# FIRST STEPS IN MACHINE CODE ON YOUR C64

Ross Symons

Ross Symons is a 16-year-old student, who lives in Tyabb, about 15 miles from Melbourne, in Australia. He is a keen sportsman, and spends most of his leisure time when not programming his computers (he has three) on the playing field.

# First Steps in Machine Code on the Commodore 64

Ross Symons

CORGI ADDISON-WESLEY

First Steps in Machine Code on the Commodore 64

A CORGI/ADDISON-WESLEY BOOK 0 552 99128 7

First publication in Great Britain

PRINTING HISTORY
Corgi/Addison-Wesley edition published 1984


Designed by Brian Shorrock



This book is set by 11/12 Mallard

Corgi Books are published by Transworld Publishers Ltd., Century House, 61–63 Uxbridge Road, Ealing, London W5 5SA

Made and printed in West Germany by Mohndruck, Gütersloh

# *Contents*

# *Introduction*

Welcome to the world of machine code on the Commodore 64. All you need is this book and your trusty computer, and you're on your way.

I've written this book to take you from where you are now – with a knowledge of BASIC, but little or none of machine code programming – to the point where you'll have a good knowledge of the fundamentals of machine code on the Commodore 64. I've gone through all the instructions – one by one – and included a host of sample programs to show them in use.

Machine code is not an easy subject to master. You'll have to concentrate, and work slowly through the book. But, if you do, I assure you that by the end you'll have a good knowledge of the fundamental building blocks of Commodore 64 machine code. Then, it is up to you to put these blocks together to create dazzling programs of your own.

To show you how effective these can be, I've included two complete games which are a mixture of machine code and BASIC. The first one is called PUB SQUASH, and the second is a racing car game. Both of these ran so fast when they were first written, I had to add delay loops to slow them down enough for you to see them! That fact alone illustrates one of the great attractions of working in machine code.

Don't try to hurry through this book. You are acquiring a skill which will bring you a lot of satisfaction in the coming years, so it is worth getting it right from the beginning. Enter *every* program as you come to it, and make sure you understand each program before moving on to the next one. I assure you the effort will be worthwhile.

I'd like to thank Tim Hartnell for the assistance he gave me while I was writing this book.

Time to get underway,

Ross Symons,
Tyabb, Victoria, Australia.
1984

# CHAPTER ONE — HEXADECIMAL AND BINARY NOTATION

In assembly language there are two *different* forms of numbers apart from decimal. These number systems are called 'binary' and 'hexadecimal' notation.

## Hexadecimal Notation

Hexadecimal numbers are numbers based on sixteen, just as decimal numbers are based on ten. Hexadecimal digits range from a value of 0 to 15, and use the figures 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

Here is a list of hexadecimal numbers and their decimal equivalents:

| Hexadecimal | | Decimal Equivalent |
|---|---|---|
| 1 | — | 1 |
| 2 | — | 2 |
| 3 | — | 3 |
| 4 | — | 4 |
| 5 | — | 5 |

| | | |
|---|---|---|
| 6 | — | 6 |
| 7 | — | 7 |
| 8 | — | 8 |
| 9 | — | 9 |
| A | — | 10 |
| B | — | 11 |
| C | — | 12 |
| D | — | 13 |
| E | — | 14 |
| F | — | 15 |

The table is suitable when you're only dealing in single digits, like 9. It cannot be used for digits such as AF or A9E. To find out the decimal equivalent of multiple digit numbers we must examine how they are constructed. For this example, we will use the decimal number 6754. The following will show how this number is constructed:

$6754 = 6*10^3+7*10^2+5*10^1+4*10^0$

In the same way as above we can see how a hexadecimal number is made up. We will use the number A045 (hexadecimal):

$A045 = A*16^3+0*16^2+4*16^1+5*16^0 =$
$10*4096+0*256+4*16+5*1 = 41029$

As shown above, powers of digits in a number increase as you go to the left.

Over 90% of the numbers in this book are hexadecimal, and therefore it is advisable that you learn to use hexadecimal numbers. However, if you're lazy, you can use the program in the next chapter which will convert numbers from decimal to hex (abbreviation for hexadecimal) and from hex to decimal.

# Binary Notation

You may have come across binary numbers before if you have created sprites on your computer. Binary digits are based on two, and are either 0 or 1. Here is a table with binary numbers and their decimal and hex equivalents:

| Binary | | Decimal | | Hexa-decimal |
|---|---|---|---|---|
| 1 | — | 1 | — | 1 |
| 10 | — | 2 | — | 2 |
| 11 | — | 3 | — | 3 |
| 100 | — | 4 | — | 4 |
| 101 | — | 5 | — | 5 |
| 110 | — | 6 | — | 6 |
| 111 | — | 7 | — | 7 |
| 1000 | — | 8 | — | 8 |
| 1001 | — | 9 | — | 9 |
| 1010 | — | 10 | — | A |

| | | | | |
|---|---|---|---|---|
| 1011 | — | 11 | — | B |
| 1100 | — | 12 | — | C |
| 1101 | — | 13 | — | D |
| 1110 | — | 14 | — | E |
| 1111 | — | 15 | — | F |

The above table, like the earlier one, can be only used for simple numbers. To convert more complex numbers we must examine how binary numbers are formed. For the following example we will use binary 10110010:

$$
\begin{aligned}
10110010 &= 1*2^7+0*2^6+1*2^5+1*2^4+0* \\
&\quad\ 2^3+0*2^2+1*2^1+2*0^0 \\
&= 1*128+0*64+1*32+1*16+0*8+0* \\
&\quad\ 4+1*2+0*1 \\
&= 128+0+32+16+0+0+2+0 \\
&= 178
\end{aligned}
$$

The above method is time-consuming and so is not the best way to convert binary to decimal. However, the example does show you how binary numbers are formed. If you have designed your own sprites, you will be familiar with this method:

```
128 64 32 16 8 4 2 1
 1   0  1  1 0 0 1 0 = 128+32+16+2
                      = 178
```

To work in this way, you first write down the table (*128 64* etc.) and write your binary number directly under it, as in the above example. After this, write down every number in the table which has a 1 below it. Then all you do is add the numbers you have written down. The result is the decimal equivalent of the binary number.

# CHAPTER TWO — ASSEMBLER/ DISASSEMBLER

BASIC is not the true language of the Commodore 64, although the computer enables you to use BASIC by interpreting it into assembly language.

To write directly in assembly language we must use an interpreter. The interpreter will change the assembly language that you write into numbers that the computer can understand.

There is an interpreter in this chapter. It's called an 'assembler', because it assembles assembly language into numbers (machine code). The same program has a disassembler option which will convert machine code back to assembly language.

Most of this book was written with this assembler and before you continue with the following chapters it is necessary to type the following program into your machine and save it.

```
1 OPEN5Ø,Ø
1Ø PRINT"{CLR}ASSEMBLER/DISASSEMBLER":PR
INT"BY ROSS SYMONS,1984"
2Ø PRINT"{CUR DN}1.ASSEMBLE CODE"
3Ø PRINT"{CUR DN}2.DISASSEMBLE CODE"
4Ø PRINT"{CUR DN}3.CHR$ INTERPRETATION"
5Ø PRINT"{CUR DN}4.DATA INTERPRETATION"
6Ø PRINT"{CUR DN}5.SAVE MEMORY"
7Ø PRINT"{CUR DN}6.LOAD MEMORY"
8Ø PRINT"{CUR DN}7.EXECUTE CODE"
9Ø PRINT"{CUR DN}8.HEX-DECIMAL CCONVERSI
ON"
1ØØ POKE198,Ø:WAIT198,1:GETA$
11Ø A=ASC(A$)-48:IFA<ØORA>8THEN1ØØ
12Ø PRINT:ON A GOSUB 1ØØØ,4ØØØ,5ØØØ,6ØØØ
,7ØØØ,76ØØ,8ØØØ,9ØØØ
13Ø GOTO1Ø
1ØØØ INPUT"START OF CODE ";P$:IFLEN(P$)<
>4THEN1ØØØ
1Ø1Ø HEX$=LEFT$(P$,2):GOSUB2ØØØ:P=DEC*25
6
1Ø15 HEX$=RIGHT$(P$,2):GOSUB2ØØØ:P=P+DEC
1Ø2Ø PRINTP$;" ";:INPUT#5Ø,C$:PRINT:L=LE
N(C$):IFL=ØTHEN1Ø2Ø
1Ø22 IFC$="X"THENRETURN
1Ø25 IF L=3THENB1=1:C3$=C$:OP=-1:GOTO123
Ø
1Ø3Ø FOR R=1TOLEN(C$)
1Ø4Ø IFMID$(C$,R,1)="$"THENC1$=LEFT$(C$,
R-1):R=LEN(C$)
1Ø5Ø NEXT
1Ø6Ø FOR R=LEN(C$)TO1STEP-1
1Ø7Ø C2$=MID$(C$,R,1):IFC2$<"G"ANDC2$=>"
Ø"THENC2$=RIGHT$(C$,LEN(C$)-R):R=1
1Ø8Ø NEXT
```

```
1090 C3$=C1$+C2$:C4$=MID$(C$,LEN(C1$)+2,
LEN(C$)-LEN(C3$)-1)
1095 IFLEN(C4$)<>2ANDLEN(C4$)<>4THENPRIN
TSPC(18);"{CUR UP}?INCORRECT DIGITS":GOT
O1020
1100 HI$=LEFT$(C4$,2):LO$=RIGHT$(C4$,2)
1110 HEX$=HI$:GOSUB2000:HI=DEC
1120 HEX$=LO$:GOSUB2000:LO=DEC
1200 OP=-1
1210 IFLEN(C4$)=2THENB1=2
1220 IFLEN(C4$)=4THENB1=3
1230 RESTORE
1240 FOR R=0 TO 255
1250 READOP$:BY=ASC(LEFT$(OP$,1))-48:IFB
Y<>B1THEN1270
1255 OP$=RIGHT$(OP$,LEN(OP$)-1)
1260 IFOP$=C3$THENOP=R:R=255
1270 NEXT
1280 IFOP=-1THENPRINTSPC(18);"{CUR UP}?U
NKNOWN CODE":GOTO1020
1290 IFOP=160ROP=480ROP=80ROP=112ORP=1
440ROP=176OROP=208OROP=240THEN1330
1300 IFB1=3THENPOKEP,OP:POKEP+1,LO:POKEP
+2,HI
1305 IFB1=2THENPOKEP,OP:POKEP+1,LO
1307 IFB1=1THENPOKEP,OP
1310 P=P+B1:DEC=P:GOSUB3000:P$=HEX$
1320 GOTO1010
1330 AD=HI*256+LO:DI=0
1340 IF AD>P THEN DI=AD-P:IF DI>127 THEN
PRINTSPC(18);"{CUR UP}?BAD BRANCH":GOTO1
010
1350 IF AD<PTHENDI=(P-AD)*-1:IF DI<-128
THENPRINTSPC(18);"{CUR UP}?BAD BRANCH":G
OTO1010
```

```
1360 IFDI<ØTHENLO=254+DI
1365 IFDI=ØTHENLO=254
1367 IFDI=1THENLO=255
1370 IFDI>1THENLO=DI-2
1380 B1=2:GOTO1305
2000 DEC=Ø
2010 FOR R=ØTO1
2020 S=ASC(MID$(HEX$,2-R,1))-48
2030 IFS>1ØTHENS=S-7
2040 IF S<16ANDS>-1THENDEC=DEC+(16^R)*S
2050 NEXT
2060 RETURN
3000 D(1)=INT(DEC/4096)
3010 DEC=DEC-D(1)*4096
3020 D(2)=INT(DEC/256)
3030 DEC=DEC-D(2)*256
3040 D(3)=INT(DEC/16)
3050 DEC=DEC-D(3)*16
3060 D(4)=DEC:HEX$=""
3070 FOR R=1TO4
3080 IF D(R)>9THEND(R)=D(R)+7
3090 HEX$=HEX$+CHR$(D(R)+48)
3100 NEXT
3110 RETURN
4000 INPUT"START OF CODE ";P$:IFLEN(P$)<
>4THEN4000
4010 HEX$=LEFT$(P$,2):GOSUB2000:P=DEC*25
6
4020 HEX$=RIGHT$(P$,2):GOSUB2000:P=P+DEC
4030 OP=PEEK(P):RESTORE:FOR R=ØTOOP
4040 READOP$
4050 NEXT:BY=VAL(LEFT$(OP$,1)):OT$=""
4053 IFOP$="?"THENOP$="????":BY=1
4055 L=LEN(OP$)-1:OP$=RIGHT$(OP$,L)
4060 IFBY=1THENOP$=RIGHT$(OP$,3):GOTO411
Ø
```

```
4065 IFOP=160ROP=480ROP=800ROP=1120ROP=1
440ROP=1760ROP=2080ROP=240THEN4180
4070 IFRIGHT$(OP$,3)=").Y"THENOT$=").Y":
OP$=LEFT$(OP$,L-3):GOTO4120
4080 IFRIGHT$(OP$,2)=".X"THENOT$=".X":OP
$=LEFT$(OP$,L-2)
4090 IFRIGHT$(OP$,2)=".Y"THENOT$=".Y":OP
$=LEFT$(OP$,L-2)
4100 IFRIGHT$(OP$,3)=".X)"THENOT$=".X)":
OP$=LEFT$(OP$,L-3):GOTO4120
4105 IFRIGHT$(OP$,1)=")"THENOT$=")":OP$=
LEFT$(OP$,L-1)
4110 IFBY=1THENPRINTP$;"  ";OP$
4120 IFBY=2THENDEC=PEEK(P+1):GOSUB3000:P
RINTP$;"  ";OP$;"$";RIGHT$(HEX$,2);OT$
4130 IFBY=3THENDEC=PEEK(P+2)*256+PEEK(P+
1):GOSUB3000:PRINTP$;"  ";OP$;"$";HEX$;OT
$
4140 P=P+BY:DEC=P:GOSUB3000:P$=HEX$
4150 GETA$:IFA$="X"THENRETURN
4160 IF A$<>""THENFORR=0TO100:NEXT:GOTO4
150
4170 GOTO4010
4180 DI=PEEK(P+1):IFDI>127THENDI=(256-DI
)*-1
4190 DEC=P+DI+2:GOSUB3000:BY=2
4200 PRINTP$;"  ";OP$;"$";HEX$
4210 GOTO4140
5000 INPUT"START OF INTERPRETATION ";P$:
IFLEN(P$)<>4THEN5000
5010 HEX$=LEFT$(P$,2):GOSUB2000:P=DEC*25
6
5020 HEX$=RIGHT$(P$,2):GOSUB2000:P=P+DEC
5030 PRINTP$;" (RVS ON)";:FOR R=0 TO 19
5040 CH=PEEK(P+R)
```

```
5050 IFCH<32ORCH>95THENCH=32
5060 PRINTCHR$(CH);
5070 NEXT
5080 P=P+20:DEC=P:GOSUB3000:P$=HEX$
5090 PRINT
5100 GETA$:IFA$="X"THENRETURN
5110 IF A$=" "THENFORR=0TO100:NEXT:GOTO5
100
5120 GOTO5010
6000 INPUT"START OF DATA ";P$:IFLEN(P$)<
>4THEN6000
6010 HEX$=LEFT$(P$,2):GOSUB2000:P=DEC*25
6
6020 HEX$=RIGHT$(P$,2):GOSUB2000:P=P+DEC
6030 PRINTP$;" ";:FOR S=0 TO 9
6040 DEC=PEEK(P+S):GOSUB3000
6050 PRINT" ";RIGHT$(HEX$,2);
6060 NEXT
6070 P=P+10:DEC=P:GOSUB3000:P$=HEX$
6080 PRINT
6090 GETA$:IFA$="X"THENRETURN
6100 IF A$=" "THENFORR=0TO100:NEXT:GOTO6
090
6110 GOTO6010
7000 INPUT"DISK OR TAPE (D/T) ";A$:A$=LE
FT$(A$,1):IFA$="T"THENPOKE251,1
7010 IFA$="D"THENPOKE251,8
7020 IFA$<>"D"ANDA$<>"T"THEN7000
7030 INPUT"FILENAME (1-6 CHARACTERS) ";A
$:IFLEN(A$)>6ORLEN(A$)=0THEN7030
7040 POKE820,LEN(A$):FORR=0TOLEN(A$)-1
7050 POKE821+R,ASC(MID$(A$,R+1,1))
7060 NEXT:OP$="SAVE ":IFZ=-1THENOP$="LOA
D "
7070 PRINT"START OF ";OP$;:INPUT#50,P$:P
```

```
RINT
7080 HEX$=LEFT$(P$,2):GOSUB2000:POKE253,
DEC
7090 HEX$=RIGHT$(P$,2):GOSUB2000:POKE252
,DEC:IFZ=-1THEN7130
7100 PRINT"END OF ";OP$;:INPUT#50,P$:PRI
NT
7110 HEX$=LEFT$(P$,2):GOSUB2000:POKE1021
,DEC
7120 HEX$=RIGHT$(P$,2):GOSUB2000:POKE102
0,DEC
7130 RESTORE:FORR=0TO255:READA$:NEXT
7140 FORR=0TO26:READA:POKE679+R,A:NEXT
7150 IFZ=-1THENFORR=0TO26:READA:POKE679+
R,A:NEXT
7160 Z=0:SYS679:RETURN
7600 Z=-1:GOTO7000
8000 INPUT"ADDRESS OF THE CODE ";P$:IFLE
N(P$)<>4THEN8000
8010 HEX$=LEFT$(P$,2):GOSUB2000:P=DEC*25
6
8020 HEX$=RIGHT$(P$,2):GOSUB2000:P=P+DEC
8030 SYS(P)
8040 PRINT:PRINT"   AR   XR   YR   SP"
8050 DEC=PEEK(780):GOSUB3000:AR$=RIGHT$(
HEX$,2)
8060 DEC=PEEK(781):GOSUB3000:XR$=RIGHT$(
HEX$,2)
8070 DEC=PEEK(782):GOSUB3000:YR$=RIGHT$(
HEX$,2)
8080 POKE820,186:POKE821,96:SYS820
8090 DEC=PEEK(781):GOSUB3000:SP$=RIGHT$(
HEX$,2)
8100 PRINT"   ";AR$;"   ";XR$;"   ";YR$;
"   ";SP$
```

```
8110 PRINT:PRINT"PRESS ANY KEY":POKE198,
0:WAIT198,1
8120 RETURN
9000 INPUT"HEX TO DECIMAL (H) OR DECIMAL
 TO HEX (D)";A$:A$=LEFT$(A$,1)
9010 IFA$<>"H"ANDA$<>"D"THEN9000
9020 IFA$="H"THEN9100
9030 PRINT:PRINT"USE NUMBERS BETWEEN 0 A
ND 65535 ONLY"
9040 INPUT#50,A$:IFA$="X"THENRETURN
9050 DEC=VAL(A$):IFDEC>65535ORDEC<0THEN9
030
9060 GOSUB3000:PRINTTAB(10);HEX$
9070 GOTO9040
9100 PRINT:PRINT"USE ONE TO FOUR DIGIT N
UMBERS"
9110 INPUT#50,HEX$:IFHEX$="X"THENRETURN
9120 IFLEN(HEX$)>4THEN9100
9125 IFLEN(HEX$)=1ORLEN(HEX$)=3THENHEX$=
"0"+HEX$
9130 P=0:IFLEN(HEX$)=2THENGOSUB2000:GOTO
9150
9140 A$=HEX$:HEX$=LEFT$(A$,2):GOSUB2000:
P=DEC*256:HEX$=RIGHT$(A$,2):GOSUB2000
9150 P=P+DEC:PRINTTAB(10);P:GOTO9110
10000 DATA"1BRK","2ORA(.X)","?","?","?",
"2ORA","2ASL","?","1PHP","2ORA#","1ASL"
10010 DATA"?","?","3ORA","3ASL","?","3BP
L","2ORA().Y","?","?","?","2ORA.X"
10020 DATA"2ASL.X","?","1CLC","3ORA.X","
?","?","?","3ORA.X","3ASL.X","?","3JSR"
10030 DATA"2AND(.X)","?","?","2BIT","2AN
D","2ROL","?","1PLP","2AND#","1ROL"
10040 DATA"?","3BIT","3AND","3ROL","?","
3BMI","2AND().Y","?","?","?"
```

```
1005Ø DATA"2AND.X","2ROL.X","?","1SEC","
3AND.Y","?","?","?","3AND.X","3ROL.X"
1006Ø DATA"?","1RTI","2EOR(.X)","?","?",
"?","2EOR","2LSR","?","1PHA","2EOR#"
1007Ø DATA"1LSR","?","3JMP","3EOR","3LSR
","?","3BVC","2EOR().Y","?","?","?"
1008Ø DATA"2EOR.X","2LSR.X","?","1CLI","
3EOR.Y","?","?","?","3EOR.X","3LSR.X"
1009Ø DATA"?","1RTS","2ADC(.X)","?","?",
"?","2ADC","2ROR","?","1PLA","2ADC#"
1010Ø DATA"1ROR","?","3JMP()","3ADC","3R
OR","?","3BVS","2ADC().Y","?","?","?"
1011Ø DATA"2ADC.X","2ROR.X","?","1SEI","
3ADC.Y","?","?","?","3ADC.X","3ROR.X"
1012Ø DATA"?","?","2STA(.X)","?","?","2S
TY","2STA","2STX","?","1DEY","?","1TXA"
1013Ø DATA"?","3STY","3STA","3STX","?","
3BCC","2STA().Y","?","?","2STY.X"
1014Ø DATA"2STA.X","2STX.Y","?","1TYA","
3STA.Y","1TXS","?","?","3STA.X","?","?"
1015Ø DATA"2LDY#","2LDA(.X)","2LDX#","?"
,"2LDY","2LDA","2LDX","?","1TAY","2LDA#"
1016Ø DATA"1TAX","?","3LDY","3LDA","3LDX
","?","3BCS","2LDA().Y","?","?","2LDY.X"
1017Ø DATA"2LDA.X","2LDX.Y","?","1CLV","
3LDA.Y","1TSX","?","3LDY.X","3LDA.X"
1018Ø DATA"3LDX.Y","?","2CPY#","2CMP(.X)
","?","?","2CPY","2CMP","2DEC","?"
1019Ø DATA"1INY","2CMP#","1DEX","?","3CP
Y","3CMP","3DEC","?","3BNE","2CMP().Y"
1020Ø DATA"?","?","?","2CMP.X","2DEC.X",
"?","1CLD","3CMP.Y","?","?","?","3CMP.X"
1021Ø DATA"3DEC.X","?","2CPX#","2SBC(.X)
","?","?","2CPX","2SBC","2INC","?"
1022Ø DATA"1INX","2SBC#","1NOP","?","3CP
```

```
X","3SBC","3INC","?","3BEQ","2SBC().Y"
1Ø23Ø DATA"?","?","?","2SBC.X","2INC.X",
"?","1SED","3SBC.Y","?","?","?"
1Ø24Ø DATA"3SBC.X","3INC.X","?"
1Ø25Ø DATA166,251,32,186,255,173,52,3,16
2,53,16Ø,3,32,189,255,174,252,3,172,253
1Ø26Ø DATA3,169,252,32,216,255,96
1Ø3ØØ DATA166,251,32,186,255,173,52,3,16
2,53,16Ø ,3,32
1Ø31Ø DATA189,255,174,252,3,172,253,3,16
9,Ø,32,213,255,96
```

There will be explanations on how to use the program in following chapters.

# CHAPTER THREE — ACCESSING MACHINE CODE

Now that you have finished typing out the assembler program, RUN it. There should be a menu on the screen. Press the 1 key. The computer should then prompt you with 'START OF CODE?' Answer this by typing C000 and RETURN.

Now you will be in a position to enter assembly language at 'location' C000. In assembly language we don't have line numbers but instead have addresses. C000 is an address. You don't know any assembly language yet so press X and RETURN. This will return you to the main menu.

Now press 2. Again you will be prompted with 'START OF CODE'. This time answer it with F301. The computer will now be printing out the instructions from address F301 and onward. It doesn't matter that you don't understand what is being printed. You will understand it shortly.

To stop the printing, hold the SPACE BAR down. Now press X again. You should return to the main menu. Press 7. This time, when you are prompted with 'ADDRESS OF THE CODE', type FFD2. The computer will now RUN the assembly language at the address FFD2 and print some numbers and characters on the screen. These numbers and characters will be explained in the next chapter. Press any key and you will be returned to the main menu.

After this, press 8, then D. This allows you to enter decimal numbers and the computer will convert them to hex. After you have tried a few, press X. Now press 8 again, then H. Now you will be able to enter hex numbers and the computer will convert them to decimal. Press X if you want to return to the main menu.

# CHAPTER FOUR — LOADING REGISTERS

There are three main registers in assembly language on the Commodore 64. These registers may be thought of as variables. These three variables or registers are called the Accumulator (A), X register and Y register. Unlike BASIC variables these can only have values from zero to 255 (0 - FF in hex).

To make a register equal to a value, we must *load* it with that value. For example: In BASIC, to make the variable A equal 55, we would type 'A = 55'. In assembly language it would be 'LDA# $37', the LDA stands for 'load A'. The # means 'with the following value' and the $ means a hex number. The A register is loaded with 37 and not 55 because we must use hex numbers in assembly language, and 37 in hex is the same as 55 decimal.

Registers may be loaded in many different ways. The one above is called Immediate. In due course, I'll show you all the ways to load registers. However, first we'll discuss how to use the assembler to write the assembly language.

LOAD and RUN the assembler. Press 1, then type C000, we will put our code at addresses C000 and on.

Now type 'LDA# $37' and press RETURN. When the cursor re-appears type RTS. This RTS instruction will return the computer to BASIC when it is RUN. If you don't put an RTS at the end of an assembly language program, the computer won't return and could CRASH.

The code that you have typed in would look like this:

```
C000 LDA#$37

C002 RTS
```

Now press X. You should return to the main menu. Press 7. You should be prompted with 'ADDRESS OF CODE'. Type C000 and press RETURN. The computer will now run the assembly language that you typed in. The A register will be loaded with 37.

After the computer has returned from the assembly language, it will print out the values of the registers on the screen. AR stands for A register, XR for X register, YR for Y register and SP for Stack Pointer. (Don't worry about the SP for now, as I will explain this in later chapters.)

Notice that the A register has a value of 37, the value that we loaded it with. The X register and Y register can be loaded in the same way, here are some examples:

```
C000 LDX#$9A

C002 RTS


C000 LDY#$50

C002 RTS
```

The first example loads the X register with 9A, the second loads the Y register with 50. (It is important to remember that all the figures given in this chapter and following chapters are hexadecimal, unless I state otherwise.)

# Zero Page

Before reading on you should be familiar with the PEEK statement. If not, consult your User's Guide, pages 62 and 126. All of the following instructions perform PEEK functions. Zero Page is the name given to the addresses between 0 and FF.

In BASIC, to make the variable A equal the value of location C5, we would type 'A = PEEK (197)' (197 is decimal for C5). In assembly language it is 'LDS$C5'. The difference between this and the previous load instruction, as you can see, is that there is no # sign.

Here is a demonstration program which loads the A register with the value of location C5. Remember to use option 7 on the main menu to run it.

```
C000 LDA$C5

C002 RTS
```

The X and Y registers can be loaded in the same way:

```
C000 LDX$D7

C002 RTS


C000 LDY$03

C002 RTS
```

The first example loads the X register with the value of location D7. The second program loads the Y register with the value of location 03. Note that it is 03 and not 3. Although they mean the same thing, the assembler will only accept two-digit or four-digit numbers.

# Indexed Addressing (Zero Page)

In BASIC if you wanted to make the variable A equal location 45+X you would type 'A = PEEK (45+X)'. The assembly language equivalent is 'LDA$45.X'. Here is a program to do this:

```
C000 LDA$45,X

C002 RTS
```

In the previous example, if X had a value of 10 then the A register would have been loaded with the value of location 55.

The X register can be loaded with the Y register as an index, but not with the A register or itself. In the following example, the X register will be loaded with the value of location 67, because the Y register is equal to 27 and 40+27=67:

```
C000 LDY#$27

C002 LDX$40,Y

C004 RTS
```

The Y register can also be loaded in this way, using the X register as the index. The following program loads the Y register with the value of location 03 (1+2=3):

```
C000 LDX#$02

C002 LDY$01,X

C004 RTS
```

# Absolute Addressing

Unlike Zero Page addressing which has a limited range, Absolute enables you to use figures ranging from 00 to FFFF.

Location 0286 holds the current colour of the cursor. To find out the colour of the cursor we can make the A register equal to the value of location 0286.

Here is the program which will do this:

```
C000 LDA$0286

C003 RTS
```

After you have run through it once, change the colour of the cursor. As you change the colour of the cursor, the value will change.

The X and Y registers can also be loaded in this way.

# Absolute Indexed Addressing

This is much the same as Zero Page Indexed Addressing. However, because it is absolute you can use numbers ranging from 00 to FFFF.

The A register can use either the X or Y register as an index. On the Commodore 64 the screen starts at address 0400. So, to load the A register with the first character on the screen we would type 'LDS$0400'. If we wanted to load the A register with Xth character on the screen we would use 'LDA$0400.X'.

Here is a program to load the Xth character on the screen:

```
C000 LDX#$09

C002 LDA$0400.X

C005 RTS
```

If you change the value of the X register, you will load the A register with a different character.

The X register can also be loaded in this way, although it can only use the Y register as an index.

The Y register uses the X register as an index in Absolute Indexed Addressing.

# Indexed Indirect Addressing

Only the A register can use this addressing. Before we go any further, it's important that you understand what the terms 'HI byte' and 'LOW byte' mean.

LOW byte is the first two digits of an address. For example, in the address FEA9, the LOW byte is A9. The address 764F has a LOW byte of 4F.

HI byte is the second pair of digits in an address. The address FEA9 has a HI byte of FE, and 764F has a HI byte of 76.

Let's join our bytes to Indexed Indirect Addressing to carry out the instruction 'LDA($40.X)'. When the computer comes across this instruction, it adds the value of the X register to the number in brackets. For this example, let's assume that the X register equals 10. If we did make this assumption, the total inside the brackets would be 50. (For this example, let's assume that location 50 equals D2.) Then the computer will

get the HI byte from location 51 (we'll assume that location 51 equals FF).

The computer now has a LOW byte and a HI byte, so it can form an address. The address will be FFD2, LOW byte D2,HI byte FF. Now that the address has been formed the A register will be loaded with the value of location FFD2.

I will go through another example before we actually use this instruction on the computer. For this example, assume that the X register equals 20, location 30 equals 43 and location 31 equals FE. The instruction for this example will be 'LDA($10.X)'. The LOW byte will be the value of location 30, which is 43. The HI byte would be the value of location 31, which is FE. The complete address will be FE43, so the A register will be loaded with the value of location FE43:

```
C000 LDX#$0B

C002 LDA($20.X)

C004 RTS
```

When you RUN the program the A register should end up with a value of OB. All I should need to tell you is:- location 2B equals 01; location 2C equals 08; and location 0801 equals OB.

# Indirect Indexed Addressing

Like the last mode, this one can only be used by the A register. An example of this mode in use is 'LDA($78).Y'. The computer gets its LOW byte from location 78, then its HI byte from location 79. After it forms an address, it adds the value of the Y register to the address.

If location 78 equalled 56, location 79 equalled 98 and the Y register equalled 03 then the following would happen:

The LOW byte (location 78) would be 56, the HI byte (location 79) would be 98. The address would therefore be 9856, although this wouldn't be the final address. The computer would next add the value of the Y register to the address to get a final address of 9859. The A register would then be loaded with the value of location 9859.

Here is an example program:

```
C000 LDY#$10

C002 LDA($43),Y

C004 RTS
```

In the above example, location 43 equals 04, location 44 equals 02 and location 0204 equals 20.

There are no more load instructions for you to learn, so here's a small test of your knowledge to date:

Location A0 holds the HI byte of the 64's clock. Location A1 holds the MID byte (two middle digits) and location A2 holds the LOW byte.

How might you read the whole clock, all at once?

There are many answers to this question, although the best answer follows.

```
C000 LDA$A0

C002 LDX$A1

C004 LDY$A2

C006 RTS
```

Whether your program is the same as this one doesn't really matter, so long as it worked. If your program doesn't work I suggest that you go back over this chapter carefully.

# CHAPTER FIVE — STORING THE REGISTERS IN MEMORY

Now that we know how to load registers it is important that we know how to store the values of them. We will need to store the registers because there are only three of them. Could you imagine writing a BASIC program with just three variables?

In this chapter you will learn how to store registers in memory. Once they are in memory you can load the values back into the registers at will, using the instructions from the previous chapter.

## Zero Page Addressing

The first store instruction we will examine is STA. This stands for Store the A register. We are using Zero Pages so we will be able to store the A register in locations 00 to FF.

The following program stores the A register in location FB:

```
C000 LDA#$80

C002 STA$FB

C004 RTS
```

Execute the program, then return to the main menu. Now press 4. You should then be prompted with 'START OF DATA?' Type 00FB then RETURN. The computer will start printing out numbers. These numbers are the values of memory locations. Notice that the first one is 80, the value we stored in location FB.

The X and Y registers can also have their values stored in Zero Page. They use the instructions STX (Store the X register in memory) and STY (Store the Y register in memory).

Here is an example of each:

```
C000 LDX#$23

C002 STX$FE

C004 RTS


C000 LDY#$0D

C002 STY$CB

C004 RTS
```

After you execute the code you can check that it worked by using option 4 from the main menu.

# Zero Page Indexed Addressing

This works in the same way as it did for the load instructions, except for the fact that this time we are storing. For this instruction the A register uses the X register as an index. 'STA$25.X' is an example of an actual instruction. This one would store the A register at location 25+X (X register).

Here is a program to show this:

```
C000 LDX#$45

C002 LDA#$FF

C004 STA$25.X

C006 RTS
```

The A register is stored in location 6A, because 25+X = 25+45 = 6A. You can check this by using option 4 on the main menu.

The X register can also store itself in this way, although it uses the Y register as an index. The Y register uses the X register as an index when it stored by this method.

Here is an example of each:

```
C000 LDX#$80

C002 LDY#$05

C004 STX$F0.Y

C006 RTS


C000 LDY#$00

C002 LDX#$F0

C004 STY$11.X

C006 RTS
```

The first example stores the X register in location F5, because F0+Y = F0+5 = F5. The second example stores the Y register in location 101, because 11+X = 11+F0 = 101.

# Absolute Addressing

All three registers can use Absolute Addressing to store themselves. We will examine the A register first.

As you might recall, I told you that the colour of the cursor is stored at location 0286.

The following program will change the colour of the cursor by changing the value of location 0286:

```
C000 LDA#$00

C002 STA$0286

C005 RTS
```

By changing the value of the A register in the previous program, you can change the colour, that is, you load the A register with a different value.

Location 028A controls the key repeat. If this location equals 80 then the keys repeat, otherwise they don't.

The following program uses the X register to set key repeat:

```
C000 LDX#$80

C002 STX$028A

C005 RTS
```

To test that it worked, break out of the program using RUN/STOP and hold a key down.

Location D020 holds the background colour of the screen.

The following program will use the Y register to change the colour of the background:

```
C000 LDY#$00

C002 STY$D020

C005 RTS
```

By changing the value that Y is loaded with you can change the background colour to the one that you want.

# Absolute Indexed Addressing

The A register is the only register that can be stored using Absolute Indexed Addressing. This time the A register may use either the X or Y register as an index.

The following programs store a character on the screen. See if you can spot them:

```
C000 LDX#$29

C002 LDA#$21

C004 STA$0400,X

C007 RTS
```

```
C000 LDY#$F0

C002 LDA#$21

C004 STA$0400,X

C007 RTS
```

The first program stores a character in the top left of the screen, the second stores a character on the right side of the screen. They both work because the

screen memory starts at location 0400.

# Indexed Indirect Addressing

Again the A register is the only register that can use this form of addressing. An example of this instruction would be 'STA($32.X)'. If this instruction was executed and X equalled 14, then the LOW byte would be the value of location 46 (32+X) and the HI byte would be the value of location 47. After this the computer would form an address and store the A register at that location.

Here is an example program (a full explanation follows the program):

```
C000 LDX#$21

C002 STX$FB

C004 LDX#$D0

C006 STX$FC

C008 LDA#$00

C00A LDX#$01

C00C STA($FA.X)

C00E RTS
```

When you RUN the assembly, the background colour should change.

C000: this line loads the X register with the LOW byte.
C002: the LOW byte is stored in location FB.
C004: the X register is loaded with the HI byte.
C006: the HI byte is stored in location FC.
C008: this loads the A register with 00.
C00A: the X register is loaded with 01.
C00C: the computer gets the LOW byte from location FB (FA+X), then the HI byte from location FC (FA+X+1). The computer then forms the address D021, which is the location which holds the background colour. The A register is then stored there, changing the colour to black.

# Indirect Indexed Addressing

As you know, the A register is the only register which can use this form of addressing. This form of addressing is much like the previous one, except that it uses the Y register as an index and it adds the Y register after the address has been formed, not before.

Here is a program to show this. It stores a character in the top left of the screen:

```
C000 LDX#$28

C002 STX$FB

C004 LDX#$04

C006 STX$FC

C008 LDY#$28

C00A LDA#$23

C00C STA($FB),Y

C00E RTS
```

C000: this loads the X register with the LOW byte.
C002: the LOW byte is now stored in location FB.
C004: now the X register is loaded with the HI byte.
C006: the HI byte is stored in location FC.
C008: now the Y register (the index) is loaded with 28.
C00A: the A register is loaded with 23.
C00C: the computer gets the LOW byte from location FB, then the HI byte from location FC. It then forms the address 0400, next it adds Y to the address making it 0428. The A register is then stored at that address.

That was the final store instruction. Before you move onto the next chapter, I have devised another test for you.

The border colour is controlled by location D020 and the background colour is controlled by location D021.

The problem is:
Can you make a program that changes the border to the same colour as the background?

The answer follows.

```
C000 LDA$D021

C003 STA$D020

C006 RTS
```

The above answer is not the only answer, as we could have done it with the X or Y registers. If your answer is similar to mine, and it works, then proceed with the next chapter. If it didn't, then I advise you to go through the chapter again.

# CHAPTER SIX — INCREMENT, DECREMENT AND TRANSFER

Having only three registers sometimes becomes frustrating and on occasions it isn't practical to store the registers. We can get over this problem by transferring the value of one register to another.

For example, if the A register had a value that we didn't want to lose and we needed to use the A register for another function, we could transfer its value to the X register. This would be done with the instruction TAX, which stands for Transfer A to X. After the A register had been used, we could transfer the value from the X register back to A register. We do this with the instruction TXA (Transfer X to A).

In this example program, the A register is used, but still ends up with the value it started with:

```
C000 TAX

C001 LDA#$00

C003 STA$D020

C006 TXA

C007 RTS
```

C000: transfer the value of the A register to the X register.
C001: load the A register with 00.
C003: store the A register in location D020 (this changes the border to black).
C006: transfer the original value back into the A register.

The Y register can also be transferred, using TYA (Transfer Y to A) and TAY (Transfer A to Y).

# INX and INY

INX stands for Increment the X register. This means add 01 to the X register. INY stands for Increment the Y register, which means add 01 to the Y register.

Here is an example of each:

```
C000 LDX#$01

C002 INX

C003 RTS


C000 LDY#$08

C002 INY

C003 RTS
```

In the first example, the X register is increased from a value of 01 to 02. In the second example the Y register is incremented from a value of 08 to 09.

# Increment

Memory, like registers, can be incremented. Memory is incremented by using the command INC (INCrement memory).

# Zero Page Addressing

'INC$45' is an example of the increment instruction using Zero Page.

The following program makes location FB equal 04, and then increments it to a value of 05:

```
C000 LDA#$04

C002 STA$FB

C004 INC$FB

C006 RTS
```

To check whether it worked or not, use option 4 on the main menu.

# Zero Page Indexed

The INC instruction uses the X register as an index for this mode of addressing. An example of this instruction is 'INC$97.X', which would increment location 97+X.

This next program increments location 9A from a value of 44 to 45:

```
C000 LDA#$44

C002 STA$9A

C004 LDX#$03

C006 INC$97,X

C008 RTS
```

Again, to check it use option 4 on the main menu.

# Absolute Addressing

With this instruction, you can increment any location in memory. Every time you execute the next program, the border changes colour:

```
C000 INC$D020

C003 RTS
```

The border will keep changing each time you execute the program because the program increments location D020, which holds the colour of the border.

# Absolute Indexed Addressing

Again the X register is used as an index. The following program will show a peculiarity of the increment instructions:

```
C000 LDA#$FF

C002 STA$07F8

C005 INC$07F8

C008 RTS
```

If you use option 4 on the main menu you will see that the result of the increment is 00. This is an important point to remember. It means that any register or location of memory which equals FF before it is incremented will end up as 00.

# DEX and DEY

DEX stands for DEcrement the X register, and means subtract 01 from the X register. DEY stands for DEcrement the Y register, and means subtract 01 from the Y register.

Here is an example of each:

```
C000 LDX#$E4

C002 DEX

C003 RTS


C000 LDY#$09

C002 DEY

C003 RTS
```

# Zero Page Addressing

Now we move onto the next instruction, DEC, which stands for DECrement memory.

Here is an example of Zero Page DEC:

```
C000 LDA#$45

C002 STA$FB

C004 DEC$FB

C006 RTS
```

The program, when executed, stores 45 in location FB, then this is decremented to 44.

# Zero Page Indexed

For this instruction the X register is used as an index. I think it is worth noting that the BASIC ROM stores some of its values on Zero Page. That is why, when using Zero Page, you have to be careful which locations you alter. The Commodore 64 Reference Guide provides useful information on this.

Here is an example program for Zero Page Indexed INC:

```
C000 LDA#$FF

C002 STA$CC

C004 LDX#$0C

C006 INC$C0.X

C008 RTS
```

After you have executed this program the cursor should be flashing. We changed location CC, which is where the BASIC ROM stores the cursor enable.

# Absolute Addressing

With this instruction you can decrement any memory location, as in the following program:

```
C000 DEC$0286

C003 RTS
```

Each time you execute this program the cursor colour will change. This is because location 0286 holds the cursor colour.

# Absolute Indexed Addressing

The X register is used as an index for this instruction:

```
C000 LDA#$00

C002 STA$03FD

C005 LDX#$1D

C007 DEC$03E0.X

C00A RTS
```

After you have RUN the program, check the value of location 03FD, using option 4 on the main menu. It should be FF. This happened because we decremented a memory location which had a value of 00. From this we can see that any location that is equal to 00, and is decremented, will end up with a value of FF.

That was the last instruction in this chapter, so it's test time again!

As you may have noticed, the A register has no decrement instruction, nor has it an increment instruction. The problem:

Load the A register with 55 and decrement it to a value of 54. (HINT!! Transfer).

There are two equally good answers. See if you can work them both out before seeing how I did it.

```
C000 LDA#$55

C002 TAY

C003 DEY

C004 TYA

C005 RTS
```

OR

```
C000 LDA#$55

C002 TAX

C003 DEX

C004 TXA

C005 RTS
```

If your program worked, or better still you had either of the above, then proceed with the next chapter. Otherwise I suggest you go over this chapter again carefully.

# CHAPTER SEVEN — JUMPING

Before reading on it is necessary for you to be familiar with BASIC's GOTO and GOSUB. Explanations on these can be found in the User Guide.

# JMP

This instruction is very similar to the GOTO instruction. If you want to jump to a new address you can use this instruction.

For example, if you wanted to jump to the address B000, you would type JMP$B000.

Here is an example program (a full explanation follows the program):

```
C000 JMP$C006

C003 LDA#$00

C005 RTS

C006 LDA#$FF

C008 RTS
```

C000: jump to location C006.
C003: this loads the A register with 00, although it will never be executed, because we have jumped over it.
C005: this would return the computer to BASIC, although it has also been jumped over.
C006: this is where the computer has jumped to. This line loads the A register with FF.
C008: return to BASIC.

# Indirect Addressing

The JMP command supports indirect addressing. It is written in the form JMP ($XXXX), where XXXX is a four-digit hex number.

When you execute this instruction the program will stop, and the cursor will appear. This happens because the LOW byte for error messages is stored at location 0302, and the HI byte is stored at location 0303. When it is executed the computer jumps to the error message routine and stops, because there is no error.

# SYS

You may have come across this before. It isn't an assembly language command but in fact is BASIC. This command will go to machine code and return to BASIC

when it encounters an RTS statement. For example, C000 in hex is the same as 49152 in decimal, so to RUN any of the programs that we have done so far just type 'SYS 49152'. The SYS command is used after you have written the assembly language routine, and no longer need the assembler.

# JSR and the STACK

Every time you have executed an assembly language program the assembler has told you the values of the registers, including the SP register. It is now time to tell you what the SP is.

SP stands for Stack Pointer. You may have heard of the stack before. The stack is the place where return addresses are put. That is, when the computer goes on a GOSUB, the line number that it must go back to when the RETURN statement is executed is stored on the stack.

These are the main points to note:

The computer stores the address on the stack.

The computer meets a return statement.

The computer gets the address back off the stack.

The computer "returns" to that address.

The assembly language equivalent for GOSUB is JSR, which stands for Jump to Sub-Routine. The equivalent for the RETURN statement is RTS, which stands for Return from Sub-routine. The stack behaves in the same way for JSR and RTS as it does for GOSUB and RETURN.

The actual stack is 255 bytes of memory that stretches from location 0100 to 01FF. The stack pointer points to the next free byte on the stack. The stack pointer starts off with a value of FF, which points to location 01FF, then BASIC takes a few bytes off and we end up with the stack pointer pointing to location 01EF.

Each time you use a JSR instruction or a SYS the computer saves the LOW byte of the return address onto the stack, then it decrements the stack pointer. After that the HI byte of the return address is stored on the stack, and once again the stack pointer is decremented.

Each time you use an RTS statement the computer takes the HI byte off the stack and increments the stack pointer. It then takes the LOW byte off the stack and increments the stack pointer again. The computer then forms the return address by putting the LOW and HI bytes together. After the address has been formed the computer returns to that address.

By jumping to subroutines that BASIC uses we can print characters, move the cursor and the like. To print a character, you load the A register with the ASCII value of that character and JSR$FFD2:

```
C000 LDA#$41

C002 JSR$FFD2

C005 RTS
```

This program prints the letter A, because the A register is loaded with the ASCII value of the letter A before the routine is called. To print other characters, look at the ASCII chart on pages 135-137 of your User's Manual.

This next routine sets the x and y co-ordinates of the cursor. First you load the X register with the x co-ordinate, then the Y register with the y co-ordinate. Now type CLC (this command will be explained in the next chapter), then JSR$FFF0:

```
C000 LDX#$00

C002 LDY#$05

C004 CLC

C005 JSR$FFF0

C008 RTS
```

As should be obvious, the cursor was set to the upper left of the screen. By combining the two previous routines you should be able to print any character at any position on the screen.

That concludes this chapter, so here is another problem for you to solve:

Can you clear the screen and set the cursor to 01,01?

```
C000 LDA#$93

C002 JSR$FFD2

C005 LDX#$01

C007 LDY#$01

C009 CLC

C00A JSR$FFF0

C00D RTS
```

The above program prints a clear home (ASCII 93 hex or 147 decimal) and then sets the cursor to 01,01.

If you didn't get the above or your program didn't work then don't worry, so long as you know how the program works. If you don't know how it works then I advise you once again to read over this chapter.

# CHAPTER EIGHT — THE PROCESSOR STATUS REGISTER

The Processor Status Register (P register) tells us the state of the computer. We will discover how to test the status in the next chapter. In this chapter I will show you what the P register is.

As you know, a byte has eight bits, and each of those bits may be a 0 or a 1. The P register uses each bit as a flag. Each bit is set or reset, depending on the status of the computer.

Here is an explanation of what each bit does:

Bit 7: this is called the negative flag. This bit is set to 1 when the result of operation is a number between 80 and FF. For example, LDA# $C7 would set this bit. The bit is reset if the result of an operation is between 00 and 7F.

Bit 6: this is called the overflow flag. It is set when an increment goes above 7F or a decrement goes below 80. Otherwise it is reset.

Bit 5: this bit doesn't do anything; it isn't used.

Bit 4: this flag is for BRK (force break) and this will be discussed later.

Bit 3: this bit is called the Decimal flag. This will also be discussed later.

Bit 2: this flag is called IRQ disable and, once again, will be discussed later, along with the BRK flag.

Bit 1: this bit is the Zero flag. It is set if the result of an operation is 00, otherwise it is reset.

Bit 0: this bit is called the carry flag. You will learn more about this flag in the next chapter.

# Setting and Clearing Flags

Some of the flags have commands that will set them to 1 or reset to 0 as follows:

CLC: Clear the carry flag.
SEC: Set the carry flag.

CLD: Clear the Decimal flag.
SED: Set the Decimal flag.

CLI: Clear the IRQ flag.
SEI: Set the IRQ flag.

CLV: Clear the Overflow flag.

# CHAPTER NINE — COMPARE INSTRUCTIONS

Before we begin studying the compare instructions it is necessary for you to add the following lines to your assembler:

```
8040 PRINT:PRINT"{CLR}    AR    XR    YR
SP"
8080 DEC=PEEK(783)
8090 GOSUB3000:SP$=RIGHT$(HEX$,2)
8101 POKE760,8:POKE761,104:POKE762,96:SY
S760
8102 BI$="":BI=PEEK(780):FOR R=7TO0STEP-
1:IFBI<(2^R)THENBI$=BI$+" 0":GOTO8106
8104 BI$=BI$+" 1":BI=BI-(2^R)
8106 NEXT
8108 PRINT"{CUR DN}{CUR DN} N V - B D I
Z C":PRINTBI$
```

After you have typed them in re-SAVE the assembler.

These new lines find the value of the P register and print out the value of each flag.

There are three compare instructions, one for each of the X, Y and A registers. The compare instruction for the X register is CPX, which stands for Compare the X register. The Y register's compare instruction is CPY, which stands for Compare the Y register. The A register's compare instruction is CMP, which stands for Compare the A register.

# Immediate Addressing

This is the form of addressing with the # symbol. An example of the A register's compare instruction would be 'CMP# $67'. This would compare the A register with the 67. The results of compare instructions are as follows:

1. If the register is less than the data it is compared with, the Negative (N) flag is set.

2. If the register and the data are equal the Zero and Carry flag will be set.

3. If the register is greater than the data the Carry flag is set.

The above points are important, they cater for every result of a compare instruction.

Here is an example of CMP using Immediate addressing:

```
C000 LDA#$56

C002 CMP#$70

C004 RTS
```

After you execute the program you will see that the Negative flag is set. This happens because the register (A register) is less than the data (70).

Here is another example, this time using CPX:

```
C000 LDX#$67

C002 CPX#$67

C004 RTS
```

The Carry flag and the Zero flag will be set when this program is executed. This happens because the register (X register) and the data (67) are equal.

Here's yet another example, this time using CPY:

```
C000 LDY#$60

C002 CPY#$20

C004 RTS
```

Only the Carry flag is set this time, because the register (Y register) is greater than the data (20).

# Zero Page Addressing

All of the registers can be compared with locations on Zero Page. The first one we will look at is CMP:

```
C000 LDX#$09

C002 STX$FB

C004 LDA#$06

C006 CMP$FB

C008 RTS
```

The A register is found to be less than location FB, because the Negative flag was set.

Here are examples of CPX and CPY:

```
C000 LDA#$57

C002 STA$FD

C004 LDX#$58

C006 CPX$FD

C008 RTS


C000 LDA#$30

C002 STA$FE

C004 LDY#$30

C006 CPY$FE

C008 RTS
```

The first program sets the Carry flag because the register (X) is greater than the data (57). The second program sets both the Carry and Zero flags because the register (Y) is equal to the data (30).

# Zero Page Indexed Addressing

The A register is the only register that has Zero Paged Indexed addressing for the compare instruction. The X register is used as the index:

```
C000 LDX#$10

C002 LDA#$01

C004 CMP$B7,X

C006 RTS
```

The above program compares the A register with location C7 (B7+X). Location C7 controls reverse/ non-reverse printing. If it is equal to 01 then the computer prints reverse. Therefore if you are printing in reverse and you execute the above program the Carry and Zero flags will be set.

# Absolute Addressing

All three registers can use this form of addressing for comparing.

Here are examples of each:

```
C000 LDA#$01

C002 CMP$0286

C005 RTS


C000 LDX#$00

C002 CPX$D015

C005 RTS


C000 LDY#$00

C002 CPY$0291

C005 RTS
```

The first example tests the A register against location 0286. The second tests the X register against location D015, which holds the sprite enable flags. The final program tests the Y register against location 0291.

# Absolute Indexed Addressing

The A register is the only register that supports this form of addressing for its compare instruction. The X register is used as an index as you can see in this example program:

```
C000 LDX#$77

C002 LDA#$64

C004 CMP$0200.X

C007 RTS
```

The above program compares the A register with the first location of the keyboard buffer, location 0277. The Y register can also be used as an index for this instruction.

# Indexed Indirect Addressing

Again the A register is the only register which supports this form of addressing. This time the X register is the only register that can be used as the index.

Here is an example program:

```
C000 LDY#$0D

C002 STY$FB

C004 LDY#$03

C006 STY$FC

C008 LDX#$1B

C00A LDA#$20

C00C CMP($E0,X)

C00E RTS
```

C000: this loads the Y register with OD.
C002: this stores the Y register in location FB.
C004: now the Y register is loaded with 03.
C006: the Y register is stored in location FC.
C008: this loads the X register with 1B.
C00A: now the A register is loaded with 20.
C00C: the computer gets the LOW byte from location FB and the HI byte from location FC. It then forms the address 030D and compares the value of that location against the A register.
C00E: this returns the computer to BASIC.

# Indirect Indexed Addressing

The A register is the only register to have this form of addressing. This time the Y register is used as an index:

```
C000 LDX#$00

C002 STX$FD

C004 LDX#$04

C006 STX$FE

C008 LDY#$09

C00A LDA#$10

C00C CMP($FD),Y

C00E RTS
```

C000: this loads the X register with 00.
C002: this stores the X register in location FD.
C004: now the X register is loaded with 04.
C006: then it is stored in location FE.
C008: this loads the Y register with 09.
C00A: now the A register is loaded with 10.
C00C: the LOW byte comes from location FD and the HI byte from location FE. Then the computer forms the address 0400. After that, it adds the value of the Y register to the address, ending in the address 0409. The A register is then compared to the value of this location.

That was the last instruction for this chapter, so we now have a problem to test your knowledge on the compare instructions.

The A register contains an unknown value. When tested against the number 34 the Carry flag is set. When compared to 55 the Negative flag is set. Select one of the following answers:

A. The A register contains a value less than 34.
B. The A register contains a value between 35 and 54.
C. The A register has a value greater than 55.

The answer is B. If you didn't get the answer, just read the first two pages of this chapter before proceeding to the next chapter.

# CHAPTER TEN — CONDITIONAL BRANCHING

Conditional branches are like BASIC's IF . . . THEN . . . statement. They carry out operations such as 'if the Negative flag is set branch to'. There are eight different conditional branch instructions which are explained in this chapter.

## BCC
## (Branch on Carry Clear)

The BCC instruction will cause a branch if the Carry flag is clear (reset). An example of the BCC instruction is 'BCC$C009'. This instruction would branch to the address C009 if the Carry flag is clear, otherwise it would continue with the next instruction. This kind of branch is called RELATIVE addressing. That means it has a certain range, relative to its present address.

This range is how far it can branch. These branch instructions can branch 80 (decimal 128) locations backward and 7F (decimal 127) locations forward. This means that a branch instruction, BCC$C0F0, at location C000 would be out of range, although if the instruction was BCC$C07F it would be within range. If the instruction BCC$C000 was at location C090 it would be out of range, although if the instruction was BCC$C010 it would not.

You don't have to worry about this, because the assembler will tell you when you are out of range. If you are out of range the assembler will give you the message 'BAD BRANCH'.

The following program compares the A register with a number, and branches if the A register is less than the number:

```
C000 LDA#$20

C002 CMP#$30

C004 BCC$C007

C006 RTS

C007 STA$D020

C00A RTS
```

C000: this loads the A register with 20.
C002: the A register compared to 30, which sets the Negative flag.
C004: a branch is taken to location C007, because the Carry flag is clear.
C006: if the branch wasn't taken the computer would return to BASIC.
C007: this stores the A register in location D020, which changes the border colour.
C00A: this returns the computer to BASIC.

# BCS
# (Branch on Carry Set)

This instruction is the opposite of BCC. It branches when the Carry flag is set:

```
C000 LDX$0286

C003 CPX#$01

C005 BNE$C008

C007 RTS

C008 LDA#$00

C00A STA$0286

C00D RTS
```

In the above example the A register is loaded with the colour of the cursor. This is compared to 01. If it is equal to, or greater than 01, a branch is taken to C007 where it is made equal to 00.

# BNE
# (Branch on Result not Zero)

This instruction causes a branch if the Zero flag is not set. The following example is a time delay which I have used often in machine code games programs:

```
C000 LDX#$00

C002 LDY#$00

C004 DEX

C005 BNE$C004

C007 DEY

C008 BNE$C004

C00A RTS
```

This mightn't seem much of a time delay when you run it, but it executes around 130,000 instructions.

This is how it runs:

C000: loads the X register with 00.
C002: loads the Y register with 00.
C004: the X register is then decremented to a value of FF, which sets the Negative flag and resets the Zero flag.

C005: because the Zero flag has been reset, the branch is taken back to location C004.

C007: this instruction is executed when the X register has been decremented to 00. The actual instruction decrements the Y register.

C008: this causes a branch back to location C004 if the Y register wasn't decremented to 00.

C00A: this returns the computer to BASIC.

# BEQ
# (Branch on Result Zero)

This is the opposite to BNE. It causes a branch when the Zero flag is not set:

```
C000 LDA$028A

C003 BEQ$C006

C005 RTS

C006 LDA#$80

C008 STA$028A

C00B RTS
```

The program tests whether location 028A equals 00. If it does a branch is taken to location C006. If the branch was taken, location 028A is set to 80 which sets the key repeat.

# BMI
# (Branch on Result Minus)

This instruction causes a branch if the Negative flag is set, as in this example program:

```
C000 LDA#$20

C002 CMP#$30

C004 BMI$C007

C006 RTS

C007 LDA#$00

C009 STA$D020

C00C RTS
```

The program tests the A register against 30, and because the A register is less than 30, the Negative flag is set. The branch is then taken to C007 and the border colour is changed.

# BPL
# (Branch on Result Plus)

This is the opposite of BMI. It causes a branch if the Negative flag is clear.

Here is an example:

```
C000 LDX#$30

C002 STA$FB

C004 LDY#$30

C006 CPY$FB

C008 BPL$C00B

C00A RTS

C00B STY$D021

C00E RTS
```

The program changed the colour of the screen, because the branch was taken from location C008 to location C00B.

# BVC and BVS

These commands cause branches depending on the Overflow flag. BVC causes a branch if the Overflow flag is clear. BVS will cause a branch if the Overflow flag is set. There will be more said about the Overflow flag and these instructions in chapter twelve.

It is time for another test. The problem is:

Devise a program that will PRINT the letter A 255 times.

```
C000 LDX#$00

C002 LDA#$41

C004 JSR$FFD2

C007 INX

C008 BNE$C004

C00A RTS
```

OR

```
C000 LDY#$00

C002 LDA#$41

C004 JSR$FFD2

C007 INY

C008 BNE$C004

C00A RTS
```

Either of the above programs will work. If you understand how they work then continue with the next chapter. Otherwise perhaps you'd better revise the previous two chapters.

# CHAPTER ELEVEN — STORING REGISTERS ON THE STACK

As you already know, we are restricted greatly when working in the machine code on the Commodore 64 by only having three registers. You are probably thinking 'but we can store them in memory, or even transfer them.' Even transferring or storing the registers may not be possible or practical in some situations.

We get around this by storing registers on the stack. This chapter shows you how to save registers to the stack and how to take them back off.

## PLA and PHA

PHA stands for Push the A register on the stack. When you push the A register on the stack its value is stored in the location pointed to by the Stack Pointer (SP). Then the SP is decremented, so that it points to the next free byte on the stack. For example, if the SP equals 89 and you push the A register on the stack the A register will be transferred to location 0189. The SP will be decremented to equal 88.

PLA stands for Pull the A register off the stack. Once you have stored the A register on the stack you use this instruction to get it back off the stack and into the A register. For example, if the SP equals 92 and you

use the PLA statement, then the A register will be loaded with location 0193. The SP will then be incremented to a value of 93.

Here is an example of saving the A register to the stack, using it for another purpose, and then retrieving it off the stack:

```
C000 LDA#$30

C002 PHA

C003 LDA#$00

C005 STA$D020

C008 PLA

C009 RTS
```

# PHP and PLP

PHP stands for Push the P register onto the stack. PHP has the same effect on the stack as PHA.

PLP stands for Pull the P register from the stack. PLP has the same effect on the stack as PLA.

Here is an example of saving the P register to the stack and retrieving it again:

```
C000 LDA#$00

C002 PHP

C003 LDA#$80

C005 STA$028A

C008 PLP

C009 RTS
```

# TXS and TSX

There are two transfer instructions that I have not yet told you about. The first one is TSX, which stands for Transfer the Stack pointer to the X register. The other one is TXS, which stands for Transfer the X register to the Stack Pointer. Both can be used to make sure that the Stack Pointer doesn't equal zero.

# Notes

* When pushing any numbers on the stack, always remember to take them back off the stack. If you don't take values off the stack the system may crash when it executes an RTS statement.

* Numbers are pulled off the stack in the reverse order that they were put on. For example, if the numbers 1, 2, 3, 4, 5 and 6 are put on the stack in that order, they would be taken off in the following

order: 6, 5, 4, 3, 2 and 1. (This order is referred to as 'last in — first out'.)

Here is a problem for you to ponder before going on to the next chapter:

Transfer *all* registers to the stack, and then retrieve them *all.*

```
C000 PHP

C001 PHA

C002 TXA

C003 PHA

C004 TYA

C005 PHA

C000 PLA

C001 TAY

C002 PLA

C003 TAX

C004 PLA

C005 PLP
```

The first program puts the registers on the stack and the second takes them off.

It isn't too important that you managed to create the above programs, but it is important that you understand them.

# CHAPTER TWELVE — SUBTRACTION AND ADDITION

This chapter deals with the mathematical functions of the 6510 (the 64's microprocessor). The 6510 can only handle subtraction and addition, and both are carried out with the A register. This is why the A register is called the accumulator.

## ADC (Add to Accumulator with Carry)

By now you should know all of the addressing modes. That is, you should know what Zero Page addressing, Absolute addressing and all the rest are. Therefore, I won't be outlining every addressing mode of an instruction any more.

The instruction, ADC, causes a value and the Carry flag to be added to the A register. For example, if the Carry flag is set, the A register has a value of 40 and you use the instruction ADC# $03, the A register will end up with a value of 44. This happens because the computer would add 03 to 40, giving it 43, then it would add the Carry flag to that, giving it a final value of 44.

If the Carry flag had not been set, the value of the A register would have been 43. Before addition we can use the command CLC to Clear the Carry so that an extra 01 isn't added to the final answer.

Here is an example program which adds 02 and 02 together:

```
C000 LDA#$02

C002 CLC

C003 ADC#$02

C005 RTS
```

The previous program added 02 to 02, and of course the answer was 04.

Here is another program. Again it adds 02 to 02, but this time the Carry flag is set:

```
C000 LDA#$02

C002 SEC

C003 ADC#$02

C005 RTS
```

The result of the addition is 05, because the Carry flag was added as well as 02. If the Carry flag was clear and you added 06 to 04 you would get an answer of 0A.

Sometimes it isn't practical to use this sort of addition, i.e. you would rather have a decimal result. It is possible to get the A register to carry out decimal addition by setting the Decimal flag. This is done by using the SED (Set Decimal flag) instruction. When the Decimal flag is set an addition such as 05 plus 07 will equal 12, and not 0C. Before returning to BASIC you must clear the Decimal flag, as the BASIC interpreter will crash if it is set.

Here is an example of Decimal addition:

```
C000 SED

C001 LDA#$07

C003 CLC

C004 ADC#$06

C006 CLD

C007 RTS
```

This will give the A register a value of 13. If the Decimal flag hadn't been set before the addition the A register would have had a value of 0D.

# Notes on ADC

* The Carry flag will be set if an addition exceeds 255 under normal circumstances. However, if the Decimal flag is set, the Carry will be set if an addition exceeds 99.

* The Zero flag will be set if the addition results in zero.

* The Negative flag will be set if the addition results in a number between 80 and FF.

* The Overflow flag will be set if the result of an addition exceeds 7F.

* Always reset the Decimal flag before returning to BASIC, otherwise the computer will crash.

* ADC can support the following modes of addressing:

  IMMEDIATE (ADC# $ZZ)

  ZERO PAGE (ADC$ZZ)

  ZERO PAGE INDEXED (ADC$ZZ.X)

  ABSOLUTE (ADC$ZZZZ)

  ABSOLUTE INDEXED (ADC$ZZZZ.X *or* ADC$ZZZZ.Y)

  INDEXED INDIRECT (ADC($ZZ.X))

  INDIRECT INDEXED (ADC($ZZ).Y)

In the above table ZZ means a two-digit number and

ZZZZ means a four-digit hex number.

# SBC (Subtract from a Register with Borrow)

SBC is used to subtract a value from the A register. If the Carry flag isn't set, an extra 01 is taken from the A register. If the Carry flag is set, the Carry flag is ignored by the subtraction.

Here is an example. Notice we set the Carry flag so that it will be ignored:

```
C000 LDA#$40

C002 SEC

C003 SBC#$20

C005 RTS
```

The result in the A register is 20, because 40-20 = 20. Had the Carry flag been clear the A register would have had a value of 1F, because an extra 01 would have been subtracted.

SBC can also use the decimal mode. When the Decimal flag is set the SBC command subtracts in decimal notation. For example, if the Decimal flag is set and you take 01 from 20 you would get an answer of 19, not 1F.

Here is an example of Decimal subtraction:

```
C000 LDA#$60

C002 SED

C003 SEC

C004 SBC#$15

C006 CLD

C007 RTS
```

The above program subtracts 15 from 60, and because the Decimal and Carry flags are set, the answer left in the A register is 45.

# Notes on SBC

* The Carry flag will be set if the result of a subtraction is Zero or positive.

* The Zero flag will be set if the result is 00.

* The Negative flag is set if the result is less than 00. In that case the seventh bit of the A register will be set.

* The Overflow flag is set if the subtraction is less than -80.

* The following addressing modes are supported by SBC:

  IMMEDIATE (SBC# $ZZ)

  ZERO PAGE (SBC$ZZ)

  ZERO PAGE INDEXED (SBC$ZZ.X)

  ABSOLUTE (SBC$ZZZZ)

  ABSOLUTE INDEXED (SBC$ZZZZ.X *or* SBC$ZZZZ.Y)

  INDEXED INDIRECT (SBC($ZZ.X))
  INDIRECT INDEXED (SBC($ZZ).Y)

Here's a problem to test your understanding of this chapter.

Make a program that will add the Y register to the X register, and add that total to the A register. You may use the following instructions:

PHA, PLA, CLC, RTS, TXA, TAX, STY$FE, STX$FE, ADC$FE.

You may use each one more than once. If you can't think of a program using these instructions, work out your own.

Here's the answer:

```
C000 PHA
C001 TXA
C002 STY$FE
C004 CLC
C005 ADC$FE
C007 TAX
C008 PLA
C009 STX$FE
C00B CLC
C00C ADC$FE
C00E RTS
```

Note that this program isn't the only answer to the problem. If your program is different to the above program and it works, then it is probably just as good.

# CHAPTER THIRTEEN — SHIFTING AND ROTATION

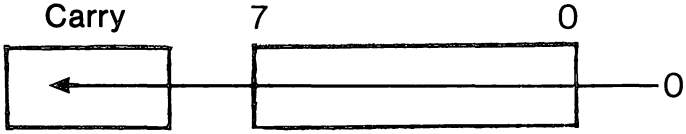
This chapter, and the one following it, rely heavily on a knowledge of binary numbers. With the following lines, the assembler will convert the values of the registers to binary numbers. Load your assembler and enter the following:

```
8050 DEC=PEEK(780):GOSUB8900:BA$=BI$:GOS
UB3000:AR$=RIGHT$(HEX$,2)
8060 DEC=PEEK(781):GOSUB8900:BX$=BI$:GOS
UB3000:XR$=RIGHT$(HEX$,2)
8070 DEC=PEEK(782):GOSUB8900:BY$=BI$:GOS
UB3000:YR$=RIGHT$(HEX$,2)

8100 PRINT"{CUR DN}AR= ";BA$:PRINT"{CUR
DN}XR= ";BX$:PRINT"{CUR DN}YR= ";BY$

8900 BI=DEC:BI$="":FOR R=7TO0STEP-1:IFBI
<(2^R)THENBI$=BI$+" 0":GOTO8920
8910 BI$=BI$+" 1":BI=BI-(2^R)
8920 NEXT:RETURN
```

After you have typed the lines in, re-save your assembler.

# ASL
# (A register Shift Left)

ASL causes a shift of one bit to the left. Here is a diagram:




When you use this instruction a 0 enters bit 0, bit 0 enters bit 1, bit 1 enters bit 2, and so on, until bit 7 enters the Carry flag.

Here is an example showing a value before and after an ASL instruction:

Before rotation:

Carry flag = 0, memory to be rotated = 10010110

After rotation:

Carry flag = 1, memory = 00101100

You should be able to see that the memory was shifted one bit to the left and the Carry received the seventh bit.

Here is a program that shifts the A register from a value of 81 (10000001) to 2 (00000010) and sets the Carry flag:

```
C000 LDA#$81

C002 ASL

C003 RTS

C004 LDY#$03

C006 RTS
```

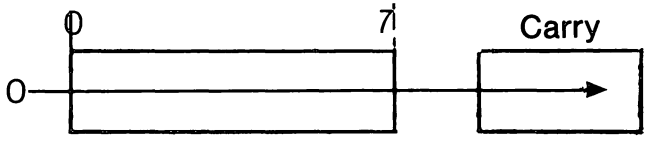
The above program sets the Carry flag because bit 7, which was 1, was shifted into the Carry flag.

The next program shifts a memory location. The X register holds the value before the shift, and the Y register holds the value after the shift:

```
C000 LDX#$AE

C002 STX$FE

C004 ASL$FE

C006 LDY$FE

C008 RTS
```

# LSR
# (Logical Shift Right)

This does the opposite of ASL. It shifts memory one bit to the right. Here is a diagram of LSR:




The whole byte is shifted one bit right and a 0 enters bit 0, while bit 7 enters the Carry flag. Here is an example program that shifts the A register from a value of A7 (10100111) to 53 (01010011) and sets the Carry flag:

```
C000 LDA#$A7

C002 LSR

C003 RTS
```

* LSR and ASL both support the following modes of addressing:

  ACCUMULATOR (A REGISTER) ADDRESSING (ASL, LSR)

  ZERO PAGE ADDRESSING (LSR$ZZ, ASL$ZZ)

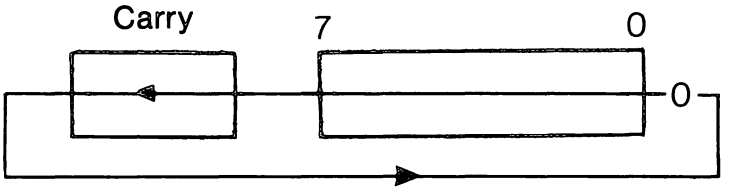
  ZERO PAGE INDEXED ADDRESSING (LSR$ZZ.X, ASL$ZZ.X)

ABSOLUTE ADDRESSING (LSR$ZZZZ, ASL$ZZZZ)

ABSOLUTE INDEXED ADDRESSING (LSR$ZZZZ.X, ASL$ZZZZ.X)

# ROL
# (Rotate One bit Left)

ROL is exactly the same as ASL, except instead of a 0 entering bit 0, the Carry flag is shifted there. Here is a diagram:




As the diagram shows, the Carry flag enters bit 0 and the whole byte is shifted one bit to the left. Here is an example program that shifts the A register from a value of 81 (10000001) to 3 (00000011). Notice that the Carry flag is set before the rotation.

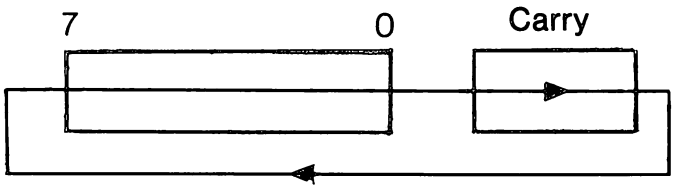
```
C000 LDA#$81

C002 SEC
```

```
C003 ROL

C004 RTS
```

The A register ended up with 3, because the Carry flag entered bit 0, the register was shifted left, then bit 7 entered the Carry flag.

# ROR
# (Rotate One bit Right)

This instruction is the opposite of ROL. It causes the Carry flag to enter bit 7, the memory to be shifted right and bit 7 to enter the Carry flag. Here is a diagram:




In the next program, FF (11111111) is rotated to 7F (01111111), and the Carry is also set as a result:

```
C000 LDA#$FF

C002 CLC
```

```
C003 ROR

C004 RTS
```

* Both ROL and ROR support the same modes of addressing as ASL and LSR.

It is time again for you to solve a problem.

Before a rotate or shift takes place the A register equals 78 (01111000) and the Carry flag is set. After the rotation the A register has a value of F1 and the Carry flag is clear. Which instruction did I use: ROL, ROR, ASL or LSR?

The instruction I used was ROL. Here is the actual program I used:

```
C000 LDA#$78

C002 SEC

C003 ROL

C004 RTS
```

If you didn't get ROL as the answer, but can understand your mistake, then continue. If, however, you are unsure please re-read this chapter very carefully as it covers a great deal in a small amount of text.

# CHAPTER FOURTEEN — LOGICAL INSTRUCTIONS

Logical instructions change the bits of a byte. Unlike most of the instructions we have studied so far, they may store values in bit. This chapter explains every logical instruction on the 6510.

## AND
## (AND Memory with the A Register)

This does the same as BASIC's AND instruction. The A register is the only register that can use the AND instruction. The actual instruction compares each bit in a number with each bit in another number. If two bits are set (1) the result will have that bit set. Otherwise it will be reset (0).

Here is an example:

```
10111010
10001011
--------
10001010
```

The two numbers above the line are being ANDed. The number below the line is the result. Notice that the result has bits set that were set in both of the numbers that were ANDed. When you use the AND instruction in assembly language the A register always ends up with the result.

Here is an example in which 45 (010000101) is ANDed with CE (11001110):

```
C000 LDA#$45

C002 AND#$CE

C004 RTS
```

The AND instruction is useful in turning certain bits off. For example to turn bit 0 off you would AND the number with FE (11111110). If you wanted to turn bits 7 and 0 off you would AND the number with 7E (01111110).

# Notes on AND

* If the result is between 80 and FF the Negative flag will be set.

* If the result is 00 then Zero flag will be set.

* The A register supports the following modes of addressing for the AND instruction:

IMMEDIATE (AND# $ZZ)
ZERO PAGE (AND$ZZ)
ZERO PAGE INDEXED (AND$ZZ.X)
ABSOLUTE (AND$ZZZZ)
ABSOLUTE INDEXED (AND$ZZZZ.X) *or*
(AND$ZZZZ.Y)
INDEXED INDIRECT (AND($ZZ.X))
INDIRECT INDEXED (AND($ZZ).Y)

# ORA
# (OR memory with the A register)

This operates in the same way as BASIC's OR instruction. Two numbers are compared bit by bit. If either or both of the bits are set, that bit will be set in the result.

Here is an example:

```
10111001
00100101
--------
10111101
```

The number below the line is the result of the OR. OR can be used to turn certain bits of a byte on.

Here is a program that turns the seventh bit of location 028A on:

```
C000 LDA$028A

C003 ORA#$80

C005 STA$028A

C008 RTS
```

The previous program turns the key repeat on, because bit 7 controls key repeat, 1 = on, 0 = off. The A register always gets the result of an ORA.

# Notes on ORA

* If the result is between 80 and FF the Negative flag will be set.

* If the result is 00 the Zero flag will be set.

* The A register supports the following modes of addressing for ORA:

  IMMEDIATE (ORA# $ZZ)
  ZERO PAGE (ORA$ZZ)
  ZERO PAGE INDEXED (ORA$ZZ.X)
  ABSOLUTE (ORA$ZZZZ)
  ABSOLUTE INDEXED (ORA$ZZZZ.X) *or*
  (ORA$ZZZZ.Y)
  INDEXED INDIRECT (ORA($ZZ.X))
  INDIRECT INDEXED (ORA($ZZ).Y)

# EOR (Exclusive-Or with the A register)

This is exactly the same as ORA, with one difference. If the A register and the value it is tested against both have the same bit set that bit will be a 0 in the result.

Here is an example:

```
11001011
01010010
--------
10011001
```

The result is the number below the line. From the example you should be able to see the difference between EOR and ORA.

Here is a program that switches the screen colour each time you run it.

```
C000 LDA$D021

C003 EOR#$07

C005 STA$D021

C008 RTS
```

The program works because each time it is run it inverts bit 0, 1 and 2.

# Notes on EOR

* If the result of an EOR is between 80 and FF the Negative flag will be set.

* If the result is 00 the Zero flag will be set.

* The A register supports the following modes of EOR:

IMMEDIATE (EOR# $ZZ)
ZERO PAGE (ORA$ZZ)
ZERO PAGE INDEXED (ORA$ZZ.X)
ABSOLUTE (ORA$ZZZZ)
ABSOLUTE INDEXED (ORA$ZZZZ.X) *or*
(ORA$ZZZZ.Y)
INDEXED INDIRECT (ORA($ZZ.X))
INDIRECT INDEXED (ORA($ZZ).Y)

# BIT
# (Test bits in memory with a register)

Unlike most of the instructions we have looked at so far, this instruction does not affect registers or memory. When you use this instruction the memory's seventh bit goes to the Negative flag and its sixth bit goes to the Overflow flag. The A register is then ANDed with the memory, and if the result is 00 the Zero flag is set. The result of this AND is not stored anywhere.

Here is an example in which the A register is BITed with location FB:

```
C000 LDX#$8A

C002 STX$FB

C004 LDA#$89

C006 BIT$FB

C008 RTS
```

The BIT command has only two modes of addressing; Zero Page and Absolute. It can usually only serve one purpose which is to test the status of a byte of memory.

BIT was the last logical instruction to be covered, so we have come to the end of yet another chapter. Here is a problem to test your knowledge of logical instructions:

I wrote a program to lead the A register with A9 (10101001). It then performed a logical instruction and the A register ended up with 08 (00001000). What was the logical instruction performed in the program?

```
C000 LDA#$A9

C002 EOR#$A1

C004 RTS
```

The program above is the same as the one in the problem. As you can see the answer to the problem is EOR# $A1. This diagram will show you how it worked:

```
A register =  10101001
      data =  10100001
A register =  00001000
```

# CHAPTER FIFTEEN — INTERRUPTS

There is yet another register we have not yet discussed yet. It is called the PC (Program Counter). The PC works independently, that is it does everything on its own. The PC keeps track of which address the computer is at. For example, if the computer was executing an instruction at address C000, the PC would equal C000.

Interrupts, as their name suggest, interrupt the normal flow of a program. Every 1/60th of a second the Commodore is interrupted. It is interrupted to update the clock and scan the keyboard.

We can use this to our advantage. We can change the interrupt so that it jumps to our routines every 1/60th sec. We can do this by changing the vector that the interrupt jumps through.

A vector is two bytes which point to an address and are in LOW-HI byte form.

The vector for this interrupt (IRQ-Interrupt Request) is at locations 0314 and 0315. This vector normally points to location EA31. When we change the vector to point to our routine we must end our routine with 'JMP$EA31'. This will ensure that the keyboard will be scanned and the clock updated.

Here is a program to change the IRQ vector to point to location C010:

```
C000 SEI

C001 LDA#$10

C003 STA$0314

C006 LDA#$C0

C008 STA$0315

C00B CLI

C00C RTS
```

Make sure you don't run this program before we write a routine at location C010. You would have probably noticed the SEI and CLI instructions in the program. SEI sets the IRQ disable. That means that when the IRQ flag equals 1, interrupts are ignored. The CLI instruction clears the IRQ flag and enables interrupts.

Here is the routine that we will run using the IRQ routine:

```
C010 INC$D020

C013 JMP$EA31
```

Now, after you have typed in the above, run the assembly language at location C000. The screen should start flickering. This happens because the screen is changing colour so rapidly that your eyes can't keep up with it.

# Note

While you are changing the IRQ vector, *always* set the IRQ flag. After you have changed the vector you can then use the CLI command to clear the IRQ flag.

# BRK
# (Force Break)

BRK, like IRQ, is an interrupt. There is, however, a difference. BRK is an instruction. When you use the BRK instruction, a jump is taken through a vector at locations 0316 and 0317.

We can change the vector to point to any location where we have a routine, but first we must learn more about BRK.

When a BRK instruction is executed, it jumps through its vector to yet another routine. It is then returned by the RTI instruction, which stands for Return from Interrupt. RTI is exactly the same as RTS, except it returns from interrupts, not subroutines.

When the RTI instruction is executed the PC is taken off the stack and it is incremented twice.

The fact that the PC is incremented twice means that it won't return to the next address, but to the one after that. That means that we must fill up the location after the BRK. We can fill this location with a NOP instruction. NOP stands for No Operation, this

instruction does absolutely nothing except take up space.

This shows how BRK works:

C000 BRK (This causes the computer to go through the vector.)

C001 NOP (This instruction is "skipped".)

C002 . . . (The computer returns to this instruction after an RTI instruction)

The following program changes the BRK vector and then uses the BRK instruction:

```
C000 LDA#$10

C002 STA$0316

C005 LDA#$C0

C007 STA$0317

C00A BRK

C00B NOP

C00C RTS
```

Don't run it until you enter the following routine:

```
C010 INC$0286

C013 RTI
```

Each time you run the program at location C000 the cursor colour will change. This happens because the BRK increments location 0286, which holds the current cursor colour.

There is no test for this chapter as I think you should read it twice anyway. When dealing with interrupts you have to be very careful.

You have now learned every command that the 6510 microprocessor offers. Now we get to the good bits; learning how to use that knowledge.

# CHAPTER SIXTEEN — PROGRAM CREATION

By now you know the C64's machine code instructions. There is only one more thing to learn, how to put everything you know together to form practical programs.

One of the best ways to learn this is to carefully study completed programs such as the ones in this chapter.

We have two games programmes. Both games are a mixture of BASIC and machine code. The BASIC part sets up sprites and the like.

The first one is called PUB SQUASH. It is modelled on the very first arcade game made in 1976. The bat is controlled with the F1 and F3 keys.

Here is the BASIC listing:

```
Ø REM PUB SQUASH BY ROSS SYMONS,198+
1Ø IFPEEK(49152)<>165THENLOAD"GAME1 (2)"
,8,1,1
2Ø GOSUB1ØØØ:REM SET UP SPRITES
3Ø PRINT"{CLR}":FOR P=1Ø24TO1Ø54
4Ø POKEP,16Ø
5Ø POKE96Ø+P,16Ø
6Ø NEXT
7Ø FOR P=1Ø94TO2ØØØSTEP4Ø
8Ø POKEP,16Ø
9Ø NEXT
```

```
100 PRINT"{CUR DN}PRESS ANY KEY TO PLAY"
:POKE198,0:WAIT198,1:POKE251,0:POKE252,0
:SC=0
110 PRINT"{CUR UP}                              "
120 GOSUB300:REM PRINT SCORE ETC...
130 POKE679,1:POKE680,255
140 SYS49152:POKE251,PEEK(251)+1
150 IFPEEK(251)<5THENPOKEV,253:POKEV+1,5
0+INT(RND(1)*200)+1:GOTO120
160 GOSUB300
170 GOTO100
300 SC=SC+PEEK(252):PRINT"{HOME}{CUR DN}
{CUR L}{CUR L}{CUR L}{CUR L}{CUR L}{CUR
L}{CUR L}{CUR L}{CUR L}BALLS {CUR DN}{CU
R L}{CUR L}{CUR L}{CUR L}{CUR L}{CUR L}M
ISSED{CUR DN}{CUR L}{CUR L}{CUR L}{CUR L
}{CUR L}";PEEK(251)
310 POKE252,0:PRINT"{CUR DN}{CUR DN}{CUR
 DN}{CUR L}{CUR L}{CUR L}{CUR L}{CUR L}{
CUR L}{CUR L}{CUR L}{CUR L}SCORE {CUR DN
}{CUR L}{CUR L}{CUR L}{CUR L}{CUR L}   {
CUR L}{CUR L}{CUR L}";SC
330 RETURN
999 END
1000 POKE53280,0:POKE53281,0
1010 V=53248
1020 POKEV,253:POKEV+1,100
1030 POKEV+39,1:POKE2040,13
1040 FOR P=0TO62
1050 POKE832+P,0
1060 POKE896+P,0
1080 NEXT
1090 POKE863,60:POKE866,60
1100 POKE869,60:POKE872,60
1110 POKE2041,14
```

```
1120 FOR P=0TO59STEP3
1130 POKE896+P,31
1140 NEXT
1150 POKEV+21,3:POKEV+2,24:POKEV+3,100
1160 RETURN
```

If you are using cassettes to store your programs, change line 10 to the following:

```
10 IFPEEK(49152)<>165THENLOAD"",1,1
```

After you have typed out the BASIC program, SAVE it, but *don't* RUN it. Now, using the assembler, type in the following assembly language:

```
C000 LDA$C5

C002 CMP#$40

C004 BEQ$C014

C006 CMP#$05

C008 BNE$C00D

C00A INC$D003

C00D CMP#$04

C00F BNE$C014

C011 DEC$D003
```

```
C014 LDX#$20

C016 LDY#$20

C018 DEY

C019 BNE$C018

C01B DEX

C01C BNE$C016

C01E LDA$D001

C021 CMP#$36

C023 BCS$C02A

C025 LDA#$01

C027 STA$02A7

C02A LDA$D001

C02D CMP#$E0

C02F BCC$C036

C031 LDA#$FF

C033 STA$02A7

C036 LDA$D000

C039 CMP#$FC
```

```
C03B BCC$C042

C03D LDA#$FF

C03F STA$02A8

C042 LDA$D000

C045 BNE$C048

C047 RTS

C048 LDA$D01E

C04B AND#$01

C04D BEQ$C056

C04F LDA#$01

C051 STA$02A8

C054 INC$FC

C056 LDA$D000

C059 CLC

C05A ADC$02A8

C05D STA$D000

C060 LDA$D001

C063 CLC
```

```
C064 ADC$02A7

C067 STA$D001

C06A JMP$C000
```

After you have typed it in, return to the main menu. Now press 5. You should be prompted with a question as to which filing system you are using. Enter D or T, D for disk, T for tape. After this you will be asked for the file name. Enter 'GAME1'. Now you will be asked for the start and end addresses of the program. The start address is C000 and the end address is C062. Disk users will have to rename the program with OPEN15,8,15, "RO:GAME1 (2) = GAME1".

When loading assembly language from BASIC you should have extra parameters after the file name in the LOAD statement. These extra parameters are 1,1. This loads a program back into the memory space it came from.

Our second game pits you, a racing car driver, against the track. The track is constantly changing, and there is a different race every time you play. The F1 and F3 keys are used to steer your way through the course.

Here is the BASIC part of the listing:

```
10 REM**INDI 500**BY ROSS SYMONS,1984
20 IFPEEK(49152)<>165THENLOAD"GAME2 (2)"
,8,1,1
30 GOSUB1000:REM SET UP SPRITE
40 PRINT"{CLR}PRESS ANY KEY TO BEGIN":PO
KE198,0:WAIT198,1
50 FOR P=960TO1020
60 POKEP,INT(RND(1)*253)+2
70 NEXT
80 POKE1020,7:POKE1021,16:POKE1022,0:POK
EV+31,0
90 FOR P=1063TO1364STEP40
100 POKEP,160
110 NEXT
120 FOR P=1623TO2000STEP40
130 POKEP,160
140 NEXT
150 SYS49152
160 IF PEEK(1022)=0THENPRINT"{HOME}YAHOO
!!!YOU MADE IT!!":FORP=0TO2000:NEXT:RUN
170 PRINT"{HOME}BAD LUCK,MAYBE NEXT TIME
":FORP=0TO2000:NEXT:RUN
1000 POKE53280,0:POKE53281,0
1010 V=53248
1020 POKEV,30:POKEV+1,150
1030 POKEV+39,1:POKE2040,13
1040 FOR P=0TO62:READ A
1050 POKE832+P,A
1060 NEXT
1070 POKEV+21,1:POKEV+28,1
1080 RETURN
1090 DATA0,0,0,0,0,0,20,0,0,20,0,0,20,1,
64,20,1,64,42,170,128,42,170,160
1100 DATA42,170,168,21,85,84,42,170,168,
42,170,160,42,170,128,20,1,64,20,1,64
1110 DATA20,0,0,20,0,0,0,0,0,0,0,0,0,0,0
,0,0,0
```

If you are using tapes to store your program, change line 20 to the following:

```
20 IFPEEK(49152)<>165THENLOAD"",1,1
```

*Don't* RUN the program, just SAVE it or it will CRASH.

Here is the assembly language listing:

```
C000 LDA$C5
C002 CMP#$40
C004 BEQ$C014
C006 CMP#$05
C008 BNE$C00D
C00A JSR$C095
C00D CMP#$04
C00F BNE$C014
C011 JSR$C09F
C014 LDX#$20
C016 LDY#$20
C018 DEY
C019 BNE$C018
C01B DEX
```

```
C01C BNE$C016

C01E LDA#$13

C020 JSR$FFD2

C023 LDX#$00

C025 LDA#$1D

C027 JSR$FFD2

C02A LDA#$14

C02C JSR$FFD2

C02F LDA#$0D

C031 JSR$FFD2

C034 INX

C035 CPX#$18

C037 BNE$C025

C039 CLC

C03A PHP

C03B LDX#$00

C03D PLP

C03E ROL$03C0.X
```

```
C041 PHP

C042 INX

C043 CPX#$3C

C045 BNE$C03D

C047 PLP

C048 BCC$C058

C04A LDA$03FC

C04D BEQ$C05F

C04F DEC$03FC

C052 DEC$03FD

C055 JMP$C065

C058 LDA$03FC

C05B CMP#$0D

C05D BEQ$C04F

C05F INC$03FC

C062 INC$03FD

C065 LDX$03FD

C068 LDY#$26
```

```
C06A CLC
C06B JSR$FFF0

C06E LDA#$A6

C070 JSR$FFD2

C073 LDX$03FC

C076 LDY#$26

C078 CLC

C079 JSR$FFF0

C07C LDA#$A6

C07E JSR$FFD2

C081 LDA$03FB

C084 BNE$C087

C086 RTS

C087 LDA$D01F

C08A BNE$C08F

C08C JMP$C000

C08F LDA#$01

C091 STA$03FE
```

```
C094 RTS

C095 INC$D001

C098 INC$D001

C09B INC$D001

C09E RTS

C09F DEC$D001

C0A2 DEC$D001

C0A5 DEC$D001

C0A8 RTS
```

After you have typed out the program, return to the main menu. Now SAVE the program under the file name 'GAME2'. The start address is C000 and the end address is C0B0. Disk users will have to rename the program with OPEN15,8,15, "RO:GAME2 (2)= GAME2".

# APPENDIX A

# USEFUL MEMORY LOCATIONS

The following memory locations are the locations I felt would be useful to the beginner. For a complete guide to the Commodore 64 memory usage buy the Commodore 64 Reference Guide.

| LOCATION | USE |
|---|---|
| 0014-0015 | This is where BASIC stores integer variables while doing calculations. |
| 02B-002C | Pointer to the start of BASIC text (LOW-HI byte form). |
| 002D-002E | Pointer to the start of BASIC Variables. |
| 002F-0030 | Pointer to the start of BASIC Arrays. |
| 0031-0032 | Pointer to the end (+1) of BASIC arrays. |
| 0090 | Kernal input/output Status Word: ST. |
| 00C5 | Current Key pressed, you may load registers with this value or find out which keys are pressed. 64 means no key has been pressed. |
| 00C6 | This holds the number of characters in the keyboard buffer. |
| 00F3-00F4 | This points to the location of the screen colour memory. |

| | |
|---|---|
| 0277-0280 | This is the Keyboard buffer. |
| 0281-0282 | Pointer for the bottom of memory. |
| 0283-0284 | Pointer for the top of memory. |
| 0286 | Holds the current cursor colour. |
| 0289 | This holds the size of the keyboard buffer. |
| 028A | If this is $80 the keys will repeat, otherwise they won't. |
| 030C | Storage for the A register. |
| 030D | Storage for the X register. |
| 030E | Storage for the Y register. |
| 030F | Storage for the SP register. |
| 033C-03FB | Tape Buffer. Disk users may use this for their assembly language programs or sprites. |

# APPENDIX B

# 6510 INSTRUCTION SET

## MCS6510 MICROPROCESSOR

| | |
|---|---|
| **ADC** | Add Memory to Accumulator with Carry |
| **AND** | "AND" Memory with Accumulator |
| **ASL** | Shift Left One Bit (Memory or Accumulator) |

| | |
|---|---|
| **BCC** | Branch on Carry Clear |
| **BCS** | Branch on Carry Set |
| **BEQ** | Branch on Result Zero |
| **BIT** | Test Bits in Memory with Accumulator |
| **BMI** | Branch on Result Minus |
| **BNE** | Branch on Result not Zero |
| **BPL** | Branch on Result Plus |
| **BRK** | Force Break |
| **BVC** | Branch on Overflow Clear |
| **BVS** | Branch on Overflow Set |
| | |
| **CLC** | Clear Carry Flag |
| **CLD** | Clear Decimal Mode |
| **CLI** | Clear Interrupt Disable Bit |
| **CLV** | Clear Overflow Flag |
| **CMP** | Compare Memory and Accumulator |
| **CPX** | Compare Memory and Index X |
| **CPY** | Compare Memory and Index Y |
| | |
| **DEC** | Decrement Memory by One |
| **DEX** | Decrement Index X by One |
| **DEY** | Decrement Index Y by One |
| | |
| **EOR** | "Exclusive-Or" Memory with Accumulator |
| | |
| **INC** | Increment Memory by One |
| **INX** | Increment Index X by One |
| **INY** | Increment Index Y by One |
| | |
| **JMP** | Jump to New Location |
| **JSR** | Jump to New Location Saving Return Address |
| | |
| **LDA** | Load Accumulator with Memory |
| **LDX** | Load Index X with Memory |

| | |
|---|---|
| **LDY** | Load Index Y with Memory |
| **LSR** | Shift Right One Bit (Memory or Accumulator) |
| **NOP** | No Operation |
| **ORA** | "OR" Memory with Accumulator |
| **PHA** | Push Accumulator on Stack |
| **PHP** | Push Processor Status on Stack |
| **PLA** | Pull Accumulator from Stack |
| **PLP** | Pull Processor Status from Stack |
| **ROL** | Rotate One Bit Left (Memory or Accumulator) |
| **ROR** | Rotate One Bit Right (Memory or Accumulator) |
| **RTI** | Return from Interrupt |
| **RTS** | Return from Subroutine |
| **SBC** | Subtract Memory from Accumulator with Borrow |
| **SEC** | Set Carry Flag |
| **SED** | Set Decimal Mode |
| **SEI** | Set Interrupt Disable Status |
| **STA** | Store Accumulator in Memory |
| **STX** | Store Index X in Memory |
| **STY** | Store Index Y in Memory |
| **TAX** | Transfer Accumulator to Index X |
| **TAY** | Transfer Accumulator to Index Y |
| **TSX** | Transfer Stack Pointer to Index X |
| **TXA** | Transfer Index X to Accumulator |
| **TXS** | Transfer Index X to Stack Pointer |
| **TYA** | Transfer Index Y to Accumulator |

If you've mastered **BASIC** programming on your Commodore 64 and are ready to move on to bigger and better things, then this book will show you the way.

Ross Symons first introduces you to the disassembler and explains its use. After making this acquaintance, the complete instruction set for the 6510 (the chip at the heart of your computer) is listed. Each instruction is explained with the aid of a demonstration program which will help even the newcomer to machine code to get to grips with the C64 and extract the most from this powerful micro. Other sections include a discussion of the **KERNAL** operating system, and its applications such as printing, input/output devices, and scanning the keyboard, are also considered.

As an exciting bonus, the book ends with two complete machine code games which show you how to create your own worthwhile, high-speed, animated arcade-like games.

ISBN 0-552-99128-7

00495

9 780552 991285

UK £4.95
NZ $12.95
*AUS $9.95

*Recommended Price Only