 4.8, which adds together two 6's. If program 4.7 is still in 828 then 4.8 can be POKEd in by

```
POKE 831,6
POKE 836,6
```

PROGRAM 4.7

```
SED
CLC
LDAIM 2
STA   900
LDAIM 1
ADC   900
STA   7680
CLD
RTS
```

PROGRAM 4.8

```
SED
CLC
LDAIM 6
STA   900
LDAIM 6
ADC   900
STA   7680
CLD
RTS
```

When run, program 4.8 puts an R in 7680 which is its way of saying 12! This comes from the way that BCD is stored in memory as NIBBLES. The 'R' itself comes from the VIC's poke code of 18, which in binary is:

$$18_{10} = 00010010_2$$

However, the memory location is storing two nibbles rather than one byte and hence:

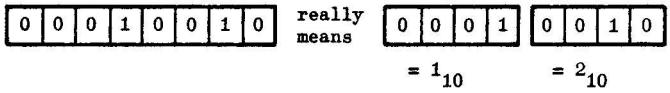


Fig. 4.3

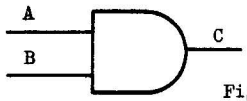
i.e. the number represents 1 x 10 + 2 x 1 or 12_{10}.

The above example emphasises the problems that arise when thinking in decimal and working in binary. As the 6502 itself works in binary and won't easily be persuaded to change, it is necessary to master the techniques of binary number processing.

If we look at the answer to program 4.8, in which the answer consisted of two nibbles held within one byte, we see one of the problems of bit manipulation. What was needed in this case was a technique for modifying individual bits within a byte. In order to extract the lower order nibble from the binary number it is only necessary to erase the higher order nibble i.e. to fill it with '0's. This can be done with an instruction:

AND	Perform a logical AND between the accumulator and the data specified.

An AND is a logical operator that compares two logic states and produces an output based on the comparison. If we examine a logic AND gate such as is used in electronic circuitry it makes the AND function clearer.



An Electronic AND

Fig. 4.4

Figure 4.4 shows an AND gate with two inputs A and B and an output C. Its function is such that if both inputs, A AND B are set at '1', then its output C is a '1'. However, if either or both of its inputs A AND B '0' then its output is '0'.

This is normally expressed in what is known as a TRUTH TABLE, that for the A, B, C AND gate in fig. 4.4 being shown in fig. 4.5.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table for Electronic AND

Fig. 4.5

To use the table, the value of C - the output - is read off for the appropriate inputs of A and B. Thus, taking an A input of '0' and a B input of '0' the output C is 0.

Exercise 4.4

Using the truth table, fig. 4.5, work out the logic output (C) obtained for the following inputs:-

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

Answer on page 9.6.

When an AND is performed by the 6502, it operates on all eight bits in the accumulator simultaneously. Thus if 255 is ANDed with 1 then:

$$255_{10} = 11111111_2$$

$$1_{10} = 00000001$$

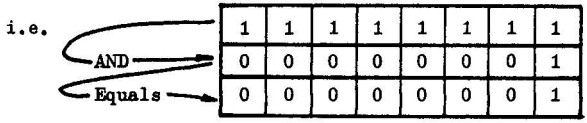


Fig. 4.6

The result of the operation is that all the bits ANDed with a '0' have been stripped off leaving only the first bit.

Exercise 4.5

What is the result when 149_{10} is ANDed with 52_{10}?
If you have problems, work it out bit by bit using the truth table.

Answer on page 9.6.

As the above exercise shows the AND instruction can be used to strip BITS from a number and could be used to convert part of the BCD 12 from program 4.8. This BCD '12' was stored as two nibbles in one byte. If the Most Significant Nibble (MSN) could be changed into four zeros then the byte would read out directly as the value of the Least Significant Nibble. Such a masking out of bits can be done by using an AND command as any '1' in the ANDing number will leave '1's in the number ANDed as they were while '0's in the ANDing number will switch any '1's in the ANDed number to '0's.

Let's try that with the BCD 12.

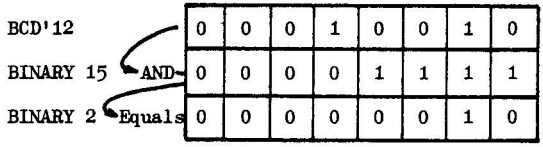


Fig. 4.7

By ANDing the BCD number with 00001111_2 (binary 15_{10}) the four most significant bits have been erased and the number converted into the LSN (in this case 2_{10}).

In a program the AND instruction may be used with several different addressing modes, the absolute mode being illustrated in:-

PROGRAM 4.9

```
LDXIM 15        Load X with '15'.
STX   900       Store X in 900.
LDAIM 18        Load A with '18'.
AND   900       AND A with 900.
STA   7680      Store A in 7680.
RTS
```

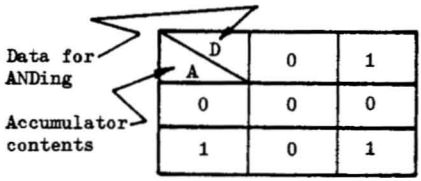
When run this will display 'B' in 7680.

Using immediate addressing, ANDIM, program 4.9 can be re-written as below:-

PROGRAM 4.9a

```
LDAIM 18
ANDIM 15
STA   7680
RTS
```

When defining the truth table of the AND instruction it is written somewhat differently from that used in electronics. This is because the result of the AND operation is deposited back in the accumulator i.e. this forms both part of the input and the output. The truth table for AND is given in Figure 4.8 below:-



Truth table for AND

Fig. 4.8

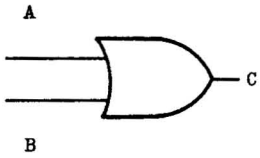
The 6502 chip also uses two other logical operators, one of these being an OR function. With the Mnemonics used in this book it is referred to as ORA

i.e. ORA(ccumulator)

It is defined formally as:-

ORA	Perform a logical inclusive OR between the Accumulator and the data specified.

In electronic circuitry, the OR is depicted as shown in Fig. 4.9.



An Electronic OR

Fig. 4.9

Its mode of operation is similar to that of the AND in reverse, in that if a '1' is present on A OR B then the output C is set to a '1'. The truth table for the 6502's ORA command is given below on figure 4.10.

A \ D	0	1
0	0	1
1	1	1

Truth table for OR

Fig. 4.10

In action the ORA command has the following effect:

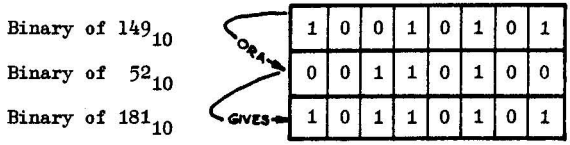


Fig. 4.11

Putting that into a program gives:-

PROGRAM 4.10

```
LDAIM 149
ORAIM 52
STA   7680
RTS
```

When run, this program will put a 181 (a reverse-field 5) in 7680.

As with AND, ORA has several modes of addressing to suit different applications.

Third among the logical operators is:

EOR	Perform a logical Exclusive OR between the accumulator and the data specified.

This operation is probably the least easy to understand and is best illustrated by means of the truth table; fig. 4.10.

A \ D	0	1
0	0	1
1	1	0

Truth table for EOR

Fig. 4.12

Using this instruction with the above example, i.e.

149_{10} EORed with 52

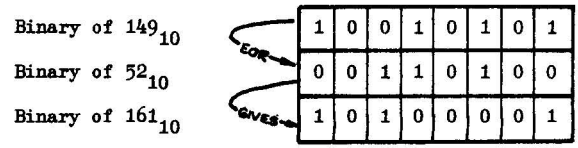


Fig. 4.13

The program to demonstrate that is below:

PROGRAM 4.11

```
LDAIM 149
EORIM 52
STA   7680
RTS
```

When run, the program will put a 161 (a reverse field "!") in 7680.

EOR, like the other logical operators has several modes of address to facilitate its use in programs.

Exercise 4.6

Calculate the result of the following logical operations:

(i) 100_{10} ANDed with 87_{10}.

(ii) 75_{10} ORed with 27_{10}.

(iii) 99_{10} EORed with 57_{10}.

(iv) 94_{10} EORed with the result of 100_{10} ANDed with 87_{10}.

Write a program to verify each operation.

Answers on page 9.7.

Other Forms of Bit Manipulation

Other 6502 instructions exist that enable one to manipulate bits within a byte and as a group these lead to the movement of bits to the right or left within the byte itself.

In the earlier example using BCD arithmetic, the AND instruction was able to isolate the LSN from the byte. It was not possible, though to extract the MSN using the available logic. This does become possible, however, using one of the bit manipulation commands:

LSR	Logical Shift of the specified contents one bit to the Right.

When this is done the bits are moved along one place to the right with the right-most bit falling off into the carry and a '0' being put into the left-most bit. Thus the LSR command operating on 149_{10} gives the following:-

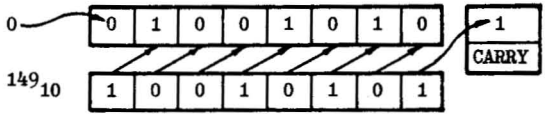


Fig. 4.14

As with most 6502 commands LSR has several addressing modes and the particular address used informs the 6502 where the data which is to be shifted resides. Thus

LSRA means logical shift right of data in A.
LSR 900 means logical shift right of data in 900.

Using the Accumulator mode LSRA to prove the above example is:

PROGRAM 4.12

```
        LDAIM 149
        LSRA
        STA   7680
        RTS
```

When run, this will put a 74_{10} (in 7680)

By using four such right shifts the MSN nibble in a byte can be moved into the place of the least significant nibble and the left-most four bits filled with 0's. This enables one to isolate the MSN in a BCD calculation.

This is demonstrated in program 4.13, this time using LSR in its absolute mode.

PROGRAM 4.13

```
LDYIM 18  )
STY   900 )   Load '18' into 900.
LDYIM 4       Set loop counter.
LSR   900 )
DEY       )   Shift 900 four
BNE   250 )   places to right.

LDA   900 )
STA  7680 )   Print out contents of 900.
RTS
```

When run, the program will print a 1 ("A") in 7680.

Exercise 4.7

The answer to a problem in BCD is 86_{10}.

Write a machine-code program to decode this and display the answer in decimal (POKE form i.e. characters) in 7680 and 7681.

A possible answer on page 9.9.

A further 6502 instruction mirrors LSR in that it moves the bits to the left, it is:

ASL	Arithmetic Shift Left: Shift the specified contents one bit to the left.

When this is implemented the left-most bit is shifted into the carry bit and bit 0 is loaded with a '0'. An ASL instruction, operating on 149 gives:-

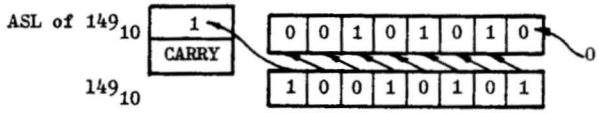


Fig. 4.15

Using the various modes of addressing, ASL can operate on data in different locations, e.g.

ASLA means Arithmetic Shift Left on data in A.
ASL 900 means Arithmetic Shift Left on data in 900.

Using the accumulator mode to test the above example is :-

PROGRAM 4.14

```
LDAIM 149
ASLA
STA 7680
RTS
```

When run, this puts a 42 (an asterisk) in 7680.

BINARY MULTIPLICATION

We have seen from programs 4.5 and 4.6, that multiplication can be carried out using a repetitive or RE-ITERATIVE process but we have also seen that this is a lengthy process and, at times very tricky. However, we tackled the problem very much from a point of view of conventional arithmetical processes and binary has its own way of doing these things! As the 6502 itself thinks in binary and has a number of instructions for manipulating the bits within its bytes, binary arithmetic has a lot to offer.

Firstly let's examine our way of doing decimal (conventional) arithmetic. Take the sum 13 x 14. We define 13 as the MULTIPLICAND and 14 as the MULTIPLIER and lay the multiplication out as below:-

```
  13      Multiplicand
x 14      Multiplier
  52
 130
 182
```

Fig. 4.16

In this conventional format, we first multiply the multiplicand by the lowest digit of the multiplier and store this as the first partial product
i.e. 4 x 13 = 52. Next we multiply the multiplicand by the second digit of the multiplier i.e.
i.e. 1 x 13 and then multiply this by 10 to obtain the second partial product
i.e. 13 x 10 = 130. Thus the total answer is the sum of the two parts i.e. 52 + 130 = 182. Actually in the

second stage it would have been more correct to say that we multiplied the multiplicand by the second digit of the multiplier and then multiplied the result by the BASE (which happened to be 10). The total answer was then the sum of the two partial products.

It is quite possible to perform the same multiplication process using numbers in binary format.

For example, to multiply 5 x 7 in binary,

5_{10} = 010 and 7_{10} = 0111 (working only to 4 bits)

i.e. 7 = 0111
x 5 x 0101

Partial Product 1	0111	Current total 1 = 0111
Partial Product 2	00000	Current total 2 = 0111
Partial Product 3	011100	Current total 3 = 011011
Partial Product 4	0000000	Current total 4 = 011011
ANS =	100011 =	$32 + 2 + 1 + 35_{10}$

Fig. 4.17

With all digits in the multiplier equal to one, all the significant bits in the partial products parts have the same pattern of digits as the multiplicand i.e. "111". Thus the multiplication process in binary reduces to one of successive addition following movement to the left of multiplicand.

8-BIT MULTIPLICATION

The block diagram for this process is given in figure 4.18, where Answer = ANS, D = Multiplicand, R = Multiplier, N = current bit number, LSB = least significant bit of multiplier.

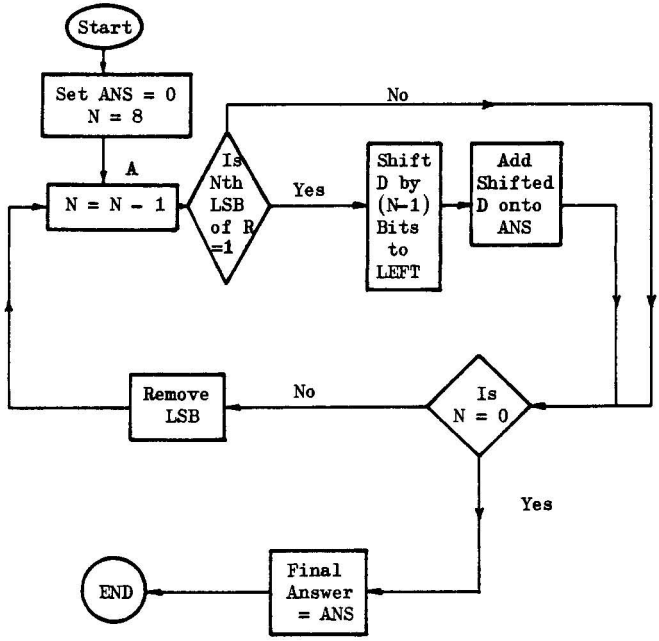


Fig. 4.18

Working through this flow diagram for the example of $2_{10} \times 2_{10}$

$$D = R = 00000010_2$$

Successive passes past A are referred to as A1, A2 etc. For this very simple example with no carries only 4 passes through the loop will be made (only 2 would suffice as only the 2 right-most digits are significant.

START	ANS = 0	N = 0
A1	N = 1	1st LSB = 0 $N \neq 8$
A2	N = 2	2nd LSB = 1
	Shift R (N-1) bits(i.e. 1) to left i.e. $0010_2 \longrightarrow 0100_2$	
	add R to ANS i.e. ANS = 0 + 0010_2 = 0010_2	
	$N \neq 8$	
A3	N = 3	3rd LSB = 0 $N \neq 8$
A4	N = 4	4th LSB = 0 $N \neq 8$

The process then carries on uneventfully for the remaining four zero bits, adding nothing to the ANS.

Thus the answer = 0010_2 = 4

Putting this into a program gives:-

PROGRAM 4.15

```
        LDXIM 2     )
        LDYIM 8     )
        STX   901   )   Initialisation
        STX   902   )
        LDAIM 0     )
    ┌──►CLC
    │
    │   LSR   902 )
    │  ►BCC   4   )   Check if multiplication necessary.
    │   CLC       )
    │   ADC   901 )   'Multiply' if necessary.
    │  ►ASL   901 )
    │   DEY       )   Check if all bits checked.
    └──►BNE   240 )
        STA   7680
        RTS
```

When run, the program will print a D (4) in 7680. It may be checked by changing the LDXIM instruction to input a different multiplicand and multiplier.

Exercise 4.8

Re-write program 4.15 so as to multiply two different numbers together.

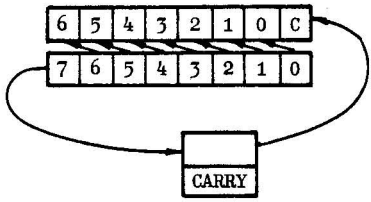
One possible answer on page 9.9.

Unfortunately, program 4.15 is only really half true as an eight bit multiplication routine and works only over a range of small numbers for the multiplier and multiplicand. In the full routine the multiplicand is multiplied by the base eight times by the instruction ASL. Thus, by the eighth shift the right-most bit would have fallen off the left-hand end of the register. In fact, with the number 2_{10}, worked through in the example, the second bit (representing 2^1 or 2_{10}) would be lost after seven shifts. However, this did not affect the overall result because after two LSR's of the 00000010, all the "1"'s had been cleared and all subsequent partial sums were equal to zero.

Were program 4.15 to be used with a larger number where the final answer involved a carry (i.e. was greater than 255) then this loss of left-most bits would be significant and the answer obtained would be wrong. Fortunately, when the left-most bit falls out of a register during an ASL it does not just fall into space but is caught in the carry. Thus the problem is one of retrieving this and transferring it to the MSB of the answer. This can be achieved by means of the command

ROL	ROtate Left the contents of the specified address.

In this operation, all the bits of the specified address are rotated left with the carry bit being loaded into the right-most bit and the left-most bit being transferred to the carry.



The ROL Operation

Fig. 4.19

As a rotation involves the carry bit, it is known as a 9-bit rotation and provides a means of picking the carry bit back up again. Using ROL, enables one to write an

eight bit multiplication accommodating the carry but produces a rather complex program. Such a program is listed in chapter 5 (page 5.7) as, to make it clearer to follow, both labels and memory labels have been used.

The instruction ROL has a right-handed colleague:

ROR	ROtate Right the contents of the specified address.

Both these instructions can be addressed in several ways and take forms such as

ROLA	ROtate Left the contents of the Accumulator.
RORA	ROtate Right the contents of the Accumulator.

Such forms will be described when utilised.

One other instruction is available for bit manipulation:

BIT	AND specified content's BITs with accumulator.

Thus,BIT 900 performs a logical AND between the bits in the specified memory location, i.e. 900 and the accumulator. It also has a zero-page version BITZ.

While BIT performs the same logical function as AND, it differs in that it leaves both accumulator and memory as they are. It does, however, modify the relevant flags in the PSW in the following way:

The Z flag is set if the result of the ANDing is a zero.

The N flag is loaded with the original contents of bit 7 of the location being tested.

The V flag is loaded with the original contents of bit 6 of the location being tested.

Using these binary instructions a process analogues to binary multiplication can be carried out.

8 BIT BINARY DIVISION

This process is analogous to the binary multiplication routine, needing only 8 re-iterations to handle an 8-bit number. It is illustrated in program 4.15a where the dividend (in this case 31) is stored in location 900 and the divisor (2) in 901. The Y register is used as the loop counter to ensure that 8 passes are made through the algorithm. By means of an ASL and an ROLA instruction, the remainder is built up in the accumulator.

PROGRAM 4.15a

```
LDXIM 31
STX   900
LDXIM 2
STX   901
LDYIM 8
LDAIM 0
ASL   900
ROLA
CMP   901
BCC   6
SBC   901
INC   900
DEY
BNE   238
LDX   900
STX   7680
STA   7682
RTS
```

When run, this will display the quotient 15 (as an 0) in 7680 and the remainder 1 (as an A) in 7682.

CHAPTER 5

The Assembler

The assembler listed in the Introduction, here referred to as ASSEMBLER 1 is a very rudimentary device designed primarily to make learning machine code easy. However, as programs increase in complexity further features are demanded of an assembler and some of these are listed in program 5.1. When added to ASSEMBLER 1, this produces a second level of assembler here referred to as ASSEMBLER 2. This is the form that is supplied on cassette or disk and needs about 5.5K of memory, i.e. a 3K memory add-on. The extra program will give you extra features at the cost of memory and on a 6.5K machine represents just about the maximum of assembler that can be fitted into memory.

When making additions to your assembler some lines will be over-written and replaced - this is intentional. The features that ASSEMBLER 2 will provide are:

LABELS

The use of labels enable a program to be directed to a named instruction without the necessity of calculating branches or jump addresses. A fancier term for label is SYMBOLIC LABEL as the label itself is symbolic of a location in memory. For instance, the instruction

```
BNE  LOOP1
```

instructs the assembler to create the machine code that tells the 6502 to branch to an instruction labelled LOOP1. Thus `LOOP1  STAX 7680` creates a label called LOOP1, whose address is the same as the "STAX" in STAX 7680. In order to tell the assembler that a label is a label and not an instruction is preceded by an asterisk (*). This rule is only a convention that has been chosen when writing this particular assembler. Thus the beginning of loop 1 would be entered

```
*LOOP1 STAX 7680
```

Further conventions must be observed when using labels particularly those concerning spaces. The asterisk, for instance, must be followed immediately by the label (no space between). It may be as long as required but must NOT contain any spaces. There is no particularly

ASSEMBLER 2

© Copyright
P HOLMES 1982

PROGRAM 5.1

```
100 POKE36879,41:POKE55,56:POKE56,29
200 PRINTTAB(7);"MON....4■"
230 ONVAL(Z$)GOTO250,640,920,1940
280 IFX$="]"ANDAA$="["THEN2040
390 GOSUB400:GOTO480
400 N=5
500 NEXTX:RESTORE:IFG1$=""ANDB>0THENPRINTI$"6":GOTO260

780 PRINTL;G$;TAB(9)"I";E$;" ";C1:PRINTTAB(9+LEN(STR$(L))-4);
790 PRINT"I■";H1$;" ";H2$"■":Y=Y+1:L=L+2:GOTO650
870 PRINTL;G$;TAB(9)"I";E$;G:PRINTTAB(9+LEN(STR$(L))-4);
880 PRINT"I■";H1$;" ";H2$;" ";H3$"■":Y=Y+1:L=L+3:GOTO650

1300 A=A+1:I=0:Z$=LEFT$(X$,1):IFZ$="*"THEN1340
1310 IFZ$="@"THENI=1:GOTO1340
1320 A=A-1:IFZ$="+"THEN1460
1330 GOTO300
1340 D=D-1:X$=RIGHT$(X$,D)
1350 FORZ=1TOD
1360 IFMID$(X$,Z,1)<>" "THENNEXT
1370 IFZ=D+1ANDI=0THENPRINTI$;"3":GOTO260
1380 IFZ=D+1ANDI=1THENPRINTI$;"6":GOTO260
1390 F$=LEFT$(X$,Z-1):Z=D:NEXTZ
1400 F$(A)=F$:F(A)=L
1410 Z=LEN(F$):Z=D-Z-1:D=Z:IFI=1THEN1440
1420 X$=RIGHT$(X$,Z)
1430 GOTO300
1440 A$=RIGHT$(X$,Z)
1450 F(A)=VAL(A$):GOTO260
1460 D=D-1:X$=RIGHT$(X$,D)
1470 IFM=0THEN1700
1480 IFX$="END"THEN1720
1490 FORX=1TOM
1500 IFX$<>M$(X)THENQ=0:GOTO1520
1510 Q=X:X=M
1520 NEXT:IFQ=0THEN1700
1530 A1=0
1540 A2=0:H=-1:FORX=M%(Q)TOM1%(Q):H=H+1
1550 FORZ=1TOA
1560 IFX<>F(Z)THEN1590
1570 A1=A1+1:F$(A+A1)=F$(Z)+STR$(M)
```

continued.....

```
1580 F(A+A1)=L+H:Z=A:A2=A2+1
1590 NEXTZ
1600 K=PEEK(X):POKEL+H,K:NEXTX:A=A+A2
1610 REM LABEL REFS
1620 A1=0:H=-1:FORX=M%(Q)TOM1%(Q):H=H+1
1630 FORZ=1TOE
1640 IFX<>K%(Z)THEN1680
1650 A1=A1+1:K$(E+A1)=K$(Z)+STR$(M)
1660 K%(E+A1)=L+H
1670 K1%(E+A1)=K1%(Z)
1680 NEXTZ
1690 NEXTX:L=L+H+1:E=E+A1:GOTO260
1700 M=M+1
1710 M$(M)=X$:M%(M)=L:GOTO260
1720 M1%(M)=L-1:GOTO260
1730 REM LABEL REFERENCES
1740 L$=RIGHT$(X$,D-J)
1750 E=E+1
1760 K$(E)=L$:K%(E)=L+1:K1%(E)=B
1770 RETURN
1780 IFE=0THEN110
1790 FORX=1TOE
1800 J$=K$(X)
1810 FORZ=1TOA
1820 IFJ$<>F$(Z)THEN1910
1830 L=K%(X)-1
1840 G=F(Z)
1850 IFK1%(X)=1THEN1880
1860 GOSUB620
1870 GOTO1910
1880 H=G-L-2:IFH>129ORH<-126THENPRINTI$;"4"
1890 IFH<0THENH=256+H
1900 POKEL+1,H
1910 NEXTZ:NEXTX
1920 GOTO110
1930 RETURN
1940 PRINT"■■   VIC MONITOR■"
1950 GOTO2040
1960 C=L:P=1:GOSUB800
1970 PRINT"  "G$;:U=-1
1980 C=L:FORU=0TO4
1990 C=PEEK(L+U):P=3:GOSUB800:PRINT" ";G$;
2000 NEXT
2010 PRINT:L=L+5
2020 IFL<=RTHEN1960
2030 IFR1>0THENR=R1:GOSUB940:GOTO1950
2040 INPUTZ$
2050 AA$=LEFT$(Z$,1)
```

continued.....

```
2060 IFAA$="M"THEN2220
2070 IFAA$="S"ORAA$="L"THEN2260
2080 IFAA$="G"THEN2470
2090 IFAA$="X"THEN110
2100 IFAA$="?"THENZ$=RIGHT$(Z$,LEN(Z$)-2):GOTO2050
2110 IFAA$="["THENL=VAL(RIGHT$(Z$,(LEN(Z$)-1))):GOTO260
2120 X$=LEFT$(Z$,2)+" "+MID$(Z$,3,2):J=-1
2130 GOSUB400:G1=G
2140 FORXX=0TO15STEP3:X$=MID$(Z$,6+XX,2)
2150 S$(0)=RIGHT$(X$,1):S$(1)=LEFT$(X$,1)
2160 G=0:FORX=1TO0STEP-1
2170 FORZZ=0TO15
2180 IFS$(X)=MID$(D$,ZZ+1,1)THENT=ZZ:ZZ=15
2190 NEXTZZ:G=G+T*16↑X:NEXTX
2200 POKE(G1),G:G1=G1+1:NEXTXX
2210 GOTO2040
2220 X$=MID$(Z$,3,2)+" "+MID$(Z$,5,2):J=-1:GOSUB400
2230 L=G
2240 X$=MID$(Z$,8,2)+" "+MID$(Z$,10,2):J=-1:GOSUB400
2250 R=G:GOTO1960
2260 AB=3:NA$=""
2270 IFMID$(Z$,2,1)<>CHR$(34)ORAB>LEN(Z$)THENPRINTI$;"?":GOTO2040
2280 IFMID$(Z$,AB,1)=CHR$(34)THEN2320
2290 NA$=NA$+MID$(Z$,AB,1):AB=AB+1
2300 IFMID$(Z$,AB,1)=CHR$(34)THEN2320
2310 GOTO2270
2320 IFAA$="L"THEN2400
2330 X$=MID$(Z$,AB+2,2)+" "+MID$(Z$,AB+4,2):J=-1:GOSUB400:SA=G
2340 X$=MID$(Z$,AB+7,2)+" "+MID$(Z$,AB+9,2):J=-1:GOSUB400:EA=G
2350 OPEN1,1,1,NA$
2360 PRINT#1,SA
2370 PRINT#1,EA
2380 FORX=SATOEA:MC=PEEK(X):PRINT#1,MC
2390 NEXT:CLOSE1:GOTO2040
2400 OPEN1,1,0,NA$
2410 INPUT#1,SA,EA
2420 FORX=SATOEA
2430 INPUT#1,A
2440 POKEX,A:NEXTX
2450 CLOSE1
2460 GOTO2040
2470 X$=MID$(Z$,3,2)+" "+MID$(Z$,5,2):J=-1:GOSUB400:SYS(G)
2480 GOTO2040
```

technical reason for this, it is simply that the assembler looks for a space in order to work out how long the label is. For this reason the label must be followed by a space and then a normal instruction. These rules may sound a little formidable but don't worry, the assembler will pick up any errors and let you know what is wrong with any particular line. When referring to a label in an instruction it is only necessary to replace the operand with the label itself.

To summarise: LABELS

i) A label is defined by the asterisk (*) that precedes it.

ii) The label may be of any length but it is as well to stick to about six characters.

iii) There must be no gap between the asterisk and the label.

iv) The label must be followed by a gap prior to the instruction.

It all sounds a little complicated so let's see it illustrated in a program. This uses two loops, simply called "LOOP1" and "LOOP2" and does some unnecessary branching and jumping by way of illustration.

PROGRAM 5.2 (In Assembly language)

```
        LDAIM 90
        LDXIM 88
        JMP   LOOP2
*LOOP1  STAX  7679
        DEX
        BNE   LOOP1
        JMP   END
*LOOP2  STAX  7767
        DEX
        BNE   LOOP2
        LDAIM 83
        LDXIM 88
        JMP   LOOP1
*END    RTS
        END
```

Although this program jumps about somewhat, it is still relatively easy to follow. It starts by initialising A and X then jumps to loop 2 and on completion jumps back

to loop 1 and from there to the END. Note that a *END will not indicate the end of assembly, the "*END" is a label whereas 'END' on its own is the pseudo-code that terminates the assembly process. Once this process is complete the program will reside in memory in exactly the same format as any other program that has been entered. To check this, list the program - from 828 - and the following should appear:

PROGRAM 5.2 (In disassembled assembly language)

	In Assembly Language	In disassembled assembly Language
	LDAIM 90	LDAIM 90
	LDXIM 88	LDXIM 88
	JMP LOOP2	JMP 844
*LOOP1	STAX 7679	STAX 7679
	DEX	DEX
	BNE LOOP1	BNE 250
	JMP END	JMP 857
*LOOP2	STAX 7767	STAX 7767
	DEX	DEX
	BNE LOOP2	BNE 250
	LDAIM 83	LDAIM 83
	LDXIM 88	LDXIM 88
	JMP LOOP1	JMP 835
*END	RTS	RTS
	END	

Program 5.2 can exist in a variety of forms, three of which are readily available. In its original form it was written in assembly language with labels and this was converted by the assembler into machine code and stored in memory in this form. When the assembler is then asked to list this program it reads the machine-code frommemory and changes this back into assembly language. Were this process to be carried out immediately after assembly then it would be possible to re-label the label points by editing the assembler but as currently written this is not so, in common with other assemblers. However, once the BASIC program has been re-run, the variables, i.e. the LABELS and LABEL REFERENCES will have been lost. Re-creating the assembly program from machine code is known as DISASSEMBLY i.e. the 'list' command could be re-titled as 'disassemble' and this process cannot re-create the labels.

When run, this program will print four rows of hearts and four of diamonds in the top eight screen rows.

Exercise 5.1

Add a further loop - loop 3 - after loop 2 in program 5.2. Re-write the program to run loop 3 first followed by loop 1 and then loop 2. Loop 3 should put three rows of asterisks on the screen below the diamonds.

A possible answer is given on page 9.10.

MEMORY LABELS

In addition to labelling instructions, the assembler also allows memory locations to be given labels. Once again, the assembler needs to be told what to expect and the presence of a memory label is indicated by an "@" at the beginning of an instruction. It is then followed immediately by the name assigned to that location and then, after a space by the location itself, in decimal. Thus the instruction:

@LSB 900

informs the assembler that memory location 900 may, in the rest of the program, be referred to as "LSB".

Program 5.3 illustrates the use of memory labels in double precision addition, it adds together two 16 bit numbers;

Number 1 = 2760_{10}

Number 2 = 948_{10}

made up of LSB1 and MSB1 and LSB2 and MSB2, the answers are stored in ANSLSB1 and ANSMSB2.

PROGRAM 5.3

```
@LSB1 900          Define memory location for LSB1.
@MSB1 901          Define memory location for MSB1.
@LSB2 902          Define memory location for LSB2.
@MSB2 903          Define memory location for MSB2.
@ANSLSB 904        Define memory location for ANS LSB.
@ANSMSB 905        Define memory location for ANS MSB.
LDAIM   10     }
STA     MSB1   }   Store MSB1 in memory.
LDAIM   200    }
STA     LSB1   }   Store LSB1 in memory.
LDAIM   3      }
STA     MSB2   }   Store MSB2 in memory.
LDAIM   180    }
STA     LSB2   }   Store LSB2 in memory.
CLC                Clear carry prior to addition.
ADC     LSB1   }   Add LSB's together, store answer
STA     ANSLSB }   in ANSLSB and print on screen.
STA     7681   }
LDA     MSB1   }   Load MSB1 and add with carry to
ADC     MSB2   }   MSB2, store answer in ANSMSB and
STA     ANSMSB }   print on screen
STA     7680   }
RTS
END
```

When run, program 5.3 will display an N (14) in 7680 and a ■ (124) in 7681.

As promised (!) in chapter 4, a listing is given below of an eight bit binary multiplication using labels and memory labels. In order to illustrate the carry operation the first few loops through this program are illustrated on fig. 5.1. The numbers 255 were chosen as multiplier and multiplicand as their binary pattern of eight ones is easy to follow. In addition they give an early carry although any multiplier over 128_{10} would have given the carry when ASLed.

In a program that uses more label references the relevant arrays will need to be declared in DIM statements. These are:

K$(X)	names of label references,
K%(X)	address of label reference,
K1%(X)	number of bytes in operand of relevant instruction.

Thus, to use 20 labels and 20 macros line 50 of the program should be entered as below:-

```
50 DIM K$(20),K%(20),K1%(20),F$(20),F(20)
```

Abbreviations used in this program are:

MPR	=	Multiplier = 255
MPD	=	Multiplicand = 255
TEMP	=	Temporary memory location for storing the carry bit.
RESLSB	=	Result - Least significant bit.
RESMSB	=	Result - Most significant bit.
ALGO	=	Start of multiplication algorithm.
NOCARRY	=	Jump-to point if no carry arises.

PROGRAM 5.4

```
        @MPR    900     }  Define memory location for Multiplier,
        LDAIM   255     }  load and store multiplier.
        STA     MPR     }
        @MPD    904     }  Define memory location for
        LDAIM   255     }  multiplicand, load and store
        STA     MPD     }  multiplicand.
        @TEMP   902     }  Define memory locations for temporary
        @RESLSB 906     }  store, and LSB/MSB of result.
        @RESMSB 907     }
        LDAIM   0       }  Initialise (by loading in zero)
        STA     TEMP    }  the temporary and result stores.
        STA     RESLSB  }
        STA     RESMSB  }
        LDYIM   8          Set loop counter to 8
*ALGO   LSR     MPR     }  Check if right-most character of
        BCC     NOCARRY }  multiplier = 0 branch if so
        LDA     RESLSB  }
        CLC             }  calculate current partial product
        ADC     MPD     }  and add in to current partial sum.
        STA     RESLSB  }
        LDA     RESMSB  }  Add current carry into MSB sum.
        ADC     TEMP    }
        STA     RESMSB  }
*NOCARRY ASL    MPD     }
        ROL     TEMP    }  Current Partial = 0, set up
        DEY             }  next loop
        BNE     ALGO    }
        LDA     RESLSB  }
        STA     7911    }  Display result of
        LDA     RESMSB  }  LSB and MSB.
        STA     7910    }
        RTS
```

When run, the program will print a ■ in 7910 and an A in 7911 i.e. an answer of 254, 1 or FE 01_{16} = 65025_{10}.

Fig. 5.1, p. 5.10 steps through the first few stages of this program, once all the registers are set up. The contents of each address are shown only when they change, thus when nothing is shown they can be assumed to be as in the previous entry.

LABEL	INSTRUCTION	Y-REGISTER	ACCUMULATOR	MPR MULTIPLIER	C	TEMP	MPD	RESLSB	RESMSB
		0000 1000	0000 0000	1111 1111		0000 0000	1111 1111	0000 0000	0000 0000
ALGO	LSR MPR			0111 1111	1				
	BCC NOCARRY								
	LDA RESLSB		0000 0000						
	CLC				0				
	ADC MPD		1111 1111						
	STA RESLSB							1111 1111	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0000						
	STA RESMSB								0000 0000
NOCARRY	ASL MPD				1		1111 1110		
	ROL TEMP				0	0000 0001			
	DEY	0000 0111							
	BNE ALGO								
ALGO	LSR MPR			0011 1111	1				
	BCC NOCARRY								
	LDA RESLSB		1111 1111						
	CLC				0				
	ADC MPD		0000 0001		1				
	STA RESLSB							0000 0001	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0001						
	STA RESMSB								0000 0001
NOCARRY	ASL MPD						1111 1100		
	ROL TEMP				0	0000 0011			
	DEY	0000 0110							

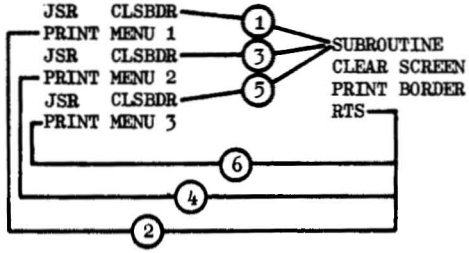
etc.

Fig. 5.1

MACRO INSTRUCTIONS

One further feature of your assembler is that it enables you to use MACRO instructions. These are blocks of code that you wish to repeat and are thus given a label. When you wish to insert these into your program you simply need to type in the label and this automatically inserts the whole macro. Thus a macro is very similar to a subroutine except that the assembler writes it in every time it's called rather than using JSR instructions. As an example, take a short routine that clears the screen and puts in a border called CLSBDR. Assuming that the program has a hierarchical menu structure and calls this routine 3 times, it could be written in as (i) below Fig. 5.2 in which the routine is called as a subroutine or in (ii) in which the routine is written in 3 times as a macro.

(i)



(ii)

```
MACRO CLSBDR
MENU 1
MACRO CLSBDR
MENU 2
MACRO CLSBDR
MENU 3
```

Fig. 5.2

The above figure is somewhat unfair on subroutines in that they are not so messy as the picture suggests, the 6502 doing muchof the organisation. Where macros score is in producing a program that reads more logically and is easier to follow. They take up more space than a subroutine which

is only written once but, as they don't jump about, run more quickly. Just whether a macro or a subroutine is more preferable in any particular case is left to the individual programmer.

To identify a macro on this particular assembler it is preceded by a "+" sign as the macro is later added into the program. As with other features, the + sign must immediately precede the macro's name which may be any length and this should be the only entry on that particular line. The first occurrence of the macro is then typed in and, on completion, the macro identifier, a "+" is typed followed immediately by "END". To add a further macro 1 into the program the line +MACRO1 is added. See Fig. 5.3.

```
+MACRO1
 LDAIM 90
 LDXIM 40
 STAX  7680
 DEX
 BNE   250
+END
+MACRO1
 RTS
 END
```

Fig. 5.3

When using labels, label references and macros on a 6.5K machine, one must beware of space limitations. This results from the storage of their names and locations in BASIC prior to assembly. When run without DIM statements, the program will thus accept up to 11 labels or memory labels, 11 label references and 11 macros. Macros which contain labels or references create new labels and references and these must also be taken into account. Thus, if a Macro with 1 labels and 2 references is repeated, an extra 2 labels and references + one macro will be created.

If large numbers of labels, references or macros are required two problems must be considered - the easiest one first:

1. Dimensioning the variables.
   a) Labels and memory labels are stored in F$(X), F(X) i.e. to enable 12 labels to be used line 100 should contain: DIM F$(12), F(12).
   b) Label references are stored in K$(X), K%(X) and K1%(X) i.e. to enable 12 label references to be used line 100 should contain:
   DIM K$(12), K%(12), K1%(12).
   c) Macros are stored in M$(X), M%(X) and M1%(X) i.e. to enable 12 macros to be used,line 100

should contain:
DIM M$(12),M%(12), M1%(12).

2. Ensuring that enough space exists for the variables.
   a) Each ocurrence of the label suite of variables F$(X) and F(X) requires 8 bytes.
   b) Each ocurrence of the label reference suite K$(X), K%(X) and K1%(X) requires 7 bytes.
   c) Each ocurrence of the macro suite M$(X), M%(X) and M1%(X) requires 7 bytes.

A reasonable rule of thumb is that each new label, reference or macro requires about 10 more bytes.

CHAPTER 6

Without the Assembler .....!

So far all the machine code programs considered have been entered via the assembler. However, as discussed in the last chapter, all the assembler does is to make it easy to POKE data into memory. The assembler is, of course, not the only way of putting in a machine code program into memory, one other way is to POKE it in directly. Program 6.1 is expressed in this direct POKE form.

PROGRAM 6.1

```
POKE 828,162
POKE 829,0
POKE 830,169
POKE 831,90
POKE 832,157
POKE 833,0
POKE 834,30
POKE 835,232
POKE 836,208
POKE 837,250
POKE 838,96
```

One easy way to put this in is to clear the screen first and type in POKE 828,162 from 7680 onwards. After each <return> , a HOME will put the cursor back on the P in POKE. When the program is in it may be run by the SYS 828 command and it will put 256 diamonds onto the screen. Can you imagine the program that these POKES have created? Fortunately, its not necessary to imagine it, as the assembler will disassemble the code from memory for you and display it. First RUN the program, select '2' at the MENU and then type in the address 828.

The assembler will then reveal the source program for 6.1a which was:

PROGRAM 6.1a (In Assembly language)

```
LDXIM 0
LDAIM 90
STAX  7680
INX
BNE   250
RTS
```

As well as entering programs without the assembler it is also possible to run them directly or from a BASIC program. For instance, to run the program that currently starts at memory location 828 it is only necessary to put an 828 into the program counter. If program 6.1 is still in memory this can be used for a demonstration of this - if it is not still in memory type it in. Next exit from the assembler and get back to BASIC with the READY and cursor.

Now type in SYS828 and the program will run, putting in the diamonds.

Now run this from BASIC. Put in this following program:-

PROGRAM 6.2

```
20000 PRINT "◆"          Clear screen.
20010 SYS 828            Direct PC to 828
20020 PRINT "ALL OVER"
```

Now type in "RUN 20000" <return> and the BASIC program should run in the following way:-

Line 20000	The BASIC program clears the screen.
Line 20010	Hands control to the machine code program at 828 Control is handed backto BASIC when the final RTS is encountered.
Line 20020	BASIC prints out the message "ALL OVER".

Running a program directly is relatively easy but the direct mode of entry of a program is obviously a tedious way of entering a long program so a further more easily entered option is offered by the storing of the machine code in DATA statements. Program 6.3 shows a machine code program loaded in via BASIC.

PROGRAM 6.3

```
1 FOR X = 0 TO 15
2 READ A
3 POKE (828+X),A
4 NEXT X:RESTORE
5 DATA 162,0,169,42,157,0, 30 ,169
6 DATA 90,157,0, 31 ,232,208,243,96
7 END
```

This is run as for a normal BASIC program with a RUN command. Once run,the DATA will have been loaded into

memory and lines 1 to 7 must be looped out and the statements REMed out.

```
1.  GO TO 10:FOR X = 0 TO 15

5.  REM DATA 162,0...
6.  REM DATA 90,157,0
```

Once this is done, the main assembler can be run and the program RUN from the MENU - it's located at 828. Of course, once the data is loaded into memory, the program can also be run by a SYS 828 command.

The VIC ASSEMBLER 2 offers a further method of entering machine code via:

THE MONITOR : Option 4

This feature offers a ready facility for examining and modifying memory. To enter MONITOR type "4" at the MENU. The computer will then display:-

```
VIC MONITOR
?
```

Once in MONITOR single letter commands inform the monitor what facility is requested. To investigate this type in:-

? M 033C 034B <return>

Take care when doing this to put in all the spaces, otherwise the MONITOR will be confused. This command should, assuming program 6.1 is still loaded, result in the display below:-

```
  VIC MONITOR
? M 033C 034B
  033C A2 00 A9 2A 9D
  0341 00 1E A9 5A 9D
  0346 00 1F E8 D0 F3
  034B 60 31 36 30 20
?
```

Fig. 6.1

The M command requests the monitor to list the block of memory defined i.e. in this case from 033C to 034B. The block listed above is, in fact,that program loaded by PROGRAM 6.1 and the listing tells that 033C contains A2, 033D contains 00 and so on to 034B which is loaded with 60. One of the most useful facilities offered by the monitor, in conjunction with the M command, is the ability to edit machine code programs as if they were BASIC Programs. For instance, while program 6.1 is loaded, go into monitor and list 033C to 034B.

Type

.M 033C 034B <return>

At this stage the cursor should be flashing immediately below the 0 of 034B beneath the display. Now move the cursor over the first '2' of 2A at 033F, replace this with a '5'. The cursor will move over the 'A' and this should be replaced by a '3' after changing these, enter it into memory by pressing return.

Next repeat this to replace the 5A in 0344 with a 58, and again press return to enter this into memory.

The display should now appear as in fig. 6.2:-

```
   VIC MONITOR
? M 033C 034B
  033C A2 00 A9 53 9D
  0341 00 1E A9 58 9D
  0346 00 1F E8 D0 F3
  034B 60 31 36 30 20
?
```

Fig. 6.2

In order to exit from the monitor the command is X followed by return. Try this and the machine will return to the MENU.

The Assembler program may then be RUN and on LISTing the LDAIM 42 and LDAIM 90 instructions will be seen to have been changed to LDAIM 83 and LDAIM 88. On running the asterisk will be replaced by a heart and the diamonds will be replaced by clubs.

Not only can programs be edited in monitor, they can also be entered. Try this with the following exercise, using the monitor to enter the following code:

i) Enter Monitor.
ii) List memory contents Type M 033C 034E
return
iii) Replace contents with:

PROGRAM 6.5

```
   VIC MONITOR
? M 033C 034E
  033C A0 2C A9 4D 99
  0341 FF 1D 99 57 1E
  0346 A9 4E 99 2B 1E
  034B 88 D0 F0 60 88
? X <return>
```

Note that the final "88" in this program is really irrelevant as it is after the 60 or RTS. Thus any value could be put into this location.

The machine code program is now entered into memory and may be RUN either from the Assembler or by SYS 828. It can also be dis-assembled using the assembler to reveal the Assembly language version.

A very useful built-in feature of the monitor is its ability to move whole blocks of code from one location to another. This is very simply achieved by changing the address displayed on the monitor listing and pressing return. For instance, the program in 033C onwards can be copied to start at 900_{10} (0384_{16}) or by the following procedure:

i) Enter MONITOR
ii) List 033C to 034E by
iii) ? M 033C 034E <return>
iv) Re-position cursor over 033C and replace this with 0384 and press <return>
v) Repeat for other addresses:

0341 goes to 0389
0346 goes to 038E
034B goes to 0393

The procedure has made a copy of the 033C program in 0384 onwards. It can be run from the assembler or by a SYS 900.

The program can also be run by a monitor command "G" which is short for "GO!". It runs a machine-code program direct from memory. Naturally the VIC has to be told where the program starts and so the total command reads:

? G 0384 <return>

Protecting Machine-code in Memory

Two monitor commands enable machine-code programs to be SAVEd onto tape and LOADed back from this. However, one slight problem arises in doing this as all machine-code programs so far have been stored in the cassette buffer and this is used when LOADing and SAVEing programs.

In order to protect programs from being over-written, they must be stored elsewhere in memory.

On the VIC, BASIC programs are stored either from 4096 or, with the 3K extension in use, from 1024. However, strings generated by a program are written from the top of memory downwards. The VIC knows where to start storing this information, as when the machine is turned on it checks what the highest available memory locations are and stores this information in the zero-page locations 55 and 56. If a machine-code program is stored at the top of memory then it will be written over when any BASIC program is run - such as the assembler - when this generates strings. However, before it stores the strings, the BASIC program checks in memory locations 55 and 56 to find where to start. This allows one to protect the top part of memory by loading an address into 55 and 56 that leaves some clear space.

With the basic VIC or with the 3K additional memory, the top of available memory is at 7679 - one byte below the screen. Thus, at switch-on this address $1D\ FF_{16}$ is to be found in 55 - LSB and 56 - MSB. In order to protect 100 bytes, this address must be lowered to 7579_{10} ($1D9B_{16}$) by means of the program

POKE $55,155(9B_{16})$

POKE $56,29(1D_{16})$

This can be achieved by modifying line 10 of the assembler to read:

```
10  POKE 36879,41:POKE 55,154:POKE 56,31
```

Once the way is clear, the 033C program can be copied to 1D9B onwards and listed. (Lines start with 1D9B, 1DA0, 1DA5, 1DAA). At these high altitudes, the 22 columns of the VIC are not quite enough but it yields a legible display.

Once stored at height the LOAD and SAVE routines can be used.

SAVE

For a program to be SAVEd, the monitor needs to know where it starts and where it ends and, as with other programs, it needs a name. Thus the full command becomes

? S"NAME".START ADD.END ADD

Translating this to save the check program it becomes

? S"CHECK".1D9B.1DAD <return>

After the return, the VIC will respond with a "PRESS RECORD AND PLAY" and when this has been done, an "OK". Finally, when the recording is complete, the VIC will respond with a "?" and the flashing cursor of the MONITOR as it awaits further instructions.

LOAD

Provision is made on the monitor for re-loading a machine code program from tape and storing this back into the location from which it was originally saved. It does this by saving the start and end addresses onto tape so that it knows where to load them back. As the tape contains this address, the LOAD command need only define the name of the program, i.e.

L"CHECK" <return>

Following the return, the VIC will respond with a "PRESS PLAY ON TAPE" and an "OK" when pressed. Finally, when the program is loaded the VIC will respond with a "?" and the flashing cursor of the MONITOR as it awaits further instructions.

Direct Assembly from MONITOR See page 6.13

COLOUR AND SOUND

One of the major features of your VIC computer is its ability to produce colour displays and the ease with which sound can be added to programs. This facility is just as readily used from machine-code as from BASIC.

Controlling the Colour

The programs developed so far in this book have utilised the screen/border colour combination set in line 10 with a POKE 36879,41 i.e. a white border and red screen. Output from the assembler is then set to white by the statement PRINT " E " in line 20.

Program 6.6 shows how this screen/border combination can be demonstrated using a short machine code program.

PROGRAM 6.6

```
LDXIM 255
STX 36879
DEX
BNE 250
RTS
END
```

Unfortunately, when this is run as it stands there is little to see as the whole program runs through in about 2600 cycles or 1300 microseconds. In order to enable the changes to be visible, a delay must be inserted to hold each change on the screen long enough for the eye and brain to perceive it. This is demonstrated in program 6.7 where three nested loops produce a delay of 201 200 cycles or 100 600 microseconds, i.e. 1/10 second.

PROGRAM 6.7

```
SEC
LDAIM 255
LDYIM 200
LDXIM 250
DEX
BNE 253
DEY
BNE 248
STA 36879
SBCIM 1
BNE 239
RTS
```

Individual characters on the screen can also be controlled by use of the screen colour codes. Thus, if memory location 38400 contains a "2", then the top left memory location, i.e. 7680 will print out on the screen in red. The complete set is given below in fig. 6.3

Code	Colour
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow

Fig. 6.3

Program 6.8 shows how the colour of blocks of screen can be defined by means of the screen colour codes.

PROGRAM 6.8

```
LDXIM 220
LDAIM 7
STAX 38399
LDAIM 160
STAX 7679
DEX
BNE 243
RTS
```

This process is taken a stage further in program 6.9 where the eight different colours are cycled through, with a delay between each.

PROGRAM 6.9

```
LDAIM 0          } Set up first colour and save
PHA              } on stack.
LDXIM 220          Set counter for 10 rows.
LDAIM 160          Set character for printing.
STAX 7679        }
PLA              }
STAX 38399       } Print block onto screen.
PHA              }
DEX              }
BNE 243          }
TXA              }
PHA              }
LDYIM 200        }
LDXIM 250        }
DEX              } Delay.
BNE 253          }
DEY              }
BNE 248          }
PLA              }
TAX              }
PLA              }
ADCIM 1          }
CMPIM 8          } Move to next colour and check
PHA              } if all done.
BNE 219          }
PLA              }
RTS              }
```

Sound in Machine-Code Programs

Another of the VIC's special features is its ability to output sound through the television set. By doing this the VIC can make the most of the television's large speaker and, hence, give a good quality sound. As with all other outputs, the 6502A treats these as memory addresses and so any value loaded into addresses 36874 to 36877 emerges as sound through the television. The volume of this sound is also controlled by an address, 36878, and the value in this, from 0 to 15 sets the level of sound. Thus, if a loop is set up in which the value in 36878 is incremented the volume will increase progressively. This is shown in the segment from program 6.10 shown below:-

```
     LDAIM 3       )
*VOL STA 36878     )  Set up volume to 3.

     CMPIM 15      )  Check if volume at 15, if so branch
     BNE CODA      )  to CODA .

     CLC           )
     ADCIM 1       )  Add one to volume.

     JMP VOL          Jump back to VOL.
     CODA.....
```

The value stored in locations 36874 to 36877 sets the frequency of the note, the higher the value stored, the higher the note. These are listed in the VIC manual and, for convenience are given in fig. 6.4.

NOTE	C	C#	D	D#	E	F	F#	G	G#	A	A#
VALUE	135	143	147	151	159	163	167	175	179	183	187
NOTE	B	C	C#	D	D#	E	F	F#	G	G#	A
VALUE	191	195	199	201	203	207	209	212	215	217	219
NOTE	A#	B	C	C#	D	D#	E	F	F#	G	G#
VALUE	221	223	225	227	228	229	231	232	233	235	236
NOTE	A	A#	B	C	C#						
VALUE	237	238	239	240	241						

Fig. 6.4

Program 6.10 demonstrates this process where an A is loaded into 36874 and outputted on channel 1. It is then held there for a short time by means of a delay while the volume is reduced from(15 to 0). Following this up the A is changed to an A when the 183 in 36874 is increased to 187 by the ADCIM 4

(not forgetting the CLC first). Again the volume is reduced and the whole process repeated for the notes B and C.

In this program the delay incorporated utilises the X and Y registers which are also used elsewhere. Thus, their contents will be destroyed during the delay routine and, therefore, if needed again, must first be saved on the stack. In this particular case A is not modified by the delay routine but it too is saved on the stack to illustrate the general procedure. Retrieval from the stack must, of course, take account of the LIFO (last in-first out) operation of the stack.

PROGRAM 6.10

```
LDXIM 4          Set up loop counter.
LDAIM 183   }
STA 36874   }    Send first note to channel 1.
LDYIM 15    }
STY 36878   }    Set volume.
PHA         |
TXA         |
PHA         }    Save X, Y and A on stack.
TYA         |
PHA         |
LDYIM 250   |
LDXIM 250   |
DEX         }    Delay
BNE 253     |
DEY         |
BNE 248     |
PLA         |
TAY         |
PLA         }    Recover A, Y and X from stack.
TAX         |
PLA         |
DEY         }
BNE 230     }    Reset volume.
CLC         |
ADCIM 4     }    Change note and check if all done.
DEX         |
BNE 219     |
LDAIM 0     }
STA 36874   }    Turn off.
RTS
```

In addition to outputting onto channel 1, as in program 6.10, the VIC can also output sound onto three other channels. Two of these channels are sound as channel one, while the other is a noise output. Program 6.12 demonstrates this by outputting a three-note chord, holding this for a time and then replacing it with another chord.

As with the earlier program, a delay is required to sustain the sound, this being stored in memory from 910 onwards. In this delay a different technique is used to extend its duration as a memory location is utilised as a register. Thus, prior to entering the delay a value is set in Y to define the delay length. Within the delay, this is stored in 940 and then decremented as if it were a register - see below.

PROGRAM 6.11

```
START ADD? 910
STY 940
LDYIM 250
LDXIM 250
DEX
BNE 253
DEY
BNE 248
DEC 940
BNE 243
RTS
```

DELAY SUBROUTINE

This program is called by program 6.12 which sends the notes G, B and D to channels 1, 2 and 3 and then loads the delay parameter 20 into Y. Following the delay it changes the chord to C, E, G and then increases the volume progressively until it finally switches off.

PROGRAM 6.12

```
LDAIM 2
STA 36878     } Set volume to 2.
LDAIM 175
STA 36874     } Send G to channel 1.
LDAIM 191
STA 36875     } Send B to channel 2.
LDAIM 201
STA 36876     } Send D to channel 3.
LDYIM 20
JSR 910       } Set delay parameter and jump to delay.
LDAIM 195
STA 36874     } Send C to channel 1.
```

cont'd.

PROGRAM 6.12 cont'd.

```
LDAIM 207   }
STA 36875   }   Send E to channel 2.
LDAIM 215   }
STA 36876   }   Send G to channel 3.
LDYIM 10    }   Set delay parameter and
JSR 910     }   jump to delay.
LDAIM 3         Set volume.
STA 36878       Output volume.
CMPIM 15        Check current volume.
BEQ 11          Branch to coda if all done.
LDYIM 2     }   Set delay parameter and
JSR 910     }   jump to delay.
CLC         }
ADCIM 1     }   Increase volume and
JMP 875     }   jump back.
LDAIM 0     }
STA 36874   }
STA 36875   }   Turn off.
STA 36876   }
RTS             Return to BASIC.
NOP         }
NOP         }
NOP         }
NOP         }
NOP         }   Fill in memory up to 910.
NOP         }
NOP         }
NOP         }
```

Direct Assembly from MONITOR

So as to avoid the necessity of exiting into MONITOR prior to entering a program in assembly-language, provision is made on the MONITOR for direct entry of 6502 mnemonics. To enter this mode simply type in an open square-brackets followed directly by the decimal start address, i.e.

```
?[ 828   <return>
```

Having done this an assembly-language program may be entered directly. If the program entered has had all branches and jumps calculated, i.e. for example BNE 253 rather than BNE START then to return to MONITOR the square-bracket is simply closed i.e.

```
?] <return>
```

However, if memory labels, labels or macros have been used, then exiting by means of a square bracket will lose the addresses. To enable the assembler to calculate the addresses

of branches etc., the entry must be terminated by an "END". After this entry the program returns to the main MENU i.e.:

```
? END <return>
```

Programs 6.13 and 6.14 below demonstrate the use of these commands.

PROGRAM 6.13

```
   VIC MONITOR
?[ 828
? LDAIM 90
? STA 7680
? RTS
?] ————————→ (remains in MONITOR)
```

PROGRAM 6.14

```
  VIC MONITOR
?[ 828
? *START LDXIM 100
? LDAIM 90
? STAX 7680
? DEX
? BNE START
? RTS
? END ————————→ (Returns to main MENU)
```

CHAPTER 7

Using the VIC's Built-in Sub-routines

As the VIC itself uses the 6502 chip, it has stored inside it, on ROM, subroutines that control the 6502 and enable the chip to deal with BASIC commands that are put into programs. As well as this, some of the RAM is given up for the storage of transient data such as, for instance, the clock that ticks on once the computer is turned on. Some of these functions, like the clock, take up only one or two bytes while others are subroutines that are entered, do their job and are then returned from. The most difficult aspect in using these functions is to know where they obtain their data from and where they deposit the data that they have generated.

First let's have a look at the contents of the accumulator using a VIC subroutine. We've already examined this by using the STA command and pointing it into a screen location, e.g. STA 7680. However, the subroutine at 65490_{10} ($FFD2_{16}$) does this outputting of the accumulator, starting from the next cursor position. This is illustrated in:

PROGRAM 7.1

```
LDAIM 42        Load accumulator with '42'
                (an asterisk).
STA 7702        Print out accumulator contents
                in 7702.
JSR   65490     Jump to VIC subroutine at 65490.
RTS
END
```

When run, this displays asterisks in the top left-hand corner (7680) and towards the centre of the screen (7702).

Unfortunately, though outputting is nowhere near as easy as this program suggests! To check this try replacing the LDAIM 42 with an LDAIM 1. You can do this with a POKE 829,1. Then when running the program an empty space is obtained at 7680 while the expected 'A' is seen at 7702. In fact, the VIC uses the ASCII character set when using subroutine 65490. This set is listed in Table 4 (Appendix 2) page A2.25.

The major advantage of this subroutine is that it locates the cursor automatically to 7680 and then increments this. As well as going straight to 'home' position on the screen, the cursor can be located by using the memory locations 209 - 210 to point to the beginning of the line and location 211 to point to the position along that line.

The line number is defined by putting the beginning memory location of that line in 209 and 210 Least-significant byte first i.e. for the top line the memory location is 7680_{10} i.e. $1E00_{16}$. That of the 10th line is 7878_{10} or $1EC6_{16}$.

Thus, for the beginning of line 10 the LSB = $C6_{16}$

and the MSB = $1E_{16}$

In order to use this to move the cursor down, these must be loaded into 209_{10} (LSB) and 210_{10} (MSB) to utilise this subroutine, as in:

PROGRAM 7.2

```
LDAIM $00 C6
STAZ 209
LDAIM $00 1E
STAZ 210
NOP
NOP
NOP
NOP
LDAIM 42
STA 7900
JSR 65490
RTS
END
```

On running, this can be seen to have put the first asterisk down to the beginning of line 10. To move the cursor along, location 211 is loaded with the number of locations along that the cursor is to be moved. Thus program 7.2 can be modified by program 7.2a.

PROGRAM 7.2a

```
836 LDAIM 10
838 STAZ 211
    END
```

To yield program 7.3:

PROGRAM 7.3

```
LDAIM $00 C6
STAZ  209
LDAIM $00 1E
STAZ  210
LDAIM 10
STAZ 211
LDAIM 42
STA  7900
JSR  65490
RTS
END
```

When run, this program moves the cursor down 10 and across 10 and prints the asterisk at this location.

Figure 7.1 below lists all the necessary co-ordinates for positioning the cursor.

Beginning of Line No.	Decimal	HEX	
		LSB	MSB
1	7680	00	1E
2	7702	16	1E
3	7724	2C	1E
4	7746	42	1E
5	7768	58	1E
6	7790	6E	1E
7	7812	84	1E
8	7834	9A	1E
9	7856	B0	1E
10	7878	C6	1E
11	7900	DC	1E
12	7922	F2	1E
13	7944	08	1F
14	7966	1E	1F
15	7988	34	1F
16	8010	4A	1F
17	8032	60	1F
18	8054	76	1F
19	8076	8C	1F
20	8098	A2	1F
21	8120	B8	1F
22	8142	CE	1F
23	8164	E4	1F

Fig. 7.1

Many BASIC programs use the GET command, which accepts a single byte input into the cassette buffer and the sub-routine that carries out this operation can be utilised. It is located at $FFE4_{16}$ or 65508_{10} and when run loops continuously until an input enters the cassette buffer and this it transfers to the accumulator. In order to see that this input has taken place, it is necessary to copy it from the accumulator onto the screen. Program 7.4 shows how this may be done.

PROGRAM 7.4

```
LDAIM 0        Set accumulator clear.
JSR   $FF E4   Jump to GET subroutine.
BEQ   251      Branch back if no input.
STA   7900     Display accumulator on screen.
RTS
```

When run, this program sets up the loop:

```
┌─►JSR  $FF E4
└──BEQ  251
```

which waits for an input. Once the input occurs the program runs through the BEQ command and executes the remainder of the program i.e. displaying the GET character. However, as with the earlier sub-routine used, this one works in ASCII rather than VIC screen code. For instance if a 'Z' is input, the screen will respond with a VIC screen code 90, a diamond. In the case of numbers, the VIC and ASCII codes co-incide so a numeric input will result in the same character being displayed on the screen.

The INPUT subroutine at $FFCF_{16}$ or 65487_{10} uses VIC screen code and is thus somewhat more convenient. However, as with INPUT in BASIC, the machine needs to be told how long the data is in some way. With a normal INPUT this is done by pressing RETURN (CHR$(13)) and this feature is also built into the machine code subroutine at 65487_{10}. This produces the shortest program yet in 7.5!

PROGRAM 7.5

```
JSR 65487
RTS
```

Once typed in, the program leaves the machine as if a BASIC INPUT command has just been executed and awaits the input followed by a return. Programmed into the subroutine is the need for a return which, of course, hands over control to the computer until it detects this return.

The output subroutine can be combined with the GET subroutine to design your own INPUT routine. In this case the GET routine is used to enter one character and then checked either for the number of characters already entered or for the inputting of a terminating character such as a return.

This facility enables one to design all manner of special INPUT routines that are terminated by a chosen key. Such a facility could detect a character in the actual INPUT itself and give a warning if it is an invalid character.

Program 7.6 shows an arrangement with a check built in to look for a 13 input (CHR$(13)).

PROGRAM 7.6

```
LDXIM 13        )
STX 900         ) Load a '13' into 900.

JSR $FF E4      ) Use GET subroutine and
BEQ 251         ) check whether character inputted.

JSR $FF D2        Output A onto screen.

CMP 900         ) Look for <return> if not present
BNE 243         ) branch back.
RTS
```

This program simulates an INPUT routine that is terminated by a RETURN. The terminator may be varied by loading a different character into 900. Try this for:

Exercise 7.1

Modify program 7.6 to accept an input that is terminated by a space, using only POKE commands.

Answer on page 9.10.

One of the major problems encountered when using built-in subroutines is that they too use the 6502. That means that they put things into A, X and Y and they also modify the PSW - the Z, N and other flags. Hence when returning from any JSR it's not reasonable to assume that everything is just as it was left before the JSR.

This of course, applies when jumping from machine code into BASIC, as "BASIC" itself is only a collection of machine code - and it's full of JSR's!

As an example of this look at the following program that is designed to input a four character string from the keyboard. Firstly it sets a loop counter in X at 4, then it uses the GET and PRINT A subroutines. On return from these it decrements X and checks for Z flag set - all very straightforward!

PROGRAM 7.8

```
LDXIM 4          Set up loop counter.
JSR $FF E4   )
BEQ 251      )   Get sub-routine.
JSR $FF D2       Display A sub-routine.
DEX
BNE 245          Check for end of loop.
RTS
```

However, when run, this program RTS's after only one character has been input! This suggests that one of the subroutines is using the X register. To verify this the program can be made to print out the X register at various stages just to check. This is done in program 7.9 where the contents of the X register are examined immediately after returning from the subroutine.

PROGRAM 7.9

```
LDXIM 4
JSR $FF E4
BEQ 251
STX 7900
JSR $FF D2
STX 7902
DEX
BNE 239
RTS
```

When run, this program prints the one character input as before but also displays two A's at 7900 and 7902 These A's (POKE 1's) indicate that the JSR $FF E4 (JSR 65508) instruction modified the X register from its set value to 1. Hence, when decremented, the program left the loop on the first pass.

To overcome this problem, the X register value must be stored somewhere prior to entering the sub-routine and then retrieved prior to being decremented. Program 7.10 shows this process, where X is temporarily stored in 900 during the subroutine.

PROGRAM 7.10

```
        @LOOPCONST  900
        LDXIM 4
*STORE  STX LOOPCONST
 *GET   JSR   $FF E4
        BEQ   GET
        JSR   $FF D2
        LDX LOOPCONST
        DEX
        BNE STORE
        RTS
```

Assembled Version

```
LDXIM 4
STX   900
JSR   65508
BEQ   251
JSR   65490
LDX   900
DEX
BNE   239
RTS
```

When run, this program allows four entries from the keyboard, displays these and then passes out of the loop.

As a technique, the use of memory storage in this way works but calls for some care in storing the data safely and retrieving it when needed. Fortunately the 6502 has a device for doing this operation automatically. It is

THE STACK

The stack (S) is a block of memory, on the VIC machines located from 511_{10} down to 256_{10} (i.e. holding 255 bytes). It is used for the rapid transfer of data and is filled downwards from 511 its next vacant location being recorded by a STACK POINTER. When anything is to be retrieved from the STACK only the last item put in is accessible. The usual analogy is with a stack of plates only the top one being accessible as this was the last one put there. However, the stack is filled DOWNWARDS i.e. from 511 towards 256 so plates are put at the bottom and retrieved from there, antipodean fashion! This mode of filling and emptying the stack is known as last-in, first-out, i.e. the stack is a last-in, first-out(LIFO)store.

One function of the stack is to record addresses during subroutine jumps which it does automatically. When the 6502 sees an instruction such as JSR 50000 it must, first of all record where the next instruction is so that it can find its new location after the subroutine has been executed and then place the "50 000" into the program counter (PC). The process is examined below with the program segment 7.11 (from 7.10)

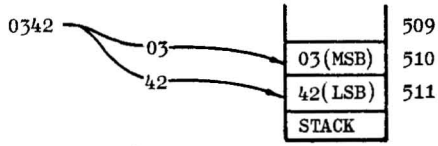
PROGRAM 7.11

```
STEP 1. 828 (033C)  STX 900  (03 84)
STEP 2. 831 (033F)  JSR $FF E4 (65508)
STEP 3. 834 (0342)  BEQ 251
```

INSTRUCTION STX 900

STEP 1.
- i) Calculate address of next instruction, i.e. 831_{10} or $033F_{16}$.
- ii) Put next address into program counter.
- iii) Execute instruction STX 900.
- iv) Retrieve address for next instruction from PC, i.e. $033F_{16}$.

STEP 2.
- v) Fetch next instruction, i.e. at 033F. This is JSR $FF E4.
- vi) Retrieve address for next instruction in program i.e. 0342_{16} and place this on the stack



- vii) Record the next vacant location in the stack pointer, i.e. 509_{10} (SP = 509_{10}).
- viii) Load FF $E4_{16}$ into program counter.
- ix) Jump to sub-routine at FF $E4_{16}$.
- x) Execute subroutine until RTS is encountered.
- xi) Look at stack pointer to find last data. SP = 509, therefore Data stored at 510 and 511.
- xii) Extract data from 510_{10} and 511_{10} (i.e. 0342_{16}) and load this into PC.
- xiii) Jump to 0342_{16}.

STEP 3. xiv) Fetch next instruction and carry on with program.

In the example quoted the first subroutine could well have met further subroutines or nested subroutines and each time a JSR was executed, the return address would have been piled on top. Then as the program returned successively through these subroutines the return addresses would have been stripped off successively to steer the 6502 back to the original point of departure. Fig. 7.2 illustrates this for subroutines nested 3 deep and starting from a program with a JSR $03 84 instruction in $033D_{16}$.

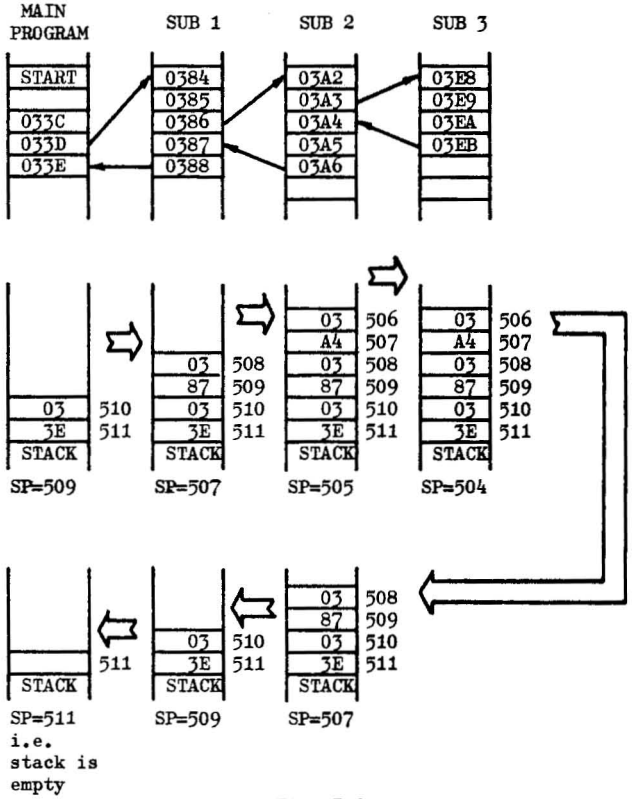


Fig. 7.2

In the above program the instructions at 03EB, 03A6 and 0388 would all be RTS. Thus when the 6502 finds an RTS at 03EB it takes the top address off the stack which is 03A4. Reading downwards at SUB 3 the stack gives

i) address for return to subroutine 2.
ii) address for return to subroutine 1.
iii) address for return to the main program.

Fortunately the operation of the stack in recording addresses when executing subroutines is automatic and so the programmer can allow the 6502 to do the job. However, as was seen when using built-in subroutines the stack does not automatically store register contents but must be programmed to do so. Only two instructions exist for storing registers, neither of these operating on the X and Y registers. These must be handled via the accumulator which is stored using:

PHA	PusH contents of Accumulator onto stack.

The contents may then be retrieved by means of:

PLA	PuLl top of stack into Accumulator

In both cases, the stack pointer is updated appropriately so that it continues to point to the next empty location in the stack.

Using these instructions, Program 7.8 can be re-written to transfer the X register into the stack and retrieving this when needed. The Stack pointer will take care of the order of the data providing that the LIFO structure of the stack is borne in mind and data is entered and retrieved in the right order.

Writing these in yields:

PROGRAM 7.12

```
LDXIM 4          Set up loop constant.
TXA        )
PHA        )     Transfer X into stack.
JSR $FF E4
BEQ 251
JSR $FF D2
PLA        )
TAX        )     Recover X from stack .
DEX        )     decrement and check for end of
BNE 241    )     loop.
RTS
```

When run, the program accepts a four character input and prints this on the screen.

As well as affecting the accumulator, a subroutine is most likely to reset one or more flags in the PSW. On returning to the main program, the reset flags are then certain to upset the course of this. To overcome this problem, the 6502 has built-in provision for saving the PSW (i.e. the condition of all the flags on the stack). This is brought about by the instruction:

PHP	PusH contents of PSW onto stack.

Retrieval of the data is achieved by:

PLP	PuLl top word of stack in PSW.

In program 7.12 the PSW was not saved on the stack as, prior to testing the Z flag with a BNE, the DEX instruction reset this. However, under other circumstances it may have been necessary to preserve the state of the PSW so this should be written into 7.12 as an exercise.

Exercise 7.2

Re-write program 7.12 so as to save the condition of PSW in the stack prior to the subroutines and retrieve this after these.

A possible answer on page 9.11.

When using the stack, the main precaution to take is to check the order of entry and retrieval several times - always LIFO. For instance a possible routine for saving the accumulator, X and Y registers and the PSW is given in Fig. 7.3.

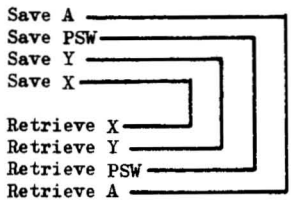


Fig. 7.3

CHAPTER 8

Interrupting the 6502, signed and floating-point numbers

When doing a job that has to be done, no-one likes interruptions until the job's done. The 6502 is just like that too! During a piece of program it has its Accumulator and X and Y registers under control and all the flags set appropriately. Thus, when an interrupt comes,all these registers have to be stored, usually in the stack and after the interrupt they must all be retrieved. An interrupt is, in fact, only a subroutine that comes when IT is ready and not the 6502! This means that the interrupt is generated from outside the 6502's cosy system either from an external device or the keyboard.

The handling of an interrupt has to be prepared for in the program and at certain times the program may well not be able to allow an interruption. If, for instance, another device is sending a stream of data into memory then a HAND-SHAKING procedure is carried out between the two machines. Quite simply, this is an exchange of messages along the line "I am ready to send data, are you ready to receive it?". "Yes". "Here's the data ..... end of data." "Thanks!" If such an interchange is interrupted then the data is likely to become garbled and, hence, worthless. During such periods when no interrupt is allowable, the program can block most interrupts - not all - to allow a particular process to be completed. The instruction that allows this blocking is:

SEI	SEt Interrupt Disable.

Ironically the first action when actually in an interrupt is to set the I (interrupt) flag, by use of SEI, in order to prevent any more interruptions. On completion of the interrupt procedure, the interrupt flag must be reset to '0' or cleared by the instruction:

CLI	CLear Interrupt flag.

Not all interrupts can be blocked as some are crucial and must get through at all costs. Many such circumstances may arise during the control of plant or when a power failure calls for immediate action. To enable the 6502 to distinguish between the two conditions, it has

two different input pins. One of these is the NMI or Non Maskable Input pin which cannot be blocked and the other the IRQ or Interrupt ReQuest pin which is masked off by the I flag.

When the 6502 receives an interrupt signal it always completes its current instruction before doing anything else. In the case of an IRQ interrupt it would then check the I flag and if not clear, continue until the program cleared this. Next, before going into the interrupt procedure, the contents of PC are saved on the stack (telling the 6502 where to return to), and then the PSW is saved. This saving of data is really only a half-measure as,almost certainly, A, X and Y would be changed during the interrupt procedure.

Next the I flag is set to '1' to prevent further interrupts and then the appropriate address for the interrupt routine is loaded into the PC. This is found at 65530 and 65531 ($FFFA_{16}$ and $FFFB_{16}$) for an NMI interrupt and 65534 and 65535 ($FFFE_{16}$ and $FFFF_{16}$) for an IRQ one. The interrupt routine is terminated with an instruction:

RTI	ReTurn from Interrupt.

On meeting this, the 6502 retrieves the PSW, resets I and looks in the PC for the return address.

Unfortunately the 6502 only does half the job at an interrupt and,as discussed in chapter 7 (page 7.7) A, X and Y should be saved on the stack. In the case of both X and Y this must be transferred into A before being pushed onto and pulled off of the stack.

The 6502 possesses one other interrupt instruction:-

BRK	BReaK

This initiates an interrupt sequence that, when called on the VIC terminates the current machine code program and returns the VIC to BASIC after first clearing the screen and then replacing the flashing cursor. When the BRK instruction is encountered, the 6502 in effect records the nature of the interrupt sequence. It then resets the PC by indexing this by one place, and then stores this address on the stack followed by the PSW. Following this, the 6502 then seeks the interrupt instruction or IRQ VECTOR at $FFFE (LSB) and $FFFF(MSB) in order to find what to do next. On the VIC this vector directs the program back into BASIC.

and leaves it there! In another environment, this vector would handle the break with the appropriate interrupt routine and then return to the main menu, retrieving the PSW and the return address from the stack.

To test this, run the following:-

PROGRAM 8.1

```
LDAIM 90
STA 7680
BRK
```

When run, this will print the diamond in 7680 and then go into the interrupt routine which, on the VIC directs the machine back to BASIC.

As the whole operation is completed in only a few cycles the diamond will not be visible on the screen and will be lost as this is cleared before re-entering BASIC. By using the assembler to disassemble the code at $FFFE and $FFFF, you will be able to find this address and from there to follow through the rest of this routine.

When the VIC encounters a BRK it resets the border to cyan, clears the screen, sets this to white and the letters to blue. It is when the 6502 chip is used as an element in a system that BRK comes into its own as the vectors can point to whatever subroutine the designers wish. On a computer like the VIC, the Commodore designers have made the decision and wisely pointed the 6502 along a path that will do least harm.

Signed Numbers

In all the mathematical exercises done so far, the numbers used have been treated as simple positive numbers. Thus, any arithmetical processes have dealt with these numbers as strings of eight bits. However, if negative integers are to be used in arithmetic, one of the bits must be used to indicate that a given byte represents a negative number. Bit 7, the left-most bit is used to do this, being set to a zero if the number is positive and a one if it is negative. By using one bit in this way, the magnitude of the number stored in the remaining seven bits is restricted to +127 to - 128. Conventionally a zero is used on bit 7 to indicate the presence of a positive number and a 1 for a minus number. One of the problems that arises from this usage is that, in theory, two forms can exist for the number zero, i.e. +zero and -zero:

$+0 = 00000000$

$-0 = 10000000$

In order to overcome the problem, the negative number is normally represented in what appears to be a weird form - TWO'S COMPLEMENT! Weird it may be, but it works!! In order to work out a two's complement representation take the number 38_{10}, which in binary is 00100110. First of all each bit of the positive number is switched from a 0 to 1 or vice versa i.e. to its COMPLEMENT:

$$00100110 \longrightarrow 11011001$$

Next 1 is added to this switched form or ONE'S COMPLEMENT i.e.

$$\begin{array}{r} 11011001 \\ + \quad\quad 1 \\ \hline 11011010 \end{array}$$

This yields the negative two's complement representation, i.e.

$$-38_{10} = 11011010_2$$

To understand the significance of this representation, three sums using it are illustrated below.

i) 38 - 38, which should of course yield zero.

$$\begin{array}{rcl} -38 & = & 11011010 \\ \underline{+38} & = & \underline{00100110} \\ 00 & = & 00000000 \end{array}$$

ii) $43_{10} - 38_{10}$

$$43_{10} = 00101011_2$$

$$-38_{10} = 11011010_2$$

i.e.

$$\begin{array}{rcl} -38 & = & 11011010 \\ +43 & = & \underline{00101011} \\ & & \underline{00000101} \quad = \quad \underline{5_{10}} \end{array}$$

iii) $24_{10} - 38_{10}$

$$24_{10} = 00011000_2$$

i.e. 24 = 00011000

-38 +11011010

11110010

As this answer has a 1 in bit 7, it is a negative two's complement representation. To convert this, first find the one's complement:

11110010 ⟶ 00001101

Next add on 1

i.e. 00001101

\+ 1

00001110 = -14_{10}

Overflows

As, in signed number arithmetic, the seven "magnitude" bits (i.e. 0 to 6) can only store a number up to +127, any attempt to store a number greater than this, will result in a 'carry' into bit 7 or as it is known in this case, an OVERFLOW. Consider the sum 100_{10} + 30

i.e.

100_{10}	=		01100100_2
+ 30_{10}		+	00011110_2
130_{10}			10000010_2

Thus 10000010 is a negative number as signified by the '1' in bit 7 and is therefore, in two's complement form. To convert this, first find the one's complement

01100100_2 ⟶ 10011011_2

and then add 1

10011011_2 ⟶ 10011100_2 = 156_{10}

The 6502 handles this situation by monitoring the accumulator and, when an overflow occurs, setting the overflow or V-flag. This flag can be tested by the instructions:-

BVC — Branch on oVerflow Clear.

BVS — Branch on oVerflow Set.

BVC tests the overflow flag and if it is clear, or not set (V=0) then a branch is executed.

BVS tests the overflow flag and, if it is set, (V=1) then a branch is executed.

When carrying out multiple precision arithmetic processes with signed integers, bit 7 must be treated as an internal carry and, when an overflow occurs, this must be transferred to the most significant byte.

Program 8.1 illustrates the use of BVC to test for an overflow as the accumulator is indexed with ones.

PROGRAM 8.2

```
CLC
LDAIM 0
ADCIM 1
BVC 252
STA 7900
RTS
END
```

When run, the content of the accumulator is increased progressively until the right-most seven bits are filled with ones. At the next increment, the seven ones reset to zeroes and a carry is generated that pops into bit seven. This sets the carry bit and stops the branching, allowing the program to run through to the RTS. The STA 7900 then prints the value at overflow as an off-reverse @ (i.e. 128_{10}).

As with other flags, provision exists for the control of the overflow flag and it can be cleared by the instruction:

CLV — CLear the 0Verflow flag.

- - - - - - - - -

Numerical Screen Output

In all the numerical examples to date screen output has been in VIC display code. While it is possible to interpret this using a table it is obviously necessary in a program to display numbers as numbers to the base 10. The major complication in this procedure lies in the fact that VIC display code is effectively a base 256 representation and can, thus display 0_{10} to 255_{10} using only one character, where base 10 displays would require 3.

Program 8.3 tackles this conversion task and firstly sets out to find if the number is greater than 200 - i.e. first digit is '2' - or if it is less than 200 and greater than 100 - i.e. first digit '1'. It then checks in a similar way for the tens and units and uses the stack to store the remainder (i.e. so far unprocessed portion) of the number while adding the conversion constant (48) to the accumulator to correct the display. In the example given, the number to be displayed - 152 - is loaded into the accumulator at the start of the program.

PROGRAM 8.3

```
        LDAIM 152    Put in number to be  O/P.
        CMPIM 200    Compare A with 200.
        BCC ONEHUND  Branch if number less than 200.
        SBCIM 200    Remove left most digit (carry already
                     set).
        PHA          Store remainder on stack.
        LDAIM 50     Load A with '2' for 2 x 100.
        STA 7680     Print A on screen.
        PLA          Retrieve A from stack.
        JMP TENS     Jump  to tens routine.
       *ONEHUND CLC  Clear carry.
        CMPIM 100    Compare A with 100.
        BCC TENS     Branch if less than 100.
        SBCIM 100    Remove left most digit (carry already
                     set).
        PHA          Store remainder on stack.
        LDAIM 49     Load A with '1' for 1 x 100.
        STA 7680     Print A on screen.
        PLA          Retrieve A from stack.
      *TENS CLC      Clear carry.
        LDYIM 0      Set Y to Zero.
        CMPIM 9      Compare A with 9.
        BCC ZEROTENS     Branch if A less than 9.
        INY          Increment Y.
        SBCIM 10     Subtract 10 from A (carry already set).
        CMPIM 9      Compare A with 9.
        BCS 247      Branch if A greater than 9.
```

..... cont'd.

PROGRAM 8.3 cont'd.

```
*ZEROTENS PHA   Store A on stack.
TYA             Transfer Y to A.
ADCIM 48        Add conversion constant to A
                (carry already clear).
STA 7681        Print A on screen.
PLA             Retrieve A from stack.
ADCIM 48        Add conversion constant to A
                (carry still clear).
STA 7682        Print A on screen.
RTS
```

When assembled the program looks as in Program 8.3A.

PROGRAM 8.3a

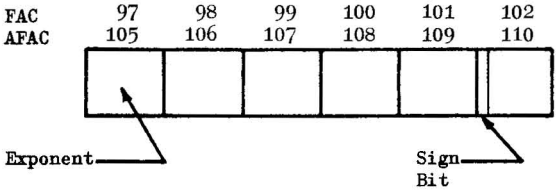
```
LDAIM 152  ┐
CMPIM 200  ├ Check if number less than 200,
BCC   12   ┘ if so branch.
SBCIM 200  ┐
PHA        │
LDAIM 50   ├ Subtract 200 from number and
STA   7680 │ print out digit '2' for 200.
PLA        │
JMP   860  ┘
CLC        ┐
CMPIM 100  ├ Check if number less than 100,
BCC   9    ┘ if so branch.
SBCIM 100  ┐
PHA        │
LDAIM 49   ├ Subtract 100 from number and print
STA   7680 │ out digit '1' for 100.
PLA        ┘
CLC        ┐
LDYIM 0    ├ Check if remainder of number less
CMPIM 9    │ than or equal to 9, if so, branch.
BCC   7    ┘
INY        ┐
SBCIM 10   │
CMPIM 9    │
BCS   247  ├ Count number of 10's left in
PHA        │ remainder and print out 10's
TYA        │ digit.
ADCIM 48   │
STA   7681 │
PLA        ┘
ADCIM 48   ┐
STA   7682 ┘ Print out unit's digit.
RTS
```

When run, this program will print '152' in the top left-hand corner of the screen. In general terms it can be used as a subroutine which prints out the contents of the accumulator.

Floating-point Numbers

With the integers used so far, the scope of the arithmetic processes carried out has been somewhat restricted. When working in BASIC, however, the binary floating-point constants have 10 digit precision and are displayed with 9 digits. Their exponents have a range of -38 to +37. Each constant is stored in five consecutive bytes of memory and, to ease the manipulation of these, two 'accumulators' are provided in memory locations 61_{16} to 66_{10} (97_{10} to 102_{10}) and 69_{16} to $6E_{16}$ (105_{10} to 110_{10}). These are known as the Floating-Point Accumulator (FAC) and Alternative Floating-Point Accumulator (AFAC).

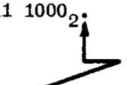
To store a binary number with 10 digits in base 10 form the floating point accumulators use four bytes of memory. This part of the floating-point representation is known as the MANTISSA and is stored in the middle four bytes. As is seen when running BASIC programs, very large or very small numbers are expressed in exponential form or scientific notation. Thus, 4079.013 can be expressed as 4.079013E+3 4.079013×10^3, while 0.0000417 can be expressed as 4.17E-5 or 4.17×10^{-5}. The exponent part of the number, expressed as a BINARY EXPONENT is stored in the first byte of the FAC and AFAC. As this has to store both positive and negative exponents, these need to be converted to a positive format by adding 128. Thus, an exponent of 20 would be stored as 128 + 20, i.e. 148, while an exponent of -20 would yield 128 - 20 i.e. 108. The sign of the mantissa is stored in byte 6 where a '1' on bit 1 signifies a negative mantissa and a '0' signifies a positive mantissa.



When loading a floating-point number, BASIC normalises its binary representation and, thus its left-most (most-significant) digit is always 1. Take, for instance the number $+1400_{10}$ (0578_{16}). Expressed in binary this is $0000\ 0101\ 0111\ 1000_2$. In this form, the binary number has an exponent of 2^0 (i.e. 1) and has an implied decimal point known as the RADIX - to the right of the least-significant

digit i.e.

$0000\ 0101\ 0111\ 1000_2$.



Decimal point or RADIX.

Putting the exponent in gives a representation:

$0000\ 0101\ 0111\ 1000_2 . *2^0$

and when NORMALISED, i.e. the decimal point moved just past the left-most significant digit:

$.1010\ 1111\ 0000\ 0000 \quad *2^{11}$

i.e. the decimal point has been moved 11 places to the left and so the number needs to be multiplied by 2" to return to its original value.

Finally this is padded up with 0's in its least-significant bytes to form a four byte mantissa.

i.e.	Byte 2	Byte 3	Byte 4	Byte 5
	1010 1111	0000 0000	0000 0000	0000 0000

The exponent of 11_{10} is converted to the correct format by adding 128_{10} to its raw form, i.e. it becomes 139_{10} or $1000\ 1011_2$ and this byte is added to the first location in the accumulator, i.e. the representation becomes:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
1000 1011	1010 1111	0000 0000	0000 0000	0000 0000

The sign of the mantissa is expressed in byte 6 by the condition of its most-significant bit, this being set to a '1' for a negative number and a '0' for a positive one. Thus, to represent $+1400_{10}$, bit 7 of byte 6 of FAC or AFAC is set at '0'. The other bits of this byte may be set at some other number. Thus the final floating-point accumulator representation of $+1400_{10}$ is:

Byte 1	Byte 2	Byte 3	Byte 4
1000 1011	1010 1111	0000 0000	0000 0000

Byte 5	Byte 6
0000 0000	0000 0000

To use this accumulator directly from machine-code is clearly none too easy, as the data has to be loaded into the memory locations in the correct format. However, some built-in sub-routines are accessible to enable this to be done. For instance, a two byte integer can be converted into floating-point and loaded into the floating-point accumulator (FAC) by means of the sub-routine at $D391_{16}$ (53869).

To accomplish this, the least-significant byte is loaded into the Y register and the MSB into the accumulator. Then a JSR $D3 91 will convert this integer into floating-point and load it into FAC. This sub-routine is demonstrated below in Program 8.4 where the LSB and MSB of an integer are converted into a floating point number.

PROGRAM 8.4

```
LDY LSB
LDA MSB
JSR $D3 91
RTS
```

These two floating-point accumulators - FAC and AFAC are used for the manipulation of floating-point data and a means is provided for transferring floating-point data into them.

Taking AFAC, for instance, the subroutine for loading floating-point data from memory resides at $DA 8C.

and it moves the contents pointed to by the Accumulator (LSB) and the Y register (MSB) into AFAC. Thus, if the data is stored at 0350_{16} onwards, Program 8.5 carries out this transfer.

PROGRAM 8.5

```
LDAIM 80
LDYIM 3
JSR $DA 8C
RTS
```

A similar subroutine re-locates data into FAC, this being located at DB $A2_{16}$ (56226_{10})

Unfortunately, as with the other routines using the floating-point accumulators a measure of faith has been called for as, in no case has the content of either FAC or AFAC been visible. To rectify this a hybrid BASIC/machine-code command is utilised:

The USR Command

This command allows the transfer of data between FAC and a machine-code program. For instance the line B = USR (A) in a BASIC program causes the system to place the value of A into the floating-point accumulator. It then jumps to a machine code routine whose address it finds at 1_{16} (1_{10}) LSB and 2_{16} (2_{10}) MSB. Any routines there that use FAC will then pick up the value set by A and when the particular arithmetic process has been carried out, place the result into FAC. These contents will then be stored as the constant and can thus be printed out by means of the standard BASIC PRINT statement.

To test this, a number can be placed into the floating-point accumulator by means of the USR function and then printed out. This is done by means of the two programs - PROGRAM 8.6a and PROGRAM 8.6b.

PROGRAM 8.6a

```
20 000 PRINT"▽"
20 010 POKE 1,60
20 010 POKE 2,3
20 030 B=1400
20 040 A=USR(B)
20 050 PRINT"A=";A
```

$60_{10}=3C_{16}$
$3_{10}=03_{16}$
} Address = 033C

PROGRAM 8.6b

```
828 RTS
```

When this program is run, an address - $033C_{16}$(828_{10}) - is loaded into locations 1 and 2 by lines 20 010 and 20 020.

When line 20 040 is run, the argument B (1400) is loaded into the floating-point accumulator and identified as A. Control is then handed over to the machine code program at$ 033C (828). At this point, the instruction RTS returns control to 20 050. This line prints out the value stored in FAC, which has not been modified by the machine code routine.

This routine offers a way of loading any number, which is valid in BASIC, into the floating-point accumulator. It also offers a way - albeit rather round-about - of examining the contents of FAC. This is not as straightforward a process as might be imagined, as most BASIC commands use the FAC when operating, thus even a PEEK command changes its contents. However, if the contents are examined in machine-code, immediately after being set they can be seen before BASIC gets its hands on them again. To do this, program 8.6b should be ammended to examine memory locations 61_{16} to 66_{16} (97_{10} to 102_{10}) and then to print these on the screen. This done in Program 8.7.

PROGRAM 8.7

```
LDXIM 6          Set up loop counter.
LDAX 96          Load one Byte of FAC.
STAX 7908        Print byte on screen.
DEX              Decrement loop counter.
BNE   247        Branch if more to do.
RTS
```

PRINT CONTENTS OF FAC ON SCREEN

As this program prints the contents in VIC display code, program 8.6a should also be modified to decode this - as below:

PROGRAM 8.8

```
20 000 PRINT"■"
20 010 POKE1,60
20 020 POKE2,3
20 030 B=1400
20 040 A=USR(B)
20 050 PRINT"A=";A
20 060 FORX=0TO5
20 070 PRINT PEEK(7909+X);"  ";
20 080 NEXT X
```

This program simply PEEKs the locations that display the contents and then prints the answer.

When run, this will display:

A = 1400

and the contents:

139 175 0 0 0 47

When compared with the calculated contents on page 8.11 it will be seen that bytes 1 - 5 are identical, while byte 6 contains something different. In fact it is only the most-significant bit that is of consequence and, as this is set to a '0' a positive mantissa is indicated. Just to verify this, change line 20 030 in program 8.8, to read

20 030 B = -1400

On re-running, byte 6 of the FAC will now read 128+47 i.e. its MSB has been re-set to a '1'.

Those earlier subroutines that required faith, can now be tested using programs 8.8 and 8.9. Take program 8.4. This, it was said converted an integer into floating-point and loaded it into FAC. Program 8.9, below, tests this by printing out the contents of FAC, it uses $1400_{10}(0578_{16})$ as the integer, i.e. LSB=$78_{16}=120_{10}$ and MSB=$05_{16}=5_{10}$. They are loaded into Y (LDYIM 120) and A (LDAIM 5)

PROGRAM 8.9

```
LDYIM 120
LDAIM 5
JSR   $D3 91
LDXIM 6
LDAX 96
STAX 7908
DEX
BNE 247
RTS
```

CONVERT INTEGER TO FLOATING-POINT

Program 8.8 should then be modified so as not to put 1400 into FAC i.e. line 20 030, should read

20 030 B=1 (or any number other than 1400).

The suite of programs can then be executed using a RUN 20000 and will display the value of A as 1400.

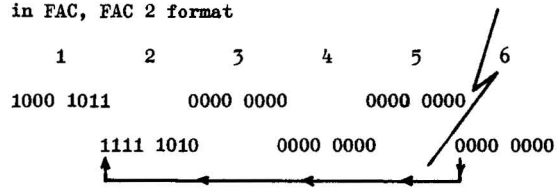
Doubters will also, most probably, wish to see program 8.5 (page 8.12) run, i.e. to load floating-point data from memory. This is done in programs 8.10 and 8.11 where the number +2000 is used. However, a slight complication arises here, as floating-point numbers are stored in a slightly different format in memory from those in FAC and AFAC. Taking +2000 for instance, this is stored in FAC and AFAC as 139 250 0 0 0 0 or:

FAC AFAC location	97 105	98 106	99 107	100 108
	139 1000 1011	250 1111 1010	0 0000 0000	0 0000 0000

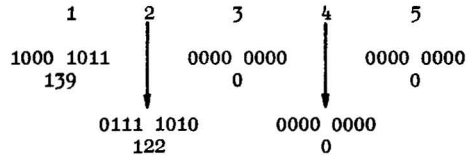
FAC AFAC location	101 109	102 110
	0 0000 0000	0 0000 0000

However, when storing large quantities of data a more economical format is used, as only bit 7 of byte 6 (memory location 102_{10} or 110_{10}) is really needed. Obviously, this has to be stored somewhere, as it gives the sign of the mantissa, so one spare bit must be found elsewhere. Such a spare location exists in byte 2 on its left-most bit as, during normalisation of the mantissa the decimal point is moved until this bit is filled by a '1'. Thus as it is always set to '1' its location can be used for another purpose and an allowance made for this by adding a 1 to the left of the mantissa when using this. This redundant bit, therefore, is used to store the sign bit i.e. a '0' for positive and a '1' for negative and byte 6 can be dispensed with. The representation of +2000 in floating point variable format then becomes:

a) in FAC, FAC 2 format



b) in variable format



To use the "load FAC from memory" subroutine the variable data has to be loaded into memory and this is most easily done using a direct POKE program - PROGRAM 8.10, below:-

PROGRAM 8.10

```
POKE 848,139
POKE 849,122
POKE 850,0
POKE 851,0
POKE 852,0
```

The machine code program to use this subroutine sets a pointer to the data in 848_{10} (0350_{16}) onwards and then transfers it with a JSR $DB A2.
Program 8.11 should be executed by means of the BASIC program 8.8

PROGRAM 8.11

```
LDAIM 80
LDYIM 3
JSR $DB A2
LDXIM 6
LDAX 96
STAX 7908
DEX
BNE 247
RTS
```

TRANSFER DATA FROM MEMORY TO FAC

When run, program 8.11 will load the floating-point representation of +2000 into FAC and print this onto the screen. BASIC program 8.8 will then decode the graphic data and print this out in numbers.

Naturally AFAC has a similar facility for transferring data, this being located at $DA8C_{16}$ (55948_{10}).

Once the data is loaded into FAC or AFAC a facility exists for copying from one to the other, that for copying AFAC into FAC residing at DB $F6_{16}$ (56316_{10}).

A similar subroutine copies the other way i.e. from AFAC to FAC, this being located at DC $0C_{16}$ (56332_{10})

Exercise 8.1

Write a suite of programs to load 2000 into AFAC from memory and then print out the contents of this from 7910 onwards. Then transfer AFAC into FAC and again print out the contents, this time from 7932 onwards.

One possible answer given on page 9.11.

Floating-point Sub-routines

Having got numbers into FAC and AFAC, use can be made of the built-in COMMODORE subroutines that offer the opportunity to handle these six-byte monsters with reasonable ease. A word of warning here, though, when using these subroutines one must first know where they are - obvious, eh? Obvious perhaps! However, to date, many of these have already had three addresses - old ROM, 2.0 BASIC and 4.0 BASIC on the PET and they are now relocated again for VIC BASIC i.e. BASIC V2. The addresses quoted in the book are for the VIC-20 and are correct for the models of VIC-20 used. However, future changes are beyond the author's control!!

A further caveat must be given along with the advice to try these. One must know where these subroutines pick up their data from and where they deposit the result, if they are to be used safely. Many of them start with a short initialisation section that prepares the data and deposits it in the right place for action. The subroutine that moves data from memory into AFAC, for instance, starts by transferring its data address into $1F_{16}$(31_{10}) and

23_{16} (35_{10}) from the accumulator and Y register, hence when started at DA 8C
it expects to find the data address in A and Y. However, at DA 90_{16} (55952_{10}) the subroutine proper begins and then retrieves its data address from 22_{16} (34_{10}) and 23_{16} (35_{10}). Thus it can be entered early with the data address in A and Y, or a few bytes later with its address in $34 and $35. As an interesting exercise it may be of use to disassemble this subroutine and try and follow through the various stages. This can be done by selecting '2' for List at the MENU and then listing from 55948_{10}.
These subroutines are just collections of 6502 instructions much as you yourself should be writing by chapter 8. Where some of these will differ is that they very cleverly use bits of other subroutines to save space and it is these JSR and JMP jaunts that are likely to prove difficult!

Appendix III lists the subroutines available for handling floating-point numbers and the exercises below are designed to give practice at using these.

Exercise 8.2

Write a program suite to input the numbers 1.047 and 4038.22 into a machine code program. Multiply these together and then take the square-root of the sum. Print out this answer on the screen from BASIC.

Second thoughts!! .....

This is not as easy as it seems, the original plan looked like this:

```
20 000 PRINT "  "
    10 POKE 1,60
    20 POKE 2,3
    30 INPUT B
    40 A=USR(B)
                    Puts B into FAC
                                    828 JSR $DC 0C (FAC - AFAC)
                                        RTS
20 050 POKE 1,72
    60 POKE 2,3
    70 INPUT D
    080 C=USR(D)
                    Puts D into FAC
                                    832 JSR $DA 2B(MULT)
                                    835 JSR $DF 71
20 090 PRINT "C=";C                 838 RTS
```

That was the plan, however, it won't work! It fails because the program suite assumes that the FAC's contents stay put while they are, in fact, constantly changing as BASIC runs a program. Following line 20 040 FAC contains B and after the JSR (FAC→AFAC) both FAC and AFAC contain B. However, in executing lines 20 050 to 20 080 the operating system utilises FAC and AFAC and, thus, changes the contents of these. Thus when JSR (MULTIPLY) is called the contents of FAC and AFAC that are multiplied together are not those expected.

The problem can only be overcome by storing the AFAC contents in memory while returning to BASIC. This is done by means of the subroutine at DB $C7_{16}$ (56263_{10})

This subroutine copies the contents of AFAC into the five bytes of memory starting at the address stored in 49 (73_{10}) and $4A_{16}$ (74). Program 8.15 illustrates this, putting AFAC into 900_{10} (0384_{16}) onwards.

PROGRAM 8.15

```
LDAIM 132  )   Load LSB of address - 132₁₀ (84₁₆) -
STAZ  73   )   into 73.
LDAIM 3    )   Load MSB of address - 3₁₀ (03₁₆) -
STAZ  74   )   into 74.
JSR $DB C7
RTS
```

COPY AFAC INTO MEMORY

Theaction of this can be checked by running the direct program: FOR X=0TO5:PRINT PEEK(900+X):NEXT X

To use this subroutine in Exercise 8.2, it is necessary to reload the data into AFAC. This is done by the subroutine at DA $8C_{16}$ (55948_{10}).

In order to operate, the subroutine needs to be told where to find the data and this is done by loading the address of the first byte of data into the accumulator (LSB) and the Y register (MSB). Thus a "reload AFAC" program to recover data from 900 onwards would look like:

PROGRAM 8.16

```
LDAIM 132      Load LSB of address.
LDYIM 3        Load MSB of address.
JSR $DA 8C     Load AFAC from memory
RTS
```

Perhaps now is a suitable time to try exercise 8.2. One possible answer is given on page 9.12.

Other Available Subroutines

Appendix III lists the 17 subroutines that have been found most useful when dealing with floating-point numbers from machine code. Those not yet discussed are covered below:

Addition

Using the subroutine at $D86A_{16}$ (55402_{10})
the floating-point numbers in FAC and AFAC are added together and the sum loaded into FAC. This is demonstrated in programs 8.17 and 8.18.

PROGRAM 8.17

```
20 000  PRINT"▽"
20 010  POKE 1,60
20 020  POKE 2,3
20 030  INPUT B
20 040  A=USR(B)
20 050  RUN 20 060
20 060  POKE 1,72
20 070  POKE 2,3
20 080  INPUT D
20 090  C=USR(D)
20 100  PRINT"C=";C
```

PROGRAM 8.18

```
828  LDAIM 132
     STAZ 73
     LDAIM 3
     STAZ 74
     JSR  $DB C7      } Store FAC in memory.

     RTS
840  LDAIM 132
     LDYIM 3
     JSR  $DA 8C      } Retrieve data from memory, store in AFAC.

     JSR  $D8 6A      } Addition subroutine

     RTS
```

ADD TWO FLOATING-POINT NUMBERS

When the program at 20000 is run, it requests two inputs and then prints out the sum of these.

Subtraction

Programs 8.17 and 8.18 can be used to demonstrate this by inserting the subroutine $D853_{16}$ (55379) at 848 and 849, i.e.

PROGRAM 8.19 (Part)

```
847 JSR $D853
```

This can be run by a RUN 20000 and will print out the answer. It subtracts the second input from the first.

Division

Again replace 848 and 849.

PROGRAM 8.20 (Part)

```
847 JSR $DB 12
```

and RUN 20000. The program divides the first input by the second.

Exponentiation

Replace 848 and 849.

PROGRAM 8.21 (Part)

```
847 JSR $DF 7B
```

and RUN 20000. The program raises the first inputted number to the power of the second input.

Other routines need only one input and operate on this alone these are:

Log

The subroutine at D9 EA (55786_{10}) computes the natural logarithm or $\log_e$ (log to the base e). Program 8.22 demonstrates this subroutine in use.

PROGRAM 8.22

```
828 JSR $D9 EA
831 RTS
```

It is called by Program 8.23.

PROGRAM 8.23

```
20 000  PRINT"☑"
20 010  POKE 1,60
20 020  POKE 2,3
20 030  INPUT B
20 040  A=USR(B)
20 050  PRINT"A=";A
```

A RUN 20000, activates both routines and prints out the $\log_e$ of the input value.

The three functions SIN ($E268_{16}$)
COS ($E261_{16}$)
and TAN ($E2B1_{16}$)
can be incorporated into a program suite such as 8.22/23. In each case, the input value is entered in radians and the computed function is returned in FAC.

Many other subroutines exist within the VIC ROM and, no doubt, the ones most needed are those not listed here.

However, when using these built-in sub-routines, the major problem lies in knowing exactly how to integrate these into any particular program as the sub-routines start-points are well documented. These problems broadly revolve around where the subroutine gets its data from and where it deposits its results. On the latter point, as will have been seen, the FAC is a common place in which to place results.

As to the linking in of data, the first few lines of a sub-routine should give the clue. First, these lines should be disassembled using the assembler LIST function. Next these should be examined for the first use of A, X and Y as once these are re-loaded in the sub-routine, any data originally in them will have been destroyed. If addresses are involved certain zero-page locations are popular such as $22_{16}/23_{16}$ ($34_{10}/35_{10}$) as well as $49_{16}/4A_{16}$ ($73_{10}/74_{10}$). Their use will be revealed by STAZ, LDXZ or LDYZ instructions.

Similarly the deposition of processed data will be revealed by the last few instructions and if FAC and AFAC are involved their memory locations should feature.

Finally if logic (and all else!) fails there's always the sledgehammer technique left, i.e. just trying it. If this is attenpted, keep good records as constant reloading after crashes can be time-consuming. Of course, after a crash the machine-code program is not lost and the computer can simply be turned off and then on quickly to regain control. The assembler will be gone but the machine-code treasure will remain.

Remember that the sub-routines in ROM are only machine-code like that you have produced and, once disassembled should be quite comprehensible, if at times somewhat involved. Don't be afraid, therefore, to use things not detailed here!

CHAPTER 9

SOLUTIONS TO EXERCISES

Chapter 1

Exercise 1.1 (Page 1.9)

```
START ADD ? 828
LDAIM 1
STA 7680
RTS
END
```

Exercise 1.2 (Page 1.10)

```
START ADD ? 828
LDAIM 6
STA 7680
LDAIM 18
STA 7681
LDAIM 5
STA 7682
LDAIM 4
STA 7683
RTS
END
```

Exercise 1.3 (Page 1.10)

```
START ADD? 828
LDAIM 24
STA 7680
STA 7701
STA 8164
STA 8185
RTS
END
```

Exercise 1.4 (Page 1.13)

```
START ADD? 826
LDAIM 26
LDXIM 1
TAY
TXA
TYA
STA 7680
STX 8185
RTS
END
```

Exercise 1.5 (Page 1.13)

```
START ADD? 828
LDAIM 90
LDXIM 42
LDYIM 5
STX 900
TAX
TYA
LDY 900
STA 7680
STA 7701
STX 8164
STY 8185
RTS
END
```

Chapter 2

Exercise 2.1 (Page 2.6)

i)

```
START ADD ? 828
LDAIM 3
JSR 900
STA 7680
RTS
END
```

ii)

```
START ADD ? 900
STA 950
ADC 950
RTS
```

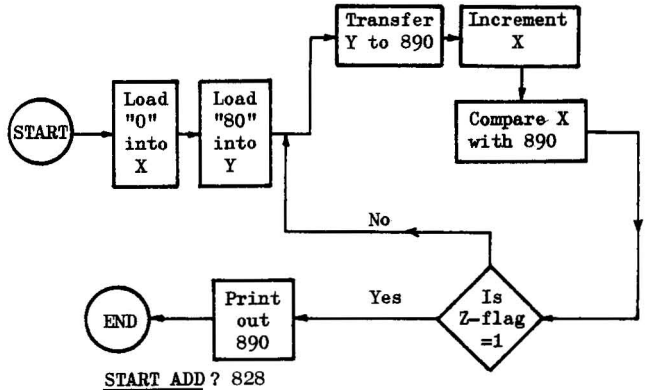
Exercise 2.2 (Page 2.8)

```
START ADD ? 828
LDYIM 100
DEX
BEQ 3
JMP 830
STX 7680
RTS
END
```

Exercise 2.3 (Page 2.11)

```
START ADD? 828
LDAIM 83
STA 890
LDYIM 0
INY
CPY 890
BEQ 3
JMP 835
STY 7780
RTS
```

Exercise 2.4 (Page 2.13)




```
START ADD? 828
LDXIM 0
LDYIM 80
STY 890
INX
CPX 890
BNE 250
STX 7680
RTS
END
```

Exercise 2.5 (Page 2.16)

```
LDYIM 1
STY   900
LDXIM 90
DEX
CPX   900
BPL   250
STX   7680
RTS
```

Chapter 3

Exercise 3.1 (Page 3.1)

The LDXIM instruction at 828 must be changed to a LDYIM. From the Table 1 (Appendix I) the object code for this is 160, so a

POKE 828,160

does the trick.

The STAX at 832 has to be changed to an STAY (object code 153)

i.e. POKE 832,153

Lastly, the DEX at 835 needs to be replaced by a DEY (object code 136)

i.e. POKE 835,136

Thus the final program reads:-

```
LDYIM 100
LDAIM 90
STAY 7679
DEY
BNE   250
RTS
```

Exercise 3.2 (Page 3.2)

```
LDXIM 100      Load X with '100'.
STX   900      Store X in 900.
LDXIM 0        Load X with '0'.
LDAIM 42       Load A with 42 (asterisk).
STAX  7679     Store A in ( 7679 + X).
INX            Increment  X.
CPX   900      Compare X with 900.
BNE   247      Branch if not equal.
RTS            Return from subroutine.
```

Chapter 4

Exercise 4.1 (Page 4.6)

$$1807_{16} = 1 \times 4096 + 8 \times 256 + 0 \times 16 + 7 \times 1$$
$$= 4096 + 2048 + 07$$
$$= 6151_{10}$$

$2AFA_{16}$ = 2 x 4096 + 10 x 256 + 15 x 16 + 10 x 1
=
= 11002_{10}

i.e. $6151_{10} + 11002_{10} = 17153_{10}$

Program to add

```
START ADD? 828
CLC
CLD
LDAIM $00 07
ADCIM $00 FA
STA 7890
LDAIM $00 18
ADCIM $00 2A
STA 7888
RTS
END
```

Screen display gives answer of - A i.e. 67_{10}, 01_{10}

Converting this to hex gives:

$67_{10} = 4 \times 16 + 3 \times 1 = 43_{16}$

$01_{10} = 0 \times 16 + 1 \times 1 = 01_{16}$

Thus $67_{10}\ 01_{10} = 4301_{16}$
$= 4x4096+3+256+0x16+1$
$= 17153_{10}$

Cross-checking this by hex addition:

```
  1807
+ 2AFA
= 4301
```

Exercise 4.2 (Page 4.8)

```
START ADD? 828
CLD
SEC
LDAIM 32
SBCIM 88
STA 7890
LDAIM 3
SBCIM 2
STA 7888
RTS

Ans = @ ■
    = 0 200
```

Exercise 4.3 (Page 4.8)

```
CLD             Clear decimal to make sure in
                binary mode.
CLC             Clear carry flag.      }
LDAIM $00 2C    Load LSB 1.            }  Add LSB's
ADCIM $00 90    Add LSB 2 to A.        }  together.
STA   900       Sum of LSB's.          }

LDAIM $00 01    Load MSB 1.            }  Add MSB's
ADCIM $00 01    Add MSB 2.             }  together
STA   901       Sum of MSB's.          }  incl.carry
                                          (if there).
SEC             Set carry in preparation
                for subtraction.
LDA   900       Load LSB 3 (of 500)    }  Subtract
SBCIM $00 F4    Subtract LSB 1+2       }  LSB's
                from LSB3.             }  and
STA   7911      Display sum of LSB's.  }  display.

LDA   901       Load MSB 3 (of 500).   }  Subtract
SBCIM $00 01    Subtract MSB 1+2       }  MSB's
                from MSB 3.            }  and
STA   7910      Display sum MSB's.     }  display
RTS
```

When run this should display @ ■ or 0,200.

Exercise 4.4 (Page 4.13)

A = 1	B = 0	C (output) = 0
A = 0	B = 1	C = 0
A = 1	B = 1	C = 1

Exercise 4.5 (Page 4.14)

$$149_{10} = 10010101_2$$

$$52_{10} = 00110100_2$$

$$149 \text{ AND } 52 = 00010100_2$$

$$= 16 + 4 = 20_{10}$$

Exercise 4.6 (Page 4.18)

	128	64	32	16	8	4	2	1
100 =	0	1	1	0	0	1	0	0
87 =	0	1	0	1	0	1	1	1
75 =	0	1	0	0	1	0	1	1
99 =	0	1	1	0	0	0	1	1
57 =	0	0	1	1	1	0	0	1
94 =	0	1	0	1	1	1	1	0
27 =	0	0	0	1	1	0	1	1

Thus:

i)

100	0	1	1	0	0	1	0	0
AND 87	0	1	0	1	0	1	1	1
=	0	1	0	0	0	1	0	0

= 01000100_2

= $0 + 64 + 0 + 0 + 0 + 4 + 0 + 0 = 68_{10}$

```
LDAIM 100
ANDIM 87
STA 7900
RTS
END
```

When run gives a - (68_{10})

ii)

75	0	1	0	0	1	0	1	1
OR 27	0	0	0	1	1	0	1	1
=	0	1	0	1	1	0	1	1

= 01011011

= $0 + 64 + 0 + 16 + 8 + 0 + 2 + 1 = 91_{10}$

```
LDAIM 75
ORAIM 27
STA 7900
RTS
END
```

When run gives a + (91_{10})

iii)

99	0	1	1	0	0	0	1	1
EOR 57	0	0	1	1	1	0	0	1
	0	1	0	1	1	0	1	0

= 01011010

= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 0 = $\underline{90_{10}}$

```
LDAIM 99
EORIM 57
STA 7900
RTS
END
```

When run gives a diamond (90_{10}).

iv)

100	0	1	1	0	0	1	0	0
AND 87	0	1	0	1	0	1	1	1
=	0	1	0	0	0	1	0	0
EOR 94	0	1	0	1	1	1	1	0
	0	0	0	1	1	0	1	0

= 00011010

= 0 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = $\underline{26_{10}}$

```
LDAIM 100
ANDIM 87
EORIM 94
STA 7900
RTS
END
```

When run gives a Z (26_{10}).

Exercise 4.7 (Page 4.20)

86_{10} in BCD = 10000110

which, in binary = 134_{10}

Thus a program is:

```
LDAIM 134
ANDIM 15
STA 7681
LDYIM 4
LDAIM 134
LSRA
DEY
BNE  252
STA  7680
RTS
```

When run this gives an output of HF (i.e.86)

Exercise 4.8 (Page 4.24)

Written to multiply together the numbers 3 and 4.

```
LDXIM 3
STX  901
LDXIM 4
STX  902
LDYIM 8

— — —      below here as program 4.12

LDAIM 0
CLC
LSR  902
BCC  4
CLC
ADC  901
ASL  901
DEY
BNE  240
STA  7680
RTS
```

When run this should print an L (12) in 7680.

Chapter 5 (Page 5.6)

Exercise 5.1 OR

```
        LDAIM 42                LDAIM 42
        LDXIM 88                LDXIM 88
        JMP   LOOP3             JMP 858

*LOOP1  STAX 7679               STAX 7679
        DEX                     DEX
        BNE   LOOP1             BNE 250
        LDAIM 90                LDAIM 90
        LDXIM 88                LDXIM 88
*LOOP2  STAX  7767              STAX 7767
        DEX                     DEX
        BNE   LOOP2             BNE 250
        LDAIM 83                LDAIM 83
        LDXIM 88                LDXIM 88
        JMP   END

*LOOP3  STAX 7833               STAX 7833
        DEX                     DEX
        BNE   LOOP3             BNE 250
        LDAIM 83                LDAIM 83
        LDXIM 88                LDXIM 88
        JMP   LOOP1             JMP 835

*END    RTS                     RTS
        END
```

Chapter 7

Exercise 7.1 (Page 7.5)

POKE 829,32 (If program loaded from 828)

The program should then read:-

```
LDXIM 32
STX   900
JSR   65508
BEQ   251
JSR   65490
CMP   900
BNE   243
RTS
```

Exercise 7.2 (Page 7.11)

```
LDXIM 4
TXA
PHA
PHP
JSR $FF E4
BEQ 251
JSR $FF D2
PLP
PLA
TAX
DEX
BNE 239
RTS
```

In the above program, the PSW could have been loaded into stack before the X register (via the accumulator) was, as long as it was then retrieved first, i.e. LIFO operation.

Exercise 8.1

The data, +2000 is to be stored from 900_{10} (0384_{16}) onwards. This data is:

139 122 0 0 0

It can be entered using:

PROGRAM 8.12

```
POKE 900,139
POKE 901,122
POKE 902,0
POKE 903,0
POKE 904,0
```

The machine code routine is:

PROGRAM 8.13

```
LDAIM 132       Set up address for subroutine.
LDYIM 3         03(16) = 03(10) : 84(16) = 132(10)
JSR $DA 8C      Put data into AFAC from memory.

LDXIM 6
LDAX 96
STAX 7909       Print FAC onto screen.
DEX
BNE  247
```

..... cont'd.

PROGRAM 8.13 .... cont'd.

```
JSR $DB FC      Transfer data AFAC → FAC.

LDXIM 6
LDAX 96
STAX 7931
DEX
BNE 247
RTS
```

TRANSFER AFAC to FAC

The BASIC program is:

PROGRAM 8.14

```
20 000  PRINT "▽"
20 010  POKE 1,60
20 020  POKE 2,3
20 030  B=1                (for example)
20 040  A=USR(B)
20 050  PRINT"A=";A
20 060  RUN 20070
20 070  FOR X = 0TO5
20 080  PRINT PEEK(7910+X);"▌";
20 090  NEXT X
20 100  PRINT
20 110  FOR X=0TO5
20 120  PRINT PEEK(7932+X);"▌";
20 130  NEXT X
```

When run this should print out:

```
A= 2000
  129  128  0  0  0  0
  139  250  0  0  0  122
READY
```

```
A▽ @@@
▽■ @@┘
```

Exercise 8.2

```
20 000  PRINT "▽"          20 060  POKE 1,70
20 010  POKE 1,60          20 070  POKE 2,3
20 020  POKE 2,3           20 080  INPUT D
20 030  INPUT B            20 090  C=USR( D )
20 040  A=USR(B)           20 100  PRINT "C=";C
20 050  RUN 20060
```

Exercise 8.2 cont'd.

```
LDAIM 132
STAZ 73
LDAIM 3
STAZ 74
JSR $DB C7
RTS
LDAIM 132
LDYIM 3
JSR $DA 8C
JSR $D8 2B
JSR $DF 71
RTS
```

Exercise A1.1

i) $0000\ 0011_2 = 0+0+0+0+0+0+2+1$
$= \underline{3_{10}}$

ii) $0000\ 0100_2 = 0+0+0+0+0+4+0+0$
$= \underline{4_{10}}$

iii) $1000\ 0000_2 = 128+0+0+0+0+0+0+0$
$= \underline{128_{10}}$

iv) $1000\ 0011_2 = 128+0+0+0+0+0+2+1$
$= \underline{131_{10}}$

v) $1011\ 0111_2 = 128+0+32+16+0+4+2+1$
$= \underline{183_{10}}$

vi) $0111\ 0011_2 = 0+64+32+16+0+0+2+1$
$= \underline{183_{10}}$

Exercise A1.2

i) $0009_{16} = 0x409+0x256+0x16+9x1$
$= 0+0+0+9$
$= \underline{9_{10}}$

ii) 0013_{16} = 0x4096+0x256+1x16+3x1
= 0+0+16+3
= 19_{10}

iii) $00A5_{16}$ = 0+0+10x16+5x1
= 160+5
= 165_{10}

iv) $0AAE_{16}$ = 0+10x256+10x16+14x1
= 2560+160+14
= 2734_{10}

v) $000E_{16}$ = 0+0+0+14
= 14_{10}

vi) $011A_{16}$ = 0+256+16+10
= 282_{10}

vii) $00EA_{16}$ = 0+0+14x16+10
= 224+10
= 234_{10}

viii) $F0A3_{16}$ = 15x4096+0+10x16+3
= 61440+160+3
= 61603_{10}

Exercise A1.3

i) 4_{10} = 0100_2(BCD)

ii) 10_{10} = 1x10+0
= 0001 0000_2(BCD)

iii) 77_{10} = 7x10+7
= 0111 0111_2(BCD)

iv) 97_{10} = 9x10+7
= 1001 0111_2(BCD)

v) 53_{10} = 5x10+3
= 0101 0011_2(BCD)

vi) 102_{10} = 1x100+0x10+2x1
= 0001 0000 0010_2(BCD)

vii) 953_{10} = 9x100+5x10+3x1
= 1001 0101 0011_2(BCD)

viii) 2579_{10} = 2x1000+5x100+7x10+9x1
= 0010 0101 0111 1001_2(BCD)

Exercise A1.4

i) 0000 0001_2(BCD) = 0x10+1x1
= $\underline{1_{10}}$

ii) 0000 1001_2(BCD) = 0x10+9x1
= $\underline{9_{10}}$

iii) 0001 0101_2(BCD) = 1x10+5x1
= $\underline{15_{10}}$

iv) 0010 0000_2(BCD) = 2x10+0x1
= $\underline{20_{10}}$

v) 0100 1001_2(BCD) = 4x10+9x1
= $\underline{49_{10}}$

vi) 1010 0011_2(BCD)

This is not a valid BCD number as the first nibble, 1010_2 = 10_{10} i.e. is greater than allowed in BCD.

vii) 1001 0111_2(BCD) = 9x10+7x1
= $\underline{97_{10}}$

viii) 1000 1000_2(BCD) = 8x10+8x1
= $\underline{88_{10}}$

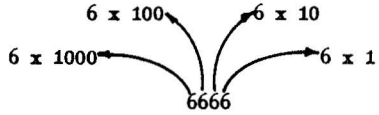
APPENDIX I

BINARY, BINARY-CODED DECIMAL AND HEXADECIMAL NOTATION

Counting systems in general use throughout the world use the decimal system and this has been developed to count on past 10 and also below 1. In this standard the digits to the left of a number are of greater value than those to the right. For instance, in the number 66, the first 6 has a value 10 times the second, i.e.

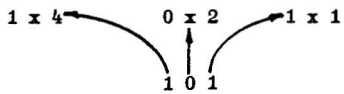


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit is a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to left is greater being multiplied by the base.

The computer, being basically electrical in operation can only recognise two states, on or off or '0' and '1' and thus, uses the Binary system - base 2. Thus any number in binary consists simply of 0's and 1's, or electrically speaking, offs and ons (or electronically, zero volts and some volts). To count past one, the binary system must resort to place-value notation and, as with other cases, the multiplying factor is the base i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e. 4 + 0 + 1 = 5. Clearly the plethora of bases presents a problem when representing numbers, as in base 10, 101

represents one hundred and one while in binary (base 2) 101 represents 5. To overcome this ambiguity a convention exists when representing numbers in that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

101_{10} = One hundred and one in base ten.

101_{2} = Five in base ten.

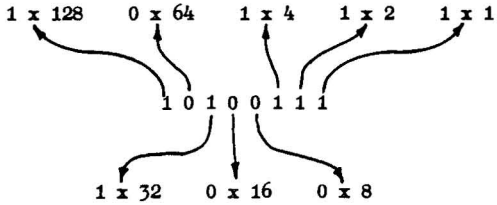
The present-day generation of computers (1982-style) use eight bit registers or memories and can, thus represent numbers up to 11111111_2 i.e. in base 10:

$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255_{10}$

1	1	1	1	1	1	1	1	Digit
128	64	32	16	8	4	2	1	Equivalent in base 10

Fig. A1.1

By way of example let's take one more conversion - say 10100111_2.



Thus 10100111 = 1 x128 + 0 x 64 + 1 x 32 + 0 x 16 +
0 x 8 + 1 x 4 + 1 x 2 + 1 x 1
= 128 + 32 + 4 + 2 + 1
= 167_{10}

Just to check your understanding, have a go at the following:

Exercise A1.1

Calculate the value of the following in base 10.

i) 00000011_2

ii) 00000100_2

iii) 10000000_2

iv) 10000011_2

v) 10110111_2

vi) 01110011_2

Answers on page 9.13.

If you remain unclear on this, or simply want to see it demonstrated type in the Binary/Hex tutor program and run this. At the MENU select 'H' for "Decimal, Binary and Hexadecimal". Then, when asked "At what No. do you want to start?" press "1" <return> . The screen will then display three rows of boxes, of which the top two are currently of interest. These represent a decimal number (marked "DEC") and a binary number (marked "BIN"). At this stage, they should contain the numbers 1_{10} and 1_2. The decimal number has three bits and, thus a capacity of 999_{10} and the binary, with its eight bits will hold up to 255_{10}.

From this point, the program will simply count - every time you press a key - the space bar is the easiest - both the decimal and binary boxes will index one. Try pressing this once and the boxes should contain a 2_{10} and a 10_2. If you carry on indexing then you will see how binary counts. If you now set the decimal to 15_{10}, the binary should read 1111_2. Now index one further and the 1111_2 will change to 10000_2. One way of looking at this is to lay out the addition:-

```
 1111   A
+   1   B
```

On adding the 1 (B) to 1 (B) this gives '2' i.e. 0, carry 1. This carry then produces another '0' + a carry, and so on.

When the register is full i.e. 11111111_2, the addition of a further 1 will clock the register back to zero and 256 will be lost. However, with the 6502 chip <u>all</u> is not lost as the chip has a carry flag that stores the fact that a carry has occurred. To try this, clock the count up to 255_{10} and observe what happens. If this is too far to go in steps of 1 you may return to the MENU by pressing <return> .

Select H and then start the count at about 250_{10} i.e. enter 250 <return> . It's now not too far to clock up to 255_{10} and 11111111_2.

Now watch the binary as you clock it past 255_{10} to 256_{10}. The binary clock back to zero + the carry bit. This is a handy feature of the 6502 but must not be relied on as more than a temporary store of the carry. It's just as easily reset to zero as it is set to 1!

In order to make sure you really understand binary notation, you may wish to try some of the exercises provided by the BIN/HEX exercises program. Type in and run the program and at the MENU select '2' to try exercises converting decimal numbers into binary and '5' to try converting them the other way. At any stage, a SPACE will delete the last entry and a <return> will send you back to the MENU.

While, the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1_2 and 1_{10} there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255_{10}) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind-bending! Time to think..... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111_2 or 15_{10} and this requires, along with the 0, sixteen different symbols. The ones adopted for this job are:

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

Fig. A1.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NIBBLES. Thus a byte consists of two nibbles, a most significant nibble (MSN)

and a least significant nibble (LSN) - see Fig. A1.3.

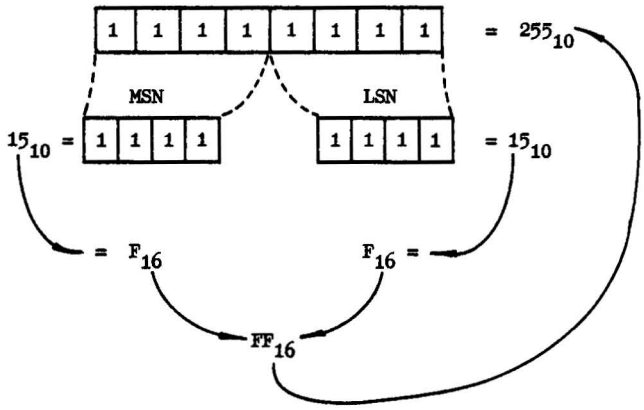


Fig. A1.3

The system described, which uses sixteen symbols is, of course, given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

If you now notice the third box on the screen that you pretended not to notice, you will see that this counts in HEX. The comparability between binary and HEX shows wherever a major carry occurs take for instance 1111_2, 15_{10} or F_{16}: one index past this clocks the binary ones to zeroes and adds a one to the left i.e. to 10000_2 or 10_{16}. These major points of correspondence occur at

$$1_2 = 1_{16} = 1_{10}$$

$$0001\ 0000_2 = 10_{16} = 16_{10}$$

$$0000\ 0001\ 0000\ 0000_2 = 100_{16} = 256_{10}$$

$$0001\ 0000\ 0000\ 0000_2 = 1000_{16} = 4096_{10}$$

As you will have noticed already the hexadecimal count is present on the DEC/BIN/HEX display and the progress of the count can be followed on this.

Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together e.g. FF_{16} = 255, once again calls for place-value notation. This time, as the base is 16 the ratio between any place and its neighbour is 16.

The values, in base 10 of the places in hexadecimal are:

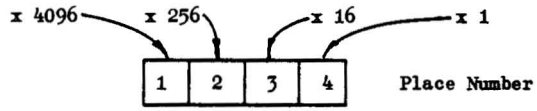


Fig. A1.4

Using fig. A1.4 the way that $E92F_{16}$ makes up 59695_{10} is explained below in figure A1.5.

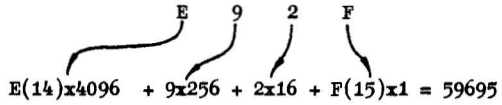


Fig. A1.5

Now that hex is totally mastered! try the following, the first two are explained fully on page 9.11.

Exercise A1.2

Calculate the value in decimal of the following:-

i) 0009_{16} v) $000E_{16}$
ii) 0013_{16} vi) $011A_{16}$
iii) $00A5_{16}$ vii) $00EA_{16}$
iv) $0AAE_{16}$ viii) $F0A3_{16}$

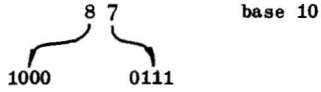
Answers on page 9.13.

If you need more exercises in hex the Bin/Hex exercises program will supply them. Simply select 'E' for exercises and then '1' for decimal to Hex practice and '4' for Hex to decimal. Remember SPACE to delete last entry and <return> for MENU.

BINARY CODED DECIMAL

As well as decimal, binary and hexadecimal notations, one other system is used in computing - binary-coded decimal. As its name suggests it is a hybrid form with elements from binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:



i.e. BCD = 1000 0111 (or in eight bits 10000111)

Fig. A1.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10} = 1001_2$, thus a BCD digit can be represented by a nibble and two digits by a byte. Figure A1.6 shows this, where 87_{10} is represented in BCD as 10000111_2. This can give rise to ambiguity in that 10000111_2 in binary is 135_{10}. To overcome this BCD representations will be given the notation 10000111_2(BCD).

Using four bits of binary, it is possible to count up to 15_{10} (i.e. $1111_2 = 15_{10}$) but in BCD the largest digit used is 9 so, inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_2 and carries the 1 to the next nibble, i.e.

8_{10}	=	0000 1000	(base 2 BCD)
9_{10}	=	0000 1001	" " "
10_{10}	=	0001 0000	" " "
11_{10}	=	0001 0001	" " "

Fig. A1.7

To examine this on the tutor tape load Bin/Hex tutor and select 'B' at the MENU. When asked "At what number....." enter a 1 <return> .

The display will then show three boxes but, this time they will contain decimal, binary and BCD. If you tap in the first 9 counts you will see that up to 9_{10} binary and BCD are identical. However, as you type in an input to change from 9_{10} to 10_{10} keep an eye on the BCD box and you will see the 1 carried over to the most significant nibble. From 10_{10} upwards BCD becomes a true hybrid representing the decimal number in a binary form.

As the number input increases, the uneconomical nature of BCD will become apparent as 99_{10} changes to 100_{10}. (You may type in 99 inputs to get there or press <return> and re-enter at, say, 96). When the input is made that indexes 99_{10} to 100_{10} the BCD generates a carry from its most significant nibble to the carry flag so the 6502 does not lose this but sets the flag.

As mentioned above, this carry is only a short-term expedient and must be picked up at the earliest possible moment if it is not to be lost. The carry is generated on the BCD boxes at 99_{10} while the binary boxes will store up to 256_{2} and the decimal of course up to 999_{10}. BCD is less economical but has its uses in particular situations.

As you know all about BCD now! Try the following:-

Exercise A1.3

Convert the following decimal numbers into BCD:

i)	4	v)	53
ii)	10	vi)	102
iii)	77	vii)	953
iv)	97	viii)	2579

Answers on page 9.14.

Exercise A1.4

Convert the following BCD numbers into decimal:-

i) 0000 0001
ii) 0000 1001
iii) 0001 0101
iv) 0010 0000
v) 0100 1001
vi) 1010 0011
vii) 1001 0111
viii) 1000 1000

Answers on page 9.15.

The Bin/Hex tape will give you any number of further examples that you might need! To get these RUN the exercises program. Then to try Decimal to BCD exercises type in a "3" or to try the reverse process use a "6" for BCD to decimal. Because of the hybrid nature of BCD - one of its relatives is decimal, you will, no doubt, find these conversions easier than the corresponding ones using straight binary numbers.

In the explanations given of the value of places in place value notation a simplification was adopted in order to make these explanations clearer for the less mathematically inclined brethren! However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX 1.

With binary numbers it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact the multiplying factor is the base raised to the power of its place starting with zero at the left. i.e. in binary:

7	6	5	4	3	2	1	0	Place
128	64	32	16	8	4	2	1	Previously stated multiplication factor.
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Mathematically more precise factor.

Thus the least significant bit is multiplied by 2^0 or 1 (If you are not sure of this try the direct program PRINT 2↑0). The next bit is multiplied by 2^1 and so on.

This rule holds for ANY base, let's apply it for hex i.e. base 16:

Least significant bit factor = 16^0 = 1

2nd most significant bit factor = 16^1 = 16

3rd most significant bit factor = 16^2 = 256

Most significant bit factor = 16^3 = 4096

- - - - - - - -

© Copyright
M D HOLMES
1982

```
10 POKE36879,8:PRINT"■"CHR$(28)"    ┌─COPYRIGHT
30 PRINT"    │C│M.D.HOLMES
40 PRINT"    └─    1982
150 PRINTTAB(9)"■■MENU■■■■■┌────■■"
180 PRINTTAB(4)"■DEC┐■■│■■■■■BIN│HB■■■│■■■■■BCD┘TTTTT
190 PRINTTAB(13)"DEC┐■■│■■■■■BIN│HH■■■│■■■■■HEX┘
192 CLR:Q$="┌──■■■■│■■└──┘
194 DIMA$(16):A$="0123456789ABCDEF
196 FORX=0TO15:A$(X)=MID$(A$,X+1,1):NEXT:X=0
270 GETB$:IFB$=""THEN270
280 IFB$="H"ANDA=1THEN370
290 IFB$="B"ANDA=1THEN340
300 IFB$="H"ORB$="B"THEN420
320 GOTO270
330 GOSUB910:GOSUB1510
340 GOSUB1010:GOSUB1620:GOSUB1310:GOSUB1760
350 GOTO390
360 GOSUB910:GOSUB1110
370 GOSUB1010:GOSUB1210:GOSUB1310:GOSUB1410
390 GETA$:IFA$=""THEN390
400 IFA$=CHR$(13)THEN10
410 GOTO280
420 PRINT"■■■■■":X$="255":IFB$="B"THENX$="99"
430 PRINT"N0.FOR START(1TO"X$")?":INPUTZZ
450 IFZZ=0ORZZ=1THEN510
460 IFZZ>VAL(X$)THEN540
480 D=ZZ-1:F=3:GOSUB630:GOSUB2030
490 GOSUB2200:FORX=1TOS:C(X)=B(X):NEXT:GOSUB2110
510 IFB$="B"THEN330
520 IFB$="H"THEN360
530 END
540 PRINT"■SORRY,NUMBER TO LARGE.
550 PRINT"PLEASE RE-ENTER.■":GOTO430
630 D$=STR$(D):D$=RIGHT$(D$,LEN(D$)-1)
650 IFLEN(D$)<FTHEND$="0"+D$:GOTO650
660 FORX=FTO1STEP-1:D(X)=VAL(MID$(D$,X,1))
680 NEXT:RETURN
690 PRINT"■■■■■■■■■■■■■■■■■■■■";Q$
770 FORX=1TO10:POKE8077,131:FORXX=1TO255:NEXT
780 POKE8077,3
790 FORXX=1TO255:NEXT:NEXT:RETURN
800 PRINT"■■■■■■■■■■■■";Q$
880 FORX=1TO10:POKE7923,131:FORXX=1TO255:NEXT
890 POKE7923,3
900 FORXX=1TO255:NEXT:NEXT:RETURN
910 PRINT"■■        DECIMAL■
915 PRINT"        ┌┬┬┬┬":A=1
```

continued.....

```
920 PRINT"        ! ! ! !
930 PRINT"        L_L_L_L
935 PRINT"■■          BINARY
940 PRINT"■  r-T-T-T-T-T-T-T-7
950 PRINT"   ! ! ! ! ! ! ! ! !
960 PRINT"   L_L_L_L_L_L_L_L_J
970 RETURN
1010 FORX=3TO1STEP-1
1020 IFD(X)=9THEND(X)=0:NEXT
1030 IFX=0THEN1050
1040 D(X)=D(X)+1:X=0:NEXT
1050 FORX=8TO1STEP-1
1060 IFB(X)=1THENB(X)=0:NEXT
1070 IFX=0THENRETURN
1080 B(X)=B(X)+1:X=0:NEXT
1090 RETURN
1110 PRINT"■■           HEX
1111 PRINT"        r-T-T-T-7
1120 PRINT"        ! ! ! ! !
1130 PRINT"        L_L_L_L_J
1140 RETURN
1210 FORX=4TO1STEP-1
1220 IFH(X)=15THENH$(X)="0":H(X)=0:NEXT
1230 IFX=0THENRETURN
1240 H(X)=H(X)+1:H$(X)=A$(H(X)+1):X=0:NEXT
1250 RETURN
1310 FORX=1TO3:POKE(7774+(X*2)),D(X)+48:NEXT
1330 FORX=1TO8:POKE7923+X*2,B(X)+48:NEXT
1350 IFD(1)=2ANDD(2)=5ANDD(3)=6THENGOSUB800
1360 RETURN
1410 FORX=1TO4:IFH(X)>9THEN1460
1430 POKE8059+X*2,H(X)+48:NEXT:GOTO1310
1460 POKE8059+X*2,H(X)-9:NEXT
1470 RETURN
1510 PRINT"■■   BINARY CODED DEC■
1520 PRINT"   r-T-T-T-T-T-T-T-T-7
1530 PRINT"   ! ! ! ! ! ! ! ! ! !
1540 PRINT"   L_L_L_L_L_L_L_L_L_J
1550 RETURN
1620 IFC(5)=1ANDC(8)=1THENC(5)=0:C(8)=0:GOTO1680
1630 FORX=8TO4STEP-1
1640 IFC(X)=1THENC(X)=0:NEXT
1650 IFX<5THENRETURN
1660 C(X)=C(X)+1:X=0:NEXT:RETURN
1670 REM GOTO1690
1680 IFC(1)=1ANDC(4)=1THEN1740
1690 FORX=4TO0STEP-1
```

continued.....

```
1700 IFC(X)=1THENC(X)=0:NEXT
1710 IFX=0THENRETURN
1720 C(X)=C(X)+1:X=0:NEXT:RETURN
1740 FORX=1TO8:C(X)=0:NEXT:RETURN
1760 FORX=1TO8:POKE8077+X*2,C(X)+48:NEXT
1780 IFD(1)=1ANDD(2)=0ANDD(3)=0THENGOSUB690
1790 RETURN
2030 H(1)=INT(D/4096)
2040 H(2)=INT((D-H(1)*4096)/256)
2050 H(3)=INT((D-(H(1)*4096)-(H(2)*256))/16)
2060 H(4)=D-(H(1)*4096)-(H(2)*256)-(H(3)*16)
2070 H$(1)=A$(H(1)):H$(2)=A$(H(2))
2080 H$(3)=A$(H(3)):H$(4)=A$(H(4))
2090 H$(5)=H$(1)+H$(2)+H$(3)+H$(4)
2100 RETURN
2110 B(1)=INT(D/128):D1=B(1)*128
2120 B(2)=INT((D-D1)/64):D2=D1+B(2)*64
2130 B(3)=INT((D-D2)/32):D3=D2+B(3)*32
2140 B(4)=INT((D-D3)/16):D4=D3+B(4)*16
2150 B(5)=INT((D-D4)/8):D5=D4+B(5)*8
2160 B(6)=INT((D-D5)/4):D6=D5+B(6)*4
2170 B(7)=INT((D-D6)/2):D7=D6+B(7)*2
2180 B(8)=INT((D-D7)):D8=D7+B(8)
2190 RETURN
2200 D1=INT(D/10):D2=D-D1*10
2220 B(1)=INT(D1/8):E1=B(1)*8
2230 B(2)=INT((D1-E1)/4):E2=E1+B(2)*4
2240 B(3)=INT((D1-E2)/2):E3=E2+B(3)*2
2250 B(4)=INT(D1-E3)
2260 B(5)=INT(D2/8):E5=B(5)*8
2270 B(6)=INT((D2-E5)/4):E6=E5+B(6)*4
2280 B(7)=INT((D2-E6)/2):E7=E6+B(7)*2
2290 B(8)=INT(D2-E7)
2300 RETURN

READY.
```

© Copyright
P HOLMES 1982

Exercises

```
10 POKE36879,8:Q$="┌■■■┐■■└─┘"
20 CLR:PRINTCHR$(28)"■     ┌─COPYRIGHT
21 Q$="┌■■■┐■■└─┘"
30 PRINT"     │C│M.D.HOLMES
40 PRINT"     └─┘ 1982
50 PRINTTAB(8)"■■MENU■■■■■┘"
55 DIMA$(16):A$="0123456789ABCDEF"
56 FORX=0TO15:A$(X)=MID$(A$,X+1,1):NEXT
57 Q$="┌■■■┐■■└─┘"
60 Z$="■■■■■■■■■■■■":GOTO1860
560 PRINTZ$"SORRY,WRONG!
570 PRINT"PLEASE TRY AGAIN
580 FORX=1TO2000:NEXT:Y=0:GOTO1980
590 PRINTZ$"■ELL DONE!THATS RIGHT!
595 FORX=1TO2000:NEXT:GOTO20
600 GOTO1860
610 FORX=1TO2000:NEXT:GOTO1910
630 D$=STR$(D)
640 D$=RIGHT$(D$,LEN(D$)-1)
650 IFLEN(D$)<5THEND$="0"+D$:GOTO650
660 FORX=5TO1STEP-1
670 D(X)=VAL(MID$(D$,X,1))
680 NEXT:RETURN
1860 PRINT"■■■■  FROM   │ TO DEC
1863 PRINT"  DEC TO │ FROM
1866 PRINT"  ──────┼──────
1868 PRINTTAB(9)"│":PRINT"  HEX..1 │■■│■ HEX..4■
1880 PRINT"  BIN..2 │■■│■ BIN..5■
1890 PRINT"  BCD..3 │■■│■ BCD..6■
1900 GETB$:IFB$=""THEN1900
1910 D=VAL(B$):IFD=0THEN1900
1920 IFD>3THENF=5
1930 IFD=1ORD=4THEND=32768:IFF=0THENF=4
1940 IFD=2ORD=5THEND=255:IFF=0THENF=8
1950 IFD=3ORD=6THEND=99:IFF=0THENF=8
1960 IFD<1OTHEN1900
1970 D=INT(D*(RND(TI)))
1980 ONVAL(B$)GOTO4000,4500,5000,5500,6000,6500
1990 GOTO1900
2000 GOSUB2030:GOTO2310
2030 H(1)=INT(D/4096)
2040 H(2)=INT((D-H(1)*4096)/256)
2050 H(3)=INT((D-(H(1)*4096)-(H(2)*256))/16)
2060 H(4)=D-(H(1)*4096)-(H(2)*256)-(H(3)*16)
```

continued

Exercises

```
2070 FORX=1TO4:H$(X)=MID$(A$,H(X)+1,1):NEXT
2090 H$(5)=H$(1)+H$(2)+H$(3)+H$(4)
2100 RETURN
2200 D1=INT(D/10):D2=D-D1*10
2220 B(1)=INT(D1/8):E1=B(1)*8
2230 B(2)=INT((D1-E1)/4):E2=E1+B(2)*4
2240 B(3)=INT((D1-E2)/2):E3=E2+B(3)*2
2250 B(4)=INT(D1-E3)
2260 B(5)=INT(D2/8):E5=B(5)*8
2270 B(6)=INT((D2-E5)/4):E6=E5+B(6)*4
2280 B(7)=INT((D2-E6)/2):E7=E6+B(7)*2
2290 B(8)=INT(D2-E7)
2300 RETURN
2310 GOSUB2320:GOTO2370
2372 FORX=4TO1STEP-1:IFH(X)>9THENH(X)=H(X)-57
2374 POKE7835+X*2,H(X)+48:NEXT:RETURN
2376 FORX=8TO1STEP-1:POKE7835+X*2,B(X)+48
2378 NEXT:RETURN
2380 FORX=5TO1STEP-1:POKE7835+(X*2),D(X)+48
2390 NEXT:RETURN
2400 GOSUB2750
2420 FORX=0TO3:IFC$(X)<>H$(X+1)THEN560
2440 NEXT:GOTO590
2442 FORX=0TO4:IFVAL(C$(X))<>D(X+1)THEN560
2444 NEXT:GOTO590
2460 FORX=1TO8:IFVAL(C$(X-1))<>B(X)THEN560
2470 NEXT:GOTO590
2770 FORX=1TO35:GETC$(Y):IFC$(Y)=""THENNEXT
2780 POKE8013+(Y*2),160:POKE8015+(Y*2),32
2790 IFC$(Y)<>""THEN2840
2800 FORX=1TO35:GETC$(Y):IFC$(Y)=""THENNEXT
2810 IFC$(Y)<>""THEN2840
2820 POKE8013+(Y*2),32:POKE8015+(Y*2),32
2830 GOTO2770
2840 IFC$(Y)=CHR$(13)THEN20
2850 IFC$(Y)=" "ANDY>0THENY=Y-1:GOTO2800
2860 ONVAL(B$)GOTO2900,2930,2930,2670,2870,2870
2870 IFC$(Y)="0"THEN2950
2880 IFVAL(C$(Y))=0THEN2990
2890 GOTO2950
2900 FORX=0TO15:IFC$(Y)=A$(X)THENX=20
2910 NEXT:IFX<20THEN2990
2920 GOTO2950
2930 IFC$(Y)="1"ORC$(Y)="0"THEN2950
2940 GOTO2990
2950 Y=Y+1:PRINT"[RVS]                    [OFF]";
2960 FORX=1TOY*2:PRINT"[RVS] [OFF]";:NEXT:PRINTC$(Y-1)
2970 IFY=FTHENRETURN
2980 L=8077+(Y*2)
```

continued.....

Exercises

```
2990 IFPEEK(L)>127THENPOKEL,PEEK(L)-128
3000 GOTO2770
3001 PRINT"
3002 PRINT"
3041 PRINTTAB(10)"
3042 RETURN
3050 PRINT"  ";Q$"
3080 RETURN:END
3090 GOSUB2330:RETURN
3100 PRINT"  ";Q$"
3130 RETURN
3140 N$(1)="HEX":N$(2)="BIN":N$(3)="BCD"
3150 IFVAL(B$)<4THENPRINT"
3160 PRINT"
4000 GOSUB2030:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3050
4001 GOSUB3041:GOSUB3002:GOSUB3050:GOSUB2390:GOSUB2770:GOTO2420
4500 GOSUB10000:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3050:GOSUB3041
4510 GOSUB3002:GOSUB3100:GOSUB2380:GOSUB2770:GOTO2460
5000 GOSUB2200:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3050:GOSUB3041
5010 GOSUB3002:GOSUB3100:GOSUB2380:GOSUB2770:GOTO2460
5500 GOSUB2030:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3050
5510 GOSUB3002:GOSUB3050:GOSUB3041:GOSUB2372:GOSUB2770:GOTO2442
6000 GOSUB10000:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3100
6010 GOSUB3002:GOSUB3050:GOSUB3041:GOSUB2376:GOSUB2770:GOTO2442
6500 GOSUB2200:GOSUB630:GOSUB3140:GOSUB3001:GOSUB3100:GOSUB3002
6510 GOSUB3050:GOSUB3041:GOSUB2376:GOSUB2770:GOTO2442
10000 DD(0)=0:FORX=1TO8:DI=2↑(8-X)
10001 TP=D-DD(X-1)
10005 B(X)=INT(TP/DI)
10010 DD(X)=DD(X-1)+B(X)*(2↑(8-X)):NEXT
10020 RETURN
```

APPENDIX II

TABLE 1

The 6502 Instruction set with Mnemonics used in this Book

Abbreviations used in this table:

A	=	Accumulator	S	=	Set (to 1)
X	=	X register	C	=	Clear (to 0)
Y	=	Y register	?	=	Condition according to data.

ADC ADd specified contents to accumulator with Carry: store answer in accumulator; Condition negative, overflow zero and carry flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ADC	109	6D	2	4	Absolute
ADCIM	105	69	1	2	Immediate
ADCIX	97	61	1	6	Indirect with X
ADCIY	113	71	1	5*	Indirect with Y
ADCX	125	7D	2	4*	Indexed with X
ADCY	121	79	2	4*	Indexed with Y
ADCZ	101	65	1	3	Zero-page
ADCZX	117	75	1	4	Zero-page indexed with X

*Plus 1 cycle if page boundary crossed.
To add without carry, clear carry flag (CLC) before ADC.
ADC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

AND — AND specified contents with accumulator: store answer in accumulator; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
AND	45	2D	2	4	Absolute
ANDIM	41	29	1	2	Immediate
ANDIX	33	21	1	6	Indirect with X
ANDIY	49	31	1	5*	Indirect with Y
ANDX	61	3D	2	4*	Indexed with X
ANDY	57	39	2	4*	Indexed with Y
ANDZ	37	25	1	3	Zero-page
ANDZX	53	35	1	4	Zero-page indexed with X

* Plus 1 cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	0
1	0	1

ASL — Arithmetic Shift Left of specified contents: bit 7 put into carry, '0' into bit zero; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ASL	14	0E	2	6	Absolute
ASLA	10	0A	0	2	Accumulator
ASLX	30	1E	2	7	Indexed with X
ASLZ	6	06	1	5	Zero-page
ASLZX	22	16	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

BCC Branch on Carry Clear:
Test carry flag, if clear (C=0) branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BCC	144	90	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: No Effect.

BCS Branch on Carry Set: test carry flag, if set (C=1), branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BCS	176	B0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BEQ Branch on result EQual to zero: test Z-flag if set, (Z=1) branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BEQ	240	F0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BIT AND specified BITs with accumulator: A remains unaltered; set Z (=1) if bits match, transfer bits 6 and 7 of specified data to V and N flags respectively; condition zero flag according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER	NO. CYCLES	ADDRESSING MODE
BIT	44	2C	2	4	Absolute
BITZ	36	24	1	3	Zero-page

N	V	B	D	I	Z	C
Bit 7	Bit 6	-	-	-	?	-

BMI Branch on result MInus: test N flag if set (N=1), branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BMI	48	30	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BNE Branch on result Not Equal to zero: test Z flag if not set (Z=0), branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BNE	208	D0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BPL Branch on result PLus: test N flag, if not set (N=0), branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BPL	16	10	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BRK BReaK into interrupt: initiate interrupt sequence; Save PC+2 on stack; set B flag (B=1); save PSW on stack; load interrupt vectors (FFFE and FFFF) into PC. Set I flag (I=1)

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BRK	0	0	0	7	Implied

N	V	B	D	I	Z	C
-	-	S	-	S	-	-

BVC Branch on oVerflow Clear: test overflow flag, if not set (V=0), branch relative.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
BVC	80	50	1	2*	Relative

*Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

BVS Branch on oVerflow Set: test overflow flag, if set (V=1), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BVS	112	70	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.

FLAGS: No Effect.

CLC CLear Carry flag: load '0' into carry flag (C=0).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLC	24	18	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	C

CLD CLear Decimal flag: load '0' into decimal flag.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLD	216	D8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	C	-	-	-

CLI CLear Interrupt disable flag: load '0' into interrupt flag (I=0).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLI	88	58	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	C	-	-

CLV CLear oVerflow flag: load '0' into overflow flag (V=0)

MNEM	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLV	184	B8	0	2	Implied

N	V	B	D	I	Z	C
-	0	-	-	-	-	-

CMP CoMPare specified data with accumulator: subtract data from accumulator, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
CMP	205	CD	2	4	Absolute
CMPIM	201	C9	1	2	Immediate
CMPIX	193	C1	1	6	Indirect with X
CMPIY	209	D1	1	5*	Indirect with Y
CMPX	221	DD	2	4*	Indexed with X
CMPY	217	D9	2	4*	Indexed with Y
CMPZ	197	C5	1	3	Zero-page
CMPZX	213	D5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPX ComPare specified data with X register: subtract data from X register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
CPX	236	EC	2	4	Absolute
CPXIM	224	E0	1	2	Immediate
CPXZ	228	E4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPY ComPare specified data with Y-register: subtract data from Y-register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
CPY	204	CC	2	4	Absolute
CPYIM	192	C0	1	2	Immediate
CPYZ	196	C4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

DEC DECrement specified memory contents by one, store result in specified memory location; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
DEC	206	CE	2	6	Absolute
DECX	222	DE	2	7	Indexed with X
DECZ	198	C6	1	5	Zero-page
DECZX	214	D6	1	6	Zero-page,indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEX DEcrement contents of X-register: store result in X-register; condition Negative and Zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEX	202	CA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEY DEcrement contents of Y-register: store result in Y-register; condition Negative and Zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEY	136	88	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

EOR Perform Exclusive OR between accumulator and specified contents: store result in accumulator. Condition Negative and Zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
EOR	77	4D	2	4	Absolute
EORIM	73	49	1	2	Immediate
EORIX	65	41	1	6	Indirect with X
EORIY	81	51	1	5*	Indirect with Y
EORX	93	5D	2	4*	Indexed with X
EORY	89	59	2	4*	Indexed with Y
EORZ	69	45	1	3	Zero-page
EORZX	85	55	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	1
1	1	0

INC INCrement specified contents by one: store result in specified location; condition Negative and Zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
INC	238	EE	2	6	Absolute
INCX	254	FE	2	7	Indexed with X
INCZ	230	E6	1	5	Zero-page
INCZX	246	F6	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INX — INcrement X-register by one: store result in X-register; condition Negative and Zero Flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
INX	232	E8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INY — INcrement Y-register by one: store result in Y-register; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
INY	200	C8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

JMP — JuMP to specified address (load specified address into Program counter).

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
JMP	76	4C	2	3	Absolute
JMPIA	108	6C	2	5	Indirect

FLAGS: No Effect.

JSR — Jump to SubRoutine at specified address: store program counter contents +2 on stack; load specified contents into program counter.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
JSR	32	20	2	6	Absolute

FLAGS: No Effect.

LDA — LoaD Accumulator with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
LDA	173	AD	2	4	Absolute
LDAIM	169	A9	1	2	Immediate
LDAIX	161	A1	1	6	Indirect with X
LDAIY	177	B1	1	5*	Indirect with Y
LDAX	189	BD	2	4*	Indexed with X
LDAY	185	B9	2	4*	Indexed with Y
LDAZ	165	A5	1	3	Zero-page
LDAZX	181	B5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDX LoaD X-register with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
LDX	174	AE	2	4	Absolute
LDXIM	162	A2	1	2	Immediate
LDXY	190	BE	2	4*	Indexed with Y
LDXZ	166	A6	1	3	Zero-page
LDXZY	182	B6	1	4	Zero-page indexed with Y

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDY LoaD Y-register with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
LDY	172	AC	2	4	Absolute
LDYIM	160	A0	1	2	Immediate
LDYX	188	BC	2	4*	Indexed with X
LDYZ	164	A4	1	3	Zero-page
LDYZX	180	B4	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LSR — Perform Logical Shift Right of specified contents: load bit 0 into carry bit and a '0' into bit 7; condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
LSR	78	4E	2	3	Absolute
LSRA	74	4A	0	2	Accumulator
LSRX	94	5E	2	3	Indexed with X
LSRZ	70	46	1	6	Zero-page
LSRZX	86	56	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

NOP — No OPeration; wait two cycles then continue.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
NOP	234	EA	0	2	Implied

FLAGS: No Effect.

ORA — Perform Logical OR between Accumulator and specified contents: store result in accumulator; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ORA	13	0D	2	4	Absolute
ORAIM	9	09	1	2	Immediate
ORAIX	1	01	1	6	Indirect with X
ORAIY	17	11	1	5*	Indirect with Y
ORAX	29	1D	2	4*	Indexed with X
ORAY	25	19	2	4*	Indexed with Y
ORAZ	5	05	1	3	Zero-page
ORAZX	21	15	1	4	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	1
1	1	1

PHA PusH Accumulator onto stack: update stack pointer; A remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHA	72	48	0	3	Implied

FLAGS: No Effect.

PHP PusH Processor status word onto stack: update stack pointer; PSW remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHP	8	08	0	3	Implied

FLAGS: No Effect.

PLA PuLl Accumulator from stack:update stack pointer; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PLA	104	68	0	4	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

PLP — PuLl Processor status word from stack: update stack pointer; condition ALL flags according to PSW retrieved.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
PLP	40	28	0	4	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

ROL — ROtate Left one place, specified contents: load carry bit into bit 0 of specified data and bit 7 of these into carry flag; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ROL	46	2E	2	6	Absolute
ROLA	42	2A	0	2	Accumulator
ROLX	62	3E	2	7	Indexed with X
ROLZ	38	26	1	5	Zero-page
ROLZX	54	36	1	6	Zero-page indexed with X

ROL is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 7

ROR ROtate Right one place, specified contents: load carry bit into bit 7 of specified data and bit 0 of these into carry flag; condition negative and zero flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ROR	110	6E	2	6	Absolute
RORA	106	6A	0	2	Accumulator
RORX	126	7E	2	7	Indexed with X
RORZ	102	66	1	5	Zero-page
RORZX	118	76	1	6	Zero-page indexed with X

ROR is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 0

RTI ReTurn from Interrupt: retrieve PSW and PC from stack, update stack pointer.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
RTI	64	40	0	6	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

RTS ReTurn from Subroutine: retrieve PC from stack and increment by one, update stack pointer.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
RTS	96	60	0	6	Implied

FLAGS: No Effect.

SBC SuBtract with Carry specified contents from accumulator: store answer in accumulator; condition negative, overflow, zero and carry flags according to result.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODES
SBC	237	ED	2	4	Absolute
SBCIM	233	E9	1	2	Immediate
SBCIX	225	E1	1	6	Indirect with X
SBCIY	241	F1	1	5*	Indirect with Y
SBCX	253	FD	2	4*	Indexed with X
SBCY	249	F9	2	4*	Indexed with Y
SBCZ	229	E5	1	3	Zero-page
SBCZX	245	F5	1	4	Zero-page indexed with X

*Plus one cycle if page boundary crossed.
To subtract without carry, set carry flag (SEC) before SBC.
SBC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

SEC SEt Carry flag: C=1

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
SEC	56	38	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	S

SED SEt Decimal flag: D = 1.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SED	248	F8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	S	-	-	-

SEI SEt Interrupt disable flag: I = 1.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SEI	120	76	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	S	-	-

STA STore Accumulator contents at specified address: A remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STA	141	8D	2	4	Absolute
STAIX	129	81	1	6	Indirect with X
STAIY	145	91	1	6	Indirect with Y
STAX	157	9D	2	5	Indexed with X
STAY	153	99	2	5	Indexed with Y
STAZ	133	85	1	3	Zero-page
STAZX	149	95	1	4	Zero-page indexed with X

FLAGS: No Effect.

STX STore contents of X-register at specified address: X remains unaltered.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
STX	142	8E	2	4	Absolute
STXZ	134	86	1	3	Zero-page
STXZY	150	96	1	4	Zero-page indexed with Y

FLAGS: No Effect.

STY STore contents of Y-register at specified address: Y remains unaltered.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
STY	140	8C	2	4	Absolute
STYZ	132	84	1	3	Zero-page
STYZX	148	94	1	4	Zero-page indexed with X

FLAGS: No Effect.

TAX Transfer contents of Accumulator to X-register: A remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TAX	170	AA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TAY Transfer contents of Accumulator to Y-register: A remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TAY	168	A8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TSX Transfer contents of Stack-pointer to X-register: SP remains unaltered; condition negative and zero flags according to data.

* TSX Not available on VIC-20.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TSX	186	BA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXA Transfer contents of X-register into Accumulator: X remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TXA	138	8A	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXS Transfer contents of X-register into Stack: X remains unaltered;

* TXS not available on VIC-20.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TXS	154	9A	0	2	Implied.

FLAGS: No Effect.

TYA Transfer contents of Y-register into Accumulator: Y remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
TYA	152	98	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TABLE 2. VIC Character Set

@	0	4	52		103		154		204
A	1	5	53		104		155		205
B	2	6	54		105		156		206
C	3	7	55		106		157		207
D	4	8	56		107		158		208
E	5	9	57		108		159		209
F	6	:	58		109		160		210
G	7	;	59		110		161		211
H	8	<	60		111		162		212
I	9	=	61		112		163		213
J	10	>	62		113		164		214
K	11	?	63		114		165		215
L	12	—	64		115		166		216
M	13	♠	65		116		167		217
N	14			66		117		168		218
O	15	—	67		118		169		219
P	16	—	68		119		170		220
Q	17	—	69		120		171		221
R	18	—	70		121		172		222
S	19			71		122		173		223
T	20			72		123		174		224
U	21	╮	73		124		175		225
V	22	╲	74		125		176		226
W	23	╯	75		126		177		227
X	24	╰	76		127		178		228
Y	25	╲	77		128		179		229
Z	26	╱	78		129		180		230
[	27	╭	79		130		181		231
£	28	╮	80		131		182		232
]	29	●	81		132		183		233
↑	30	—	82		133		184		234
←	31	♥	83		134		185		235
	32			84		135		186		236
!	33	╭	85		136		187		237
"	34	╳	86		137		188		238
#	35	○	87		138		189		239
$	36	♣	88		139		190		240
%	37			89		140		191		241
&	38	♦	90		141		192		242
'	39	+	91		142		193		243
(	40	▒	92		143		194		244
)	41			93		144		195		245
*	42	π	94		145		196		246
+	43	◥	95		146		197		247
,	44		96		147		198		248
-	45	▮	97		148		199		249
.	46	▬	98		149		200		250
/	47	▔	99		150		201		251
0	48	_	100		151		202		252
1	49			101		152		203		253
2	50	▒	102		153				254
3	51								255

TABLE 3

Hexadecimal to Decimal Conversion Chart

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

TABLE 4

ASCII CHARACTER SET

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

TABLE 5

6502 FLAG GUIDE

This guide lists all the 6502 flags along with the families of instructions that set them and the branch instructions that test their condition. As all members of the family condition the flags in the same way the whole family is referenced by the first three letters of the mnemonic. i.e. ADC refers to the Add with Carry family or ADC, ADCIM, ADCIX, ADCIY, ADCX, ADCY, ADCZ and ADCZX.

N - NEGATIVE FLAG

Instruction to condition				Instruction to test
ADC	DEY	LSR	TAX	BMI
AND	EOR	ORA	TAY	BPL
ASL	INC	PLA	TSX	
CMP	INX	PLP	TXA	
CPX	INY	ROL	TYA	
CPY	LDA	ROR		
DEC	LDX	RTI		
DEX	LDY	SBC		

With BIT instruction, bit 7 of data is transferred into N-flag.

V - OVERFLOW FLAG

Instruction to condition			Instruction to test
ADC	CLV	RTI	BVC
BIT	PLP	SBC	BVS

BIT loads V with bit 6 of the specified memory location.

B - BREAK FLAG

Instruction to condition			Instruction to test
BRK	PLP	RTI	

D - DECIMAL FLAG

Instruction to condition				Instruction to test
CLD	PLP	RTI	SED	

I - INTERRUPT FLAG

Instruction to condition				Instruction to test
BRK	CLI	PLP	RTI	
SEI				

Z - ZERO FLAG

Instruction to condition				Instruction to test
ADC	DEC	LDA	ROL	BEQ
AND	DEX	LDX	ROR	BNE
ASL	DEY	LDY	RTI	
BIT	EOR	LSR	SBC	
CMP	INC	ORA	TAX	
CPY	INX	PLA	TAY	
CPX	INY	PLP	TXA	
			TYA	

C - CARRY FLAG

Instruction to condition				Instruction to test
ADC	CPX	ROL	SEC	BCC
ASL	CPY	ROR		BCS
CLC	LSR	RTI		
CMP	PLP	SBI		

APPENDIX III

START ADDRESSES OF BUILT-IN SUB-ROUTINES USING FLOATING POINT NUMBERS

ADDRESS		
Decimal	Hexadecimal	
54161	D391	Convert integer into floating point put answer in FAC. Integer found in Y (LSB) and A (MSB) program 8.9.
55379	D853	Subtract FAC from AFAC. Put answer into FAC. Program 8.19.
55402	D86A	Add together numbers in FAC and AFAC. Put answer in FAC. Program 8.18.
55786	D9EA	Calculate loge of FAC. Put answer in FAC. Program 8.22.
55851	DA2B	Multiply FAC by AFAC, puts answer into FAC. Program in Exercise 8.2.
55948	DA8C	Copy FP data from memory to AFAC. Data address is in A (LSB) and Y (MSB). Program 8.18.
56082	DB12	Divide AFAC by FAC. Put answer into FAC. Program 8.20 .
56226	DBA2	Copy FP data from memory to FAC. Data address in A (LSB) and Y (MSB). Program 8.11.
56263	DBC7	Store FAC in memory. Address of first byte stored in 70 (LSB) and 71 (MSB). Program 8.15.
56316	DBFC	Copy AFAC into FAC. Program 8.13.
56332	DC0C	Copy FAC into AFAC. Program as 8.13.
57201	DF71	Evaluates square root of FAC, puts answer into FAC. Program in Exercise 8.2.
57211	DF7B	Raise AFAC to power of FAC. Put answer into FAC. Program as 8.22/3.
57325	DFED	Evaluate e raised to the power FAC. Put answer into FAC. Program as 8.22/3.

ADDRESS		
Decimal	Hexadecimal	
57953	E261	Calculate COS of FAC (expressed as radians). Put result in FAC. Program as 8.22/3.
57960	E268	Calculate SIN of FAC (expressed as radians). Put result in FAC. Program as 8.22/3.
58033	E2B1	Calculate TAN of FAC (expressed as radians). Put result in FAC. Program as 8.22/3.

APPENDIX IV

VIC MEMORY ALLOCATION

On a basic unexpanded VIC-20 memory locations from 4069 to 7679 are available for BASIC programs. This is the area used by the Assemblers and BIN/BCD/HEX tutor and exercises in the book. On this machine the screen is located from 7680 to 8186 and this is the range of locations used for screen displays in this book.

When a 3K expansion cartridge is added, the available memory for programs expands downwards to 1024, giving a memory availability of 1024 to 7679. With this configuration, the screen remains at 7680 onwards as with the unmodified VIC.

If the VIC-20 is expanded beyond 6.5K, i.e. by means of, for example, a 16K RAM expansion cartridge, the screen memory is moved from7680 onwards to 4096 onwards. Thus, when this book is used with an expanded machine the 7680 screen locations must be replaced by 4096 screen locations i.e. 3584 should be subtracted from these. In addition the colour-code memory map is relocated and, instead of starting at 38400 (9600_{16}) it starts at 37888 (9400_{16}).

INDEX

Absolute addressing 3.7E
Accumulator 1.2E, 1.7
ADC 1.5E, 1.6P
ADCIX 3.12
ADCIY 3.12
ADCZ 3.8
Addressing 3.6ff
Addition 4.1ff
ALU 1.7
AND 4.12E, 4.15P
ANDZ 3.8
Architecture of 6502 1.7
ASL 4.20E
ASLA 4.21E, 4.21P
ASLZ 3.8
Assembler, debugging (ii)
Assembler, entering program 1.2
Assembly from MONITOR 6.13
BCC 4.1E, 4.2P
BCD arithmetic 4.10
BCD notation A1.7
BCS 4.2E, 4.2P
BEQ 2.7E, 2.7P
Bflag 2.13E, 2.14E
Binary-coded decimal A1.7
Binary-coded decimal arithmetic 4.10
Binary notation A1.1ff
Binary multiplication 4.21
BIT 4.26E
BIT manipulation 4.18ff
BITZ 3.8, 4.26E
BMI 2.15E, 2.15P
BNE 2.8E 2.8P
BPL 2.15E, 2.16P
Branches 2.6ff
BRK 8.2
BVC 8.6E,P
BVS 8.6
Carry flag 2.13E, 2.14E, 4.3
CLC 3.7, 4.1E, 4.2P
CLD 4.11E, 4.11P
CLI 3.7, 8.1E
CLV 3.7, 8.6E
CMP 2.11E, 2.11P
CMPIX 3.12
CMPIY 3.12
CMPZ 3.8
Colour 6.7E, 6.8P
Conditional Jumps 2.6E
CPX 2.9E, 2.10P
CPXZ 3.8
CPY 2.11E
CPYIM 3.9E, 3.9P
CPYZ 3.8
Crashes 1.9
Data Bus 1.7
Delays 3.5, 6.12
DECZ 3.8
DEX 2.6E
DEY 2.8E, 2.12
Disassembly 5.6
Division 4.27
D flag 2.13E, 2.14
Double precision 4.1ff
EOR 4.17E
EORIM 4.18
EORIX 3.12
EORIY 3.12
EORZ 3.8
Error Messages (iii)
Flags 2.13ff
Floating point Accumulator 8.9
Floating point numbers 8.9
Floating point subroutines A3.1
Hexadecimal Inputs 4.6E
Hexadecimal Notation A1.4
I flag 2.13E
Immediate addressing 3.9E
Implied addressing 3.7E
INC 4.9E
INCZ 3.8
Indexed addressing 3.1, 3.9E
Index Register 2.8
Indirect Absolute Addressing 3.13
Indirect Addressing 3.11E
INX 2.8E
INY 2.8E
JMP 2.1E, 2.1P, 2.10
JMPIA 3.13E, P
JSR 2.4E, 2.5P
Jump instructions 2.1
Labels 5.1
LDAIM 1.6E, P

E = Explained
P = First Program

LDAIX 3.11E, 3.12P
LDAIY 3.12
LDAZ 3.8
LDX 1.8E
LDXZ 3.8
LDY 1.10E
LDYIM 1.10E
LDYZ 3.8
Logical operators 4.12ff
LSR 4.19
LSRA 4.19E, 4.19P
LSRZ 3.8
Machine code 1.1
Macros 5.11,
Memory Labels 5.7E,P
Monitor 6.3
Multiplication 4.8
N flag 2.13E, 2.14
Nibble 4.12, A1.4
NOP 3.7, 4.4E, 4.5P
Numerical screen output 8.7
One's compliment 8.5
Op-code (ii)
Operand 2.3
ORA 4.16E
ORAIM 4.17P
ORAIX 3.12
ORAIY 3.12
ORAZ 3.8
PHA 7.10P
PHP 7.11E
PLA 7.10E,P
PLP 7.11E
Program Counter 2.5
Protecting machine code
in memory 6.6
Pseudo code 1.2
PSW 2.13E
Relative addressing 3.10E
ROL 4.25
ROLA 4.26E
ROLZ 3.8
ROR 4.26E
RORA 4.26E
RORZ 3.8
RTI 8.2E
RTS 1.3E, 1.6P
SBC 4.7E
SBCIM 4.7
SBCIX 3.12
SBCIY 3.12
SBCZ 3.8
SEC 3.7, 4.7P,E
SED 3.7, 4.10E, 4.11P
SEI 3.7, 8.1E
Signed numbers 8.3ff
Sound 6.10
Source code (ii)
STA 1.2E, 1.3P, 1.6, 2.5
STACK 7.7E
STAIX 3.12
STAIY 3.12
STAY 3.9
Status Flags 2.6
STAX 3.1E, P
STAY 3.1E, 3.6P
STAZ 3.8
STX 1.8E, 2.10P
STXZ 3.8
STY 3.3P
STYZ 3.8
Subroutines; floating point
8.17ff, A3.1
Subtraction 4.7
TAX 1.12E, P
TAY 1.13E
Timing 3.4
Transfer instructions 1.2
Truth table 4.13E, 4.16, 4.17
Two's compliment 8.4
TXA 1.13E
TYA 1.13E
USR command 8.12
V flag 2.13E, 2.14
X-register 1.7, 3.11
Y-register 1.7, 3.12
Zero page addressing 3.7, 3.8E
Z flag 2.6E, 2.13, 2.14E

NOTES

NOTES

NOTES

- Complete beginner's book for Assembly Language on the VIC - 20
- Contains complete listings of:
  - ▶ Two - pass 6502 Assembler
  - ▶ Machine - code Monitor
  - ▶ Binary / BCD / Hex tutor
  - ▶ Base conversion exercises
- Dozens of sample programs - fully illustrated