29 Assembly Language

Introduction

The computer's 'brain' has its own language, and that language is not BASIC. Every time you run a BASIC program, each line has to be translated before this brain (the computer's *central processor unit*) can understand it all. This translation is accomplished by a device called an *interpreter*, which resides in the computer's memory. The action of this device need not concern you, but it is itself a program written in machine-code, and *machine-code* is the computer's own language.

There are 33 different instructions in machine code, which is about half the number of BASIC instructions available on the Electron. Each of these instructions acts upon one or more of the registers inside the 6502 microprocessor (6502 is the type-number of this processor - it has no significance). A register is just like a byte of memory. The 6502 contains six registers, five of them being 1 byte long, and the last being 2 bytes long. These registers are not a part of the computer's memory map (from location &0000 to &FFFF(; they live an entirely separate existence in the heart of the microprocessor. But the machine-code instructions which control these registers are stored in the computer's memory, in the position on the memory map labelled 'operating system'. These instructions don't look much like intelligible commands, for they are simply binary numbers - 1010100100001010 for example. It is very difficult to program using such low-level instructions; even in hexadecimal they hardly look any better: A9 0A. This is the reason for using *Assembly Language*.

Assembly Language uses a three-letter mnemonic to represent each machine-code instruction. Each mnemonic is a contraction of the action-in-words of that instruction.

Take the instruction given above. One of the registers in the microprocessor is called the accumulator (all the registers will be described in detail in a moment).

A9 means 'load the accumulator'.

The mnemonic for this is LDA, thereby giving you a rough guide to its function, LoaD Accumulator.

The other part of the instruction, 0A, is 10 in decimal. So A9 A0 means 'put 10 in the accumulator', and this is written in Assembly Language as:

```
LDA #10
```

(The hash (#) tells the computer that it is the 10 which is to be put into the accumulator, and not the contents of memory location 10. This will be explained in a moment.)

So, each of the 55 machine-code instructions is assigned a three-letter assembly mnemonic, which enables you the programmer to understand the function of each without having to look it up on a chart.

The Electron has another program in its memory, called an assembler, and this converts the Assembly Language directly into machine-code. During this assembly process, the computer can help you by giving error messages and a listing of the machine-code in hex. (If you were programming the 6502 direct in machine-code there would be no error messages at all - and just try finding a mistake among a few hundred machine code instructions!)

The assembler loads the machine-code into memory, and it can then by run, either as a `CALL` or `USR` from BASIC, or by using `*RUN`.

Registers in the 6502

The 6502 microprocessor has six registers as follows:

Accumulator
The accumulator is the main working register of the processor. Most of the 55 Assembly Language instructions operate on the accumulator, which gained its name from the way that results of arithmetic operations are 'accumulated'. It is an 8-bit register, meaning that it can store and operate upon eight binary digits (one byte). Each bit is designated a number, from 0 for the least significant (rightmost) to 7 for the most significant (leftmost).

Common operations involving the accumulator are:

- Loading it from memory (the locations &0000 to &FFFF).
- Storing its contents in memory.
- Addition or subtraction.
- Logical functions (AND, OR or EOR).
- Shifting its contents left or right.

Index registers X and Y
The two index registers are each 8-bits long, and are used for the following:

- To be added to the address used by an instruction. This is called *indexing*.
- As general purpose registers for various counting or short term memory duties.
- In addition to the above, both the accumulator and the two index registers are used by the Electron to pass parameters to operating system subroutine calls. This will be explained later.

Program counter
The program counter is the only 16-bit register, and it holds the memory address of the next instruction to be executed.

Operations involving the program counter are:

- Jump and branch instructions which alter the contents of the PC and thereby divert the flow of the program. (Much like GOTO in BASIC.)

Stack pointer
The stack pointer is an 8-bit register, with a ninth bit on the most significant end which is always set to 1. It is an address pointer which gives the location in memory of a special kind of data-structure used by computers called the *stack*. It can point to addresses between &0100 and &01FF. The stack is explained later, but in essence it is a section of memory which has not only a position, but also an order. Thus, data which is pushed on to the stack in one order, can only be pulled off it in the reverse order. This sort of memory is called *last in first out* (LIFO). It is used for storing data in which the order is important, e.g. execution addresses of nested subroutines.

Flags register

The flags register is different from all the others in that it operates as seven single-bit registers: N, V, B, D, I, Z, C. Each bit signals a condition in the processor, and certain instructions act upon these conditions (whether that condition is present, true; or is not present, false).

Each bit acts as follows:

Bit N is set to 1 when the last operation produced a negative result. A negative result is signified by the most significant bit of a register being 1 (the sign bit). In the case of the accumulator, a result inside it of, for example, 10010100 would set bit N of the flags register to 1. If the last operation did not produce a negative result then bit N is reset to 0.

Bit V is set to 1 when the last operation overflowed into the sign bit. As stated above, the sign bit is bit 7 in the case of the accumulator, so bit V is set to 1 when there is a carry from bit 6 to bit 7. This is important to know when using twos complement arithmetic, for it means that an error has occurred which must be corrected.

Bit B is set to 1 when the BRK command is used (break). (This command has much the same effect on a machine-code program that ESCAPE has on a BASIC program.)

Bit D, when set to 1, causes the processor to operate in BCD mode (binary coded decimal). When reset to 0, the processor works as normal in binary. BCD is beyond the scope of this book, and need not concern you.

Bit I is the interrupt mask. When it is set to 1, no interrupts are accepted. Interrupts are also beyond the scope of this book.

Bit Z is set to 1 when the last operation produced a zero result.

Bit C is the carry register. It is set to 1 by a carry from the most significant bit of one of the registers, usually the accumulator.

These flags are used by the branch instructions, which direct the flow of the program according to the conditions. For example, BEQ means ‘branch if equal to zero’. The program will branch if the Z bit is set to 1. If not it will not branch.

Addressing modes

Take a single instruction - you have seen LDA before. Its function is always to 'load the accumulator', but it may load it in different ways and from different places according to which addressing mode is used.

```
LDA #10
```

means 'load the accumulator with 10'. You know that already. However,

```
LDA 10
```

means 'load the accumulator with the contents of memory location 10.

This is an example of two different addressing modes. The first is *immediate addressing*. The instruction uses the data immediately, without looking for it in memory. The second is *zero-page addressing*. The instruction uses the contents of the address specified. It is called zero-page because the computer's memory is divided up into 256 pages each of 256 bytes. Any address which has its two most significant hex digits are zero is said to be in the zero-page of memory. The zero-page extends from locations &0000 to &00FF.

LDA may also be used with a full 16-bit address:

```
LDA &30A7
```

will 'load the accumulator with the contents of memory location &30A7'. This addressing mode is called absolute. It can access any location in the computer's memory. Notice that the assembler treats numbers as decimal, unless they are preceded by &.

Immediate, zero-page and absolute are not the only addressing modes, although they are the most simple to understand.

```
LDA &1D77,X
```

is an *indexed addressing* mode.

The address used by the instruction is &1D77 plus the contents of index register X. So the accumulator is not loaded from &1D77 but from &1D77+X. Note that the contents of index register X are added to the address, and not to its contents.

The index register used can equally well be Y:

```
LDA &2500,Y
```

(Note: When using machine-code there are several subdivisions of the above indexed addressing mode, but using the assembler takes care of all those for you. However, the assembled machine-code (in hex) will not always be the same for the same indexed instruction.)

Another still more complicated addressing mode is *indirect addressing*:

```
LDA (&1B,X)
```

The address given after the assembler mnemonic, in this case &1B, must be a location in the zero-page of memory (or an error will result). This location is then added to the contents of the X register, to give another location in the zero-page. The contents of this new location, and the contents of the location above it, together supply the full 16-bit address of the location from which the accumulator is loaded. So, if &1B+X contains &AA, and &1B+X+1 contains &BB, then the accumulator will be loaded with the contents of memory location &AABB.

The above operation is called *pre-indexed indirect addressing*; the indexing is the addition of the X register, and the direction is the use of the two consecutive locations at the intermediate address as an address pointer to the actual location used. It is called pre-indexed indirect because the indexing is done before the indirection. All pre-indexed indirections must use index register X.

Post-indexed indirect addressing is written in Assembly Language as follows:

```
LDA (&27),Y
```

In this addressing mode, the indirection occurs first. The address given after the assembler mnemonic, in this case &27, must again be a location in the zero-page of memory. The contents of this location and the contents of the location above it together give a 16-bit address. To this 16-bit address is added the contents of index register Y, and this final address is the location from which the accumulator is loaded. All post-indexed instructions must use index register Y.

The above examples show the complete range of addressing modes which can be used with the instruction LDA. However, there are three more important addressing modes which are used with certain other instructions.

All of the branch instructions use a *relative addressing* mode. BEQ was mentioned in the description of the flags register; it means 'branch if equal to zero'. A branch is an instruction which has an *offset*:

```
        ZZZ data
        BEQ Label
        AAA data
        BBB data
.Label  CCC data
        DDD data
```

In this fragment of program, the triple-letters can be assembler instructions. When a program is running, the program counter is incremented one step at a time to point at the next location which is to be executed. In this example, when BEQ Label is being executed the program counter will point to the line containing the instruction marked AAA. If the result of checking the Z flag is that the previous operation did not produce a negative result then execution will continue at the line containing AAA. If the previous operation did give a zero result then the program counter is incremented until it points at the line marked Label. This program illustrates the use of labels in assembler. They can take any name you choose (subject to the same limitations as a BASIC variable name), and are signified by the fact that they must always start with a full stop.

Branch instructions may branch to labels either forwards or backwards, but not too far. The actual distances are 128 bytes backwards or 127 bytes forwards; but remember that these are measured from the next instruction following the branch, and that each instruction may be either 1, 2 or 3 bytes long. The assembler will soon tell you if you have an address or label out of range.

The next addressing mode is *accumulator addressing*, which is used by only four instructions in the 6502 set. These are ASL, LSR, ROL and ROR and their action is explained in the reference section. In essence, they shift the bits of a memory location of the accumulator to the left or right.

```
ASL &760
```

means shift the contents of memory location &760 one bit to the left. In order to apply this instruction to the accumulator, the accumulator's own addressing mode is used:

```
ASL A
```

means shift the contents of the accumulator one bit to the left. Look up the four instructions in the reference section for more information.

The final addressing mode which you need to know about is the simplest. Certain instructions, such as BRK (break) do not need any data or memory reference at all. These are called *implied* instructions and they carry out a simple task, usually on one of the registers; for example CLC meaning 'clear the carry flag'.

Addressing mode	**Examples**	
Immediate	LDA #68	LDA #number
Zero-page	LDA &9B	LDA address
Absolute	LDA &8E17	LDA address
Indexed	LDA &A06C,Y	LDA Table,X
Pre-indexed indirect	LDA (&72,X)	LDA (pointer,X)
Post-indexed indirect	LDA (&00),Y	LDA (zero),Y
Relative	BEQ Repeat	BNE Loop
Implied	CLC	BRK
Accumulator	LSR A	ROL A

The examples on the right show the assembler mnemonics used not with specific addresses, but with BASIC variables. You will find out that this is a good way of writing assembler subroutines which are to be called from within BASIC programs by CALL or USR.

One final point about addressing modes. The JMP (jump) instruction is the only one which allows straight indirect addressing(non-indexed). JMP is very similar to BASIC's GOTO. It can take a full 16-bit address and place this value in the program counter - hence the program jumps to a new execution address. It is usually used with a label, just like branch, but without the restriction on distance. In absolute mode it would look like this:

```
JMP Label
```

If you wish to use it in indirect mode then simply enclose the address in brackets:

```
JMP(&21A7)
```

It will then use the contents of the two consecutive locations at &21A7 as an address pointer to the location to which it will jump.

```
                JMP &21A7
                -
                -
                -
&21A7           &32
                &76
                -
                -
&3276           continue execution.
```

Entering assembly mnemonics

This section tells you how to write Assembly Language subroutines, and how to call them from BASIC. You may find it worthwhile, now that you know about the 6502 processor's make-up, to read all of the assembly mnemonic definitions. You will then be able to understand much more clearly the capability of the processor, and what the short programs in this section are doing.

Sections of Assembly Language are entered as part of a BASIC program, separated from the BASIC part by the square brackets [and]. The general structure of a program containing an assembler routine is:

```
10 REM BASIC Program
100 [
110 \ Start of assembler mnemonics
200 ]
210 REM BASIC program continues
```

Notice that remarks in the Assembly Language section are signalled by a backslash \. The assembler then knows to ignore them.

Before the routine can be assembled, the computer must be told where it is to be put in computer memory. So the first line of the BASIC part must allocate some memory for this purpose, so there are two ways in which you can do this.

On entering the assembler routine, you assign to the resident integer variable P%, the value you choose to be the address of the first instruction of the assembled machine code. P% is the 'pseudo program counter', used by the assembler, to calculate addressed for branch and jump instructions and as the pointer for the assembled codes. (When O% is not being used).

The two methods for doing this are:

(i) By direct assignment: P% = &2000 for example. The problem with direct assignment is that you have to ensure that the memory location chosen is available for use.

The second method gets round this problem.

(ii) By using BASIC DIM instruction. This takes the form DIM P% 100. Note the use of spaces, and no commas or brackets, to distinguish it from an array dimension, DIM P% 100 allocates 101 bytes of memory for the machine-code, which will be stored along with all the BASIC variables above LOMEM. The number used with the DIM instruction must be large enough so that sufficient space is reserved to hold all the code, but not so large as to overlap other items in the memory.

An even better way in which to use DIM is: DIM Q% 100 followed by P%=Q%. DIM is a convenient way of reserving space for machine code routines. No check is made to prevent the assembled code from overrunning the space reserved for it.

Assembly

To get the computer to assemble the routine into machine-code, you simply RUN the program. To complete the assembly, the program has to be RUN twice. The reason for this will become clear in a moment. The assembler pseudo-operator OPT controls the listing and error output generated on assembly. This operator must be placed in the assembler routine, usually at the start, and is followed by a number from 0 to 3 which causes the following outputs:

OPT0 No errors printed, no listing given.
OPT1 No errors, but a listing is given.
OPT2 Errors are printed, but no listing.
OPT3 Both errors and a listing are given.

The listing given is of the machine-code, in hexadecimal. The errors are printed as messages on the screen.

Here's an Assembly Language routine:

```
10 DIM Q% 100
20 P% = Q%
30 [OPT 3
40 LDA &70
50 CMP #0
60 BEQ Zero
70 STA &72
80 .Zero RTS
90 ]
```

When you RUN this program, the computer will print a listing, and then the message:

```
No such variable at line 60
```

Routines which have forward references to labels (Zero is referred to on line 60 when the assembler has not yet come across it) will always generate an error. The answer to this is to inhibit errors the first time through by using OPT0, and then to RUN a second time to generate the complete code. This is called two-pass assembly.

The way to do this is to enclose the routine in a FOR . . . NEXT loop as follows:

```
10 DIM Q% 100
20 FOR I = 0 TO 3 STEP 3
30 P% = Q%
40 [OPT I
50 LDA &70
60 CMP #0
70 BEQ Zero
```

```
80 STA &72
90 .Zero RTS
100 ]
110 NEXT
```

On the first time through the loop, I=0 and so there will be no listing and no error reported. This run allows the computer to identify the forward referenced label. The second time through the loop, I=3 and hence a list of compiled code is produced, along with any programming errors. Note that the assignment statement P% = Q% is enclosed within the loop so that it is reset before each pass.

On running the program, you will see a listing of the assembled machine-code alongside the Assembly Language mnemonics:

```
>RUN
0E75                OPT I
0E75 A5 70          LDA &70
0E77 C9 00          CMP #0
0E79 F0 02          BEQ Zero
0E7B 85 72          STA &72
0E7D 60             .Zero RTS
```

This means that the mnemonics have been successfully assembled, and the corresponding machine-code has been loaded into addresses &0E75 to &0E7D. &A5 is stored in location &0E75, &70 in location &0E76, &C9 in location &0E77, and so on to 60 which is stored in location &0E7D. This is nine bytes of machine-code in all.

This routine has not yet been executed. To do that, a CALL from BASIC is required:

```
CALL Q% RETURN
```

Nothing is printed on the screen when you do this, and that's because the program is trivial; it merely loads a byte from memory location &70 into the accumulator, and if it isn't zero it is stored in memory location &72. There are some points to note about the structure of the Assembly Language routine:

- When a label is assigned to a line, as at line 90, it must be preceded by a full stop. When the label is called by an instruction, as at line 70, there must be no full stop.
- Most Assembly Language routines end with RTS (return from subroutine) which transfers control back to the BASIC interpreter.
- The above routine uses two locations in the zero page of memory. Only locations &70 to &8F in the zero page may be used by your own programs; all the remainder is taken up by the Operating System's variables, and BASIC's workspace.

Execution by USR

USR is similar to a BASIC FN (function); it gives a single value.

The format is:

R% = USR(Z)

where Z may be a label pointing to the first assembler mnemonic, or the address of the first instruction in machine-code. A label is easier to use since it requires no knowledge of where the machine-code is placed in memory. When R% = USR(Z) is executed, the least significant byte of each of the BASIC integer variables A%, X% and Y% is placed into the accumulator, X register, and Y register respectively. The least significant bit of C% is placed in the carry flag (bit C of the flags register). A%, X%, Y% and C% can therefore be used to initialise the 6502 registers before entry into the assembler routine. Control then passes to the subroutine pointed to by Z. On returning to BASIC (after RTS), the four bytes comprising R% will each contain the contents of one of the 6502 registers, as follows:

R% = PYXA

So R% contains the flags, Y register, X register, and accumulator in that order.

Any or each of these registers may be extracted from R% by setting up a *mask* using AND. To get the accumulator, the least significant byte is required:

Acc = R% AND &FF

Similarly for X, Y:

```
X = (R% AND &FF00) DIV &100
Y = (R% AND &FF0000) DIV &10000
```

To get the flags:

```
10 DIM BLOCK 3
20 !BLOCK = USR(Z)
```

Then (Acc = BLOCK?0, X = BLOCK?1, Y = BLOCK?2), the flags = BLOCK?3.

Here is a program which uses USR. The Assembly Language routine adds the numbers held in X% and A%, and gives the result in the accumulator:

```
10 DIM Q% 100
20 FOR I = 0 TO 3 STEP 3
30 P% = Q%
40 [OPT I
50 .Start STA &80
60 TXA
70 CLC
80 CLD
90 ADC &80
100 RTS
110 ]
120 NEXT
130 INPUT "First number "A%
140 INPUT "Second number "X%
150 Registers%=USR(Start)
160 Sum% = Registers% AND &FF
170 PRINT "Sum of two numbers is ";Sum%
```

When RUN, you will see the following:

```
>RUN
0F0A 8580          OPT I
0F0A 8580          .Start STA &80
0F0C 8A            TXA
0F0D 18            CLC
0F0E D8            CLD
0F0F 6580          ADC &80
0F11 60            RTS
First number       11
```

```
Second number      12
Sum of two number is 23
```

The numbers 11 and 12 are entered by the user, and are stored in the integer variables A% and X%. The USR call tells the computer to start executing the assembly routine from the label Start. Before this happens, the least significant byte of A% is placed in the accumulator, and the least significant byte of X% into the X register. The machine-code corresponding to the assembler mnemonics is now executed in sequence:

STA &80 stores the contents of the accumulator in memory location &80.

TXA transfer the contents of the X register to the accumulator.

CLC clears the carry flag prior to addition. If this is not done then a spurious carry may be added to give an incorrect result.

CLD clears the D flag so that the 6502 is working in binary mode.

ADC &80 adds the contents of the accumulator to the contents of memory location &80, plus the contents of the carry flag; and places the result in the accumulator.

RTS returns control to BASIC.

Back in the BASIC section, Registers% now contains the four 6502 registers' contents. The result is in the accumulator, so the least significant byte of Registers% is placed into Sum%, which is then printed to give the answer. Note that this routine performs only a single-byte addition, so any result given in Sum% will be MOD 256.

Execution by CALL

CALL is similar to a BASIC PROC (procedure).

Here is another addition routine:

```
10 DIM Q% 100
20 FOR I=0 TO 3 STEP 3
30 P%=Q%
40 [OPT I
50 .Start CLC
60 CLD
70 LDA &80
80 ADC &81
```

```
 90 STA &82
100 RTS
110 ]
120 NEXT
130 INPUT "First number "number1%
140 INPUT "Second number "number2%
150 ?&80 = number1%
160 ?&81 = number2%
170 CALL Start
180 Sum% = ?&82
190 PRINT "Sum of two numbers is ";Sum%
```

This program illustrates the use of the indirection operator ?. Indirection operators are very useful when calling assembly routines.

Here is a list to refresh your memory:

?&80 = J%	Will put the least significant byte of J% in location &80.
!&80 = &12345678	Will put &78 in location &80, &56 in location &81, &34 in location &82, and &12 in location &83.
$V% = "FAULT"	Will put the string "FAULT" plus a carriage return (|M) in locations starting at V%. V% must not be in zero page.
S% = ?&80	Will read the contents of location &80 (1 byte) into S%.
R% = !&87	Will read 4 bytes from locations &87 to &8A into R%; &8B being the most significant, &87 the least significant.
R$ = $&2000	Will read a string starting at &2000 into R$.

The addition program shown above has exactly the same effect as the previous example. In this instance though, the two numbers are stored into memory in the BASIC part of the program, and are added and the result stored in the Assembly Language part.

CALL may also be used with parameters, similar to PROC. This takes the form:

```
CALL Start,integer%,decimal,string%,?byte
```

The parameters are separated by commas. Start is a label, but could equally well be a specific address, &2000 for example. The above CALL

shows that any kind of variable may be passed as a parameter: integer, real, string, and single-byte. When a CALL is made, the parameters are assigned to a parameter block, which starts at memory location &600. The format of this parameter block is:

Address	Contents
&600	Number of parameters
&601	1st parameter address (low)
&602	1st parameter address (high)
&603	1st parameter type
&604	2nd parameter address (low)
&605	2nd parameter address (high)
&606	2nd parameter type

There may be any number of parameters, and this number is given in the first byte of the parameter block. Following this, each parameter's address and type is given.

The type is designated by a number:

0	A single byte (e.g. ?location)
4	A 4-byte variable (e.g. Z% or !address)
5	A 5-byte variable (e.g. number)
128	A defined string (e.g. "YES PLEASE") which must end with &D (RETURN)
129	A string variable (e.g. name$)

The way that the parameter block is laid out, it would seem that the best way to access the individual parameters is to use indirect addressing. Unfortunately, the 6502 only allows the zero-page to be used for indirect address pointers, so here is a routine which transfers the addresses from the parameter block into free locations in the zero-page:

```
        LDA &600        \ Check the number of parameters.
        BEQ End         \ If zero then finish.
        STA &70         \ If not then store this number.
        LDX #0          \ Clear the X register.
        LDY #0          \ and the Y register.
.Loop   LDA &601, Y     \ Take high address of parameter
        STA &71,X       \ and store it in zero-page.
        INX             \ Increment X register.
```

```
        INY             \ and Y register.
        LDA &601,Y      \ Take two address of parameter
        STA &71,X       \ and store it in zero-page.
        INX             \ Increment X register
        INY             \ and Y register
        INY             \ twice.
        DEC &70         \ Decrement number of parameters.
        BNE Loop        \ If still not zero then repeat.
.End    RTS             \ Return to BASIC.
```

This routine stores the address of each parameter in zero-page memory starting at location &71. 15 parameter addresses may be stored in this way before the total user zero-page memory is filled. This routine is very useful if the number of parameters passed to a particular Assembly Language subroutine is not always the same, for it will only relocate the addresses of those parameters which exist.

Here this routine is incorporated into another addition program:

```
10 DIM Q% 100
20 FOR I = 0 TO 3 STEP 3
30 P%=Q%
40 [OPT I
50 .Start CLC
60 CLD
70 LDA &600
80 BEQ End
90 STA &70
100 LDX #0
110 LDY #0
120 .Loop1 LDA &601,Y
130 STA &71,X
140 INX
150 INY
160 LDA &601,Y
170 STA &71,X
180 INX
190 INY
200 INY
210 DEC &70
220 BNE Loop1
```

```
230 .End LDX #0
240 STX &2000
250 LDY &600
260 BEQ Finish
270 .Loop2 LDA (&71,X)
280 ADC &2000
290 STA &2000
300 INX
310 INX
320 DEY
330 BNE Loop2
340 .Finish RTS
350 ]
360 NEXT
370 INPUT"First number "one%
380 INPUT"Second number "two%
390 INPUT"Third number "three%
400 CALL Start,one%
410 Sum%=?&2000
420 PRINT Sum%
430 CALL Start,one%,two%,three%
440 Sum%=?&2000
450 PRINT Sum%
460 CALL Start,one%,two%,three%
470 Sum%=?&2000
480 PRINT Sum%
```

The parameter block transfer routine ends at line 240, where the addition routine begins. Notice that the whole routine is CALLed with varying numbers of parameters, just to prove that it works. The result of adding the parameters is given in location &2000. However, as with the previous programs, the result is MOD 256.

Quadruple precision addition

Integer variables are stored in four consecutive bytes of memory. Groups of four bytes can be accessed using !, and can be added together. This is achieved a byte at a time, starting with the least significant, and storing each successive result:

```
10 DIM Q% 100
```

```
20 FOR I = 0 TO 3 STEP 3
30 P% = Q%
40 [OPT I
50 .Start CLC              \ Clear carry for ADC instruction
60 CLD
70 LDX #0                  \ Clear X register
80 LDY #4                  \ Set Y register to 4 as a counter
90 .Loop LDA &70,X         \ Put byte from one% in accumulator
100 ADC &74,X              \ Add byte from two%
110 STA &78,X              \ Store the result
120 INX                    \ Increment X register
130 DEY                    \ Decrement Y register
140 BNE Loop               \ If not zero then repeat
150 RTS
160 ]
170 NEXT
180 INPUT"First number "one%
190 INPUT"Second number "two%
200 !&70 = one%
210 !&74 = two%
220 CALL Start
230 sum% = !&78
240 PRINT"Sum of two numbers is ";sum%
```

This program will work with positive or negative integers.

Multiplication

The 6502 does not have a multiply instruction. Multiplication is achieved by adding and shifting, just like ordinary decimal long-multiplication. As a simple example, take the multiplication of two 4-bit numbers. Such a multiplication can give an 8-bit result:

(i) Test the rightmost bit of the multiplier. If it is zero then add 0000 to the most significant end of the result. If it is 1 then add the number to be multiplied to the most significant end of the result.

(ii) Shift the result one bit position to the right. Repeat (i) for the next bit of the multiplier.

Applying the above to 1101*1001, the rightmost bit of the multiplier (1001) is 1. Therefore 1101 is added to the most significant end of the

result:

1101

Shift the result right one bit position:

01101

The next bit of the multiplier is zero, so 0000 is added to the result, and it is again shifted right.

001101

The next bit is again zero:

0001101

The final bit is 1, so 1101 is added to the result, and the final shift is performed:

01110101

Notice that for 4-bit multiplication, four shifts are required, 8-bit multiplication will require eight shifts, 16-bit multiplication 16 shifts, and so on.

To put the above routine into practice on the 6502, the shift and rotate instructions are used. Here is a program to multiply two 8-bit numbers:

```
10 DIM Q% 100
20 FOR I=0 TO 3 STEP 3
30 P%=Q%
40 [ OPT I
50 .Start CLD
60 LDA #0
70 STA &72          \ Clear 16-bits
80 STA &73          \ for the result.
90 LDY #8           \ Set Y to 8 as a counter.
100 .Loop LSR &71   \ Shift multiplier right one bit.
110 BBC Noadd       \ Test this bit. Branch is zero.
120 CLC             \ Clear carry prior to addition.
```

```
130 LDA &70             \ Load accumulator with number to be
                        \ multiplied.
140 ADC &73             \ Add most significant byte of  result.
150 STA &73             \ Shift result right, with carry from addition.
160 .Noadd ROR &73      \ Decrement counter.
170 ROR &72             \ Repeat if not zero.
180 DEY
190 BNE Loop
200 RTS
210 ]
220 NEXT
230 INPUT"First number "one%
240 INPUT"Second number "two%
250 ?&70=one%
260 ?&71=two%
270 CALL Start
280 Product%=?&72 + 256*?&73
290 PRINT "Product of two numbers is  ";Product%
```

This routine is not the most efficient way of multiplying two bytes together, but it illustrates the method clearly:

Lines 60, 70 and 80 clear the two bytes in memory which will be used for the result of the multiplication. These locations are &72 (result low byte) and &73 (result high byte).

Lines 250 and 260 store the numbers to be multiplied in locations &70 and &71. It doesn't matter which of these is chosen to be the multiplier; the example uses the number in &71.

Line 90 sets the Y register to 8 as a counter. Because this is an 8-bit multiplication, eight shifts are required.

Line 100 shifts the multiplier right one bit position. The rightmost bit falls into the carry where it can be tested.

Line 110 carries out the test. If the C bit is zero then the program branches to NoAdd; if it is 1 then the addition of the number in &70 to the result high byte (&73) takes place.

Lines 120 to 150 accomplish this addition, by clearing the carry bit, loading the accumulator from &70, adding the result high byte, and then storing back in the result high byte.

Line 160, labelled NoAdd, rotates the result high byte right one byte

position. The carry from the addition in line 140 is entered from the left, and the rightmost bit falls into the carry.

Line 170 rotates the result low byte right one bit position. The leftmost bit from the high byte, now in the carry, enters the low byte from the left.

Line 190 decrements the counter, and repeats the above process until the counter is zero.

The program will give the result of multiplying two positive integers, each between 0 and 255. You can see how many instructions it takes just to do this, and can imagine the complexity of a BASIC statement when it is interpreted into machine-code.

A shorter routine to multiply two bytes uses the accumulator as the result high byte, and the multiplier as the result low-byte. As each bit of the multiplier is shifted into the carry to be tested, the leftmost bit of the multiplier location becomes vacant, so allowing the result to be shifted in.

```
.Start    CLD
          LDA #0          \Clear result high byte
          LDY #8          \Set shift counter.
.Loop     ROR &71         \Shift multiplier right one bit.
          BCC Noadd       \Test this bit. Branch if zero.
          CLC             \Clear carry prior to addition.
          ADC &70         \Ask number to be multiplied.
.Noadd    ROR A           \Shift result right, with carry from addition
          DEY             \Decrement counter.
          BNE Loop        \Repeat if not zero.
          ROR &71         \Final shift of result
          STA &72         \Store result high byte.
          RTS
```

Before using this routine, the two bytes to be multiplied are placed in locations &70 and &71. The result appears in &71 (low byte) and &72 (high byte).

To multiply two 4-byte numbers together, the additions and shifts must act on each byte in turn, and the total number of shifts must be 32.

```
.Start    CLD
          LDX #8          \ Clear
          LDA #0          \ eight
.Clear    STA &77,X       \ bytes
```

```
        DEX             \ for
        BNE Clear       \ result.
        LDY #32         \ Set shift counter.
.Loop   LSR &77         \ Shift four bytes
        ROR &76         \ of multiplier
        ROR &75         \ right
        ROR &74         \ one bit.
        BCC Noadd       \ Test this bit. Branch if zero.
        CLC             \ Clear carry prior to addition.
        LDA &70         \ Add
        ADC &7C         \ 4-byte
        STA &7C         \ multiplier
        LDA &71         \ to
        ADC &7D         \ 4-byte
        STA &7D         \ result
        LDA &72         \ and
        ADC &7E         \ store.
        STA &7E         \ "
        LDA &73         \ "
        ADC &7F         \ "
        STA &7F         \ "
.Noadd  LDY #8          \ Shift
.Shift  ROR &77,X       \ eight bytes
        DEX             \ of result
        BNE Shift       \ right
        DEY             \ one bit.
        BNE Loop        \ Repeat if not zero
        RTS
```

Before using this routine, the two numbers to be multiplied must be placed in !&70 and !&74. The result appears in the four bytes from &78 (least significant) to &7B (most significant), and is accessed as !&78. This routine will work with both positive and negative integers.

Division

Division is accomplished as the reverse of multiplication. 8-bit multiplication gave a 16-bit result, so, for division, a 16-bit numerator and 8-bit denominator will give an 8-bit result. The numerator is stored in two bytes of memory. It is shifted left one bit position and the

numerator high byte is then loaded into the accumulator. If the shift produced a carry then a 1 is shifted left into the result, the denominator is subtracted from the accumulator, and the accumulator contents are then stored in the numerator high byte. If the shift did not produce a carry then the denominator is subtracted from the accumulator in any case. If this subtraction produces a carry then a 1 is shifted left into the result and the accumulator contents are stored in the numerator high byte. If no carry, then 0 is shifted left into the result.

This whole process is repeated eight times. The division program is as follows:

```
10 DIM Q% 100
20 FOR I = 0 TO 3 STEP 3
30 P% = Q%
40 [OPT I
50 .Start CLD
60 LDY #8                \ Set shift counter.
70 .Loop ASL &72         \ Shift numerator
80 ROL &73               \ left one bit.
90 LDA &73               \ Load accumulator high byte.
100 BCC Label            \ Test carry produced by shift.
110 SBC &71              \ Subtract denominator and
120 STA &73              \ store in numerator high byte.
130 SEC                  \ Set carry prior to shifting into result
140 JMP Shift            \ Go to Shift.
150 .Label SEC           \ Set carry prior to subtraction.
160 SBC &71              \ Subtract denominator
170 BCC Shift            \ and test carry.
180 STA &73              \ Store in numerator high byte.
190 .Shift ROL &70       \ Shift either 0 or 1 into result.
200 DEY                  \ Decrement counter.
210 BNE Loop             \ Repeat if not zero.
220 RTS
230 ]
240 NEXT
250 INPUT "Numerator "numerator%
260 INPUT "Denominator "denominator%
270 P%=&71
280 [OPT 3
290 EQUB denominator%    \Store denominator at location &71.
300 EQUW numerator%      \Store numerator at locations &72 and &73.
```

```
310 RTS
320 ]
330 CALL Start
340 PRINT "Quotient is ";?&70
350 PRINT "Remainder is ";?&73
```

In this routine, the denominator is stored at location &71, and the numerator in two bytes &72 and &73. The result appears in &70, and any remainder is left in &73. Remember, this is a 16-bit by 8-bit division, so the denominator may not be greater than 255 and the numerator not greater than 65025 (2552) to give a valid result (the result must be 255 or less).

The short routine from lines 280 to 320 is used to store the data in memory, and contains some instructions which you have not yet seen or used. EQUB and EQUW are in the same class of instruction as OPT, in that they are used in the Assembly Language part of the program but are not assembly instructions. They are used simply to store data at the location(s) at which they appear when assembled into machine-code. You will see this clearly when you RUN the above program. After you have typed in the numerator and denominator you will see a listing of the machine-code from &0071 to &0074.

There are in fact four EQU instructions:

EQUB stores a byte of data.
EQUW stores a word of data (2 bytes).
EQUD stores a double-word of data (4 bytes).
EQUS stores the ASCII representation of a string.

EQUS is illustrated in the next section on error handling in Assembly Language.

Notice in the program example above how putting P% equal to &71 enables the denominator to be stored in &71 using EQUB, and the numerator to be stored in &72 and &73 using EQUW. EQUD may be used to store the contents of a full BASIC integer variable. (You may use EQUB instead of ?, and EQUD instead of !.)

Error trapping in assembler

The assembler will tell you of any mistakes which you make in typing in programs (syntax errors), and some errors associated with BASIC variables during assembly, but there is no such thing as a run-time error

in machine-code: you just have to fathom it out line by line. However, it is possible for you to trap errors generated while a machine-code program is running by using the BRK instruction. As an example, take the division program described in the previous section. Everyone knows that it is not possible to divide by zero, but the program does not know this. If you try to do so it unwittingly gives the answer 255.

It is simple to test the denominator before the division is started, and then to branch to an error routine. The whole program is not repeated here, but the following lines may be added:

```
53 LDA &71
56 BEQ Error

222 .Error BRK
224 EQUB 18
226 EQUS "Division by zero"
228 BRK
```

If you now run the program with a zero denominator, it will stop and print the message:

```
Division by zero at line 330
```

You can also type:

PRINT ERR RETURN

upon which it will give the correct error number, 18.

Any error message must take the following form:

```
BRK
EQUB errornumber (ERR)
EQUS "message"
BRK
```

Operating system calls from assembler

All the operating system calls available from BASIC, and many more, are available from a machine-code program. These routines are always accessed using a JSR to some address in the Operating System, and usually involve the passing of one or more parameters via the accumulator (for 1), X and Y (for 2 or 3), or a parameter block in memory

(for more than 3).

Here is a table showing all the Operating System calls available.

Routine		**Vector**		**Summary of function**
Name	Address	Name	Address	
		UPTV	222	User print routine
		EVNTV	220	Event interrupt
		FSCV	21E	File system control entry
OSFIND	FFCE	FINDV	21C	Open or close a file
OSBPUT	FFD4	BPUTV	218	Save a single byte to file from A
OSBGET	FFD7	BGERV	216	Load a single byte to A from file
OSARGS	FFDA	ARGSV	214	Load or save data about a file
OSFILE	FFDD	FILEV	212	Load or save a complete file
OSRDCH	FFE0	RDCHV	210	Read character (from keyboard) to A
OSASCI	FFE3	-	-	Write a character (to screen) from A plus LF if (A)=&0D
OSNEWL	FFE7	-	-	Write LF, CR (&0A, &0D) to screen
OSWRCH	FFEE	WRCHV	20E	Write character (to screen) from A
OSWORD	FFF1	WORDV	20C	Perform miscellaneous OS operation using control block to pass parameters
OSBYTE	FFF4	BYTEV	20A	Perform miscellaneous OS operation using registers to pass parameters
OSCLI	FFF7	CLIV	208	Interpret the command line given

When you use one of these routines, you must use a JSR to the corresponding address shown in the second column. For example, OSWRCH is called from assembler by typing:

```
JSR &FFEE
```

The routine stored at &FFEE uses the OSWRCH vector address, shown in the fourth column, as an indirect pointer to the actual location of the OSWRCH routine.

The reason for this is twofold:

(i) The actual address of the OSWRCH routine may be altered by the manufacturer without affecting the Operating System subroutine call in any way. JSR &FFEE will always give an OSWRCH call even though the address held in locations &20E and &20F may not be the same on every machine.

(ii) The user can alter the address held in the zero-page vector location and trap any call of that particular Operating System routine, indirecting such a call to the user's own routine anywhere in memory.

Use of Operating System calls

OSWRCH entry: &FFEE vector: &20E
This call writes the character whose ASCII code is in the accumulator to the screen.

Here is an example which will print the character L on the screen:

```
10 P% = &70
20 [OPT 3
30 .Start LDA #76          \ Load accumulator with ASCII code for L
40 JSR &FFEE               \ Jump to OSWRCH
50 RTS
60 ]
70 CALL Start
```

OSWRCH is also used with ASCII control codes (from 0 to 31). If you change line 30 to:

```
30 .Start LDA #7
```

then the program will output ASCII character 7, which is a ‘beep’.

Some BASIC instructions have ASCII values in the control code range, and these can therefore be used with OSWRCH. For example, PLOT has an ASCII value of 25, TAB an ASCII value of 31.

The following program uses TAB to print the character L half way across the screen:

```
10 P% = &70
20 [ OPT 2
30 .Start LDA #31
40 JSR &FFEE
50 LDA #19
60 JSR &FFEE
70 LDA #VPOS
80 JSR &FFEE
90 LDA #76
100 JSR &FFEE
110 RTS
120 ]
130 CALL Start
```

Each parameter is passed in turn to OSWRCH via the accumulator. The BASIC statement equivalent to the above program is:

```
PRINT TAB(19);"L";
```

(Note that this program will not work with OPT 3 because VPOS is affected.)

The BASIC instruction PLOT takes three parameters, PLOT A,X,Y. However, X may be 0 to 1279 and Y may be 0 to 1023, so each must be represented by two bytes. That means that an OSWRCH call with the accumulator set to 25 must be followed by five more OSWRCH calls to pass the parameters. The following program will plot a line on the screen:

```
10 MODE 4
20 P% = &70
30 [ OPT 3
40 .Start LDA #25
50 JSR &FFEE
```

```
60 LDA #5
70 JSR &FFEE
80 LDA #88
90 JSR &FFEE
100 LDA #2
110 JSR &FFEE
120 LDA #44
130 JSR &FFEE
140 LDA #1
150 JSR &FFEE
160 RTS
170 ]
180 CALL Start
```

This program is equivalent to

```
PLOT5,600,300
```

Lines 100 and 80 give X (2*256 + 88) and lines 140 and 120 give Y (1*256 + 44).

You'll see from the listing that the above routine, when assembled, occupies memory from &70 to &8E. Remember that user programs must use zero-page locations only between &70 and &8F, so this is almost the largest size routine that may be stored in the zero-page.

OSASCI entry: &FFE3
Writes the character whose code is in the accumulator to the screen using OSWRCH. However, if the accumulator contains &D then OSNEWL is called instead. The actual code at location &FFF3 is:

```
.OSASCI     CMP #&D
            BNE OSWRCH
.OSNEWL     LDA #&A
            JSR OSWRCH
            LDA #&D
.OSWRCH     JMP(WRCHV)
```

OSNEWL entry: &FFE7
This call issues a line feed/carriage return to the screen, as shown above.

After using OSWRCH, OSASCI or OSNEWL, the contents of the accumulator X and Y registers are unchanged. Flags C, N, V and Z are undefined, and D = 0.

OSRDCH entry: &FFE0 vector: &210
This call reads a character code from the keyboard into the accumulator.

After using OSRDCH, the contents of the X and Y registers are unchanged. Flags N, V and Z are undefined, and D=0. Flag C tells whether the read has been successful (C=0). If C=1 then an error has occurred and the error number is given in the accumulator. If C=1 and A=&1B then an escape condition has been detected and you must acknowledge this by performing an OSBYTE call with A=&7E or *FX126.

OSCLI entry: &FFF7 vector: &208
This call is used by the BASIC OSCLI instruction. From assembler it consists of a JSR to &FFF7, the command line string being placed in memory at a location given by the contents of the X register (address low byte) and Y register (address high byte). The command line string must be terminated by &D RETURN.

The following BASIC program illustrates this:

```
10 DIM address 20
20 keynumber=4
30 $address = "KEY" + STR$ keynumber + "LIST |M"
40 X%=address MOD 256
50 Y%=address DIV 256
60 CALL &FFF7
```

This will have the same effect as

```
*KEY 4 "LIST|M"
```

Note: The string indirection operator $ automatically puts a RETURN code (&D) after the string. EQUS however does not, and it must be inserted afterwards using EQUB &D or something like EQUS "FRED" + CHR$13.

OSFIND entry: &FFCE vector: &21C
Opens a file from cassette or disc for reading or writing. The contents of the accumulator determine the operation performed:

A = 0 close a file or files (CLOSE#).
A = &40 opens a file for input (OPENIN).
A = &80 opens a file for output (OPENOUT)
A = &C0 opens a file for input or output (OPENUP).

When OPENIN, OPENUP or OPENOUT is used, X and Y must contain the address of the filename. After the subroutine call, the accumulator will contain the channel number allocated to that file by the Operating System.

If CLOSE# is used then Y must contain the channel number of the file to be closed. If Y is 0 then all files will be closed.

OSBPUT entry: &FFD4 vector: &218
Writes the byte contained in the accumulator to the cassette or disc file (same as BPUT#). Y must contain the file channel number. After using OSBPUT, the contents of the accumulator, X and Y registers are unchanged.

OSBGET entry: &FFD7 vector: &216
Reads a byte from the cassette or disc file into the accumulator (same as BGET#). Y must contain the file channel number. After using OSBGET, the contents of the X and Y registers are unchanged. Flags N, V and Z are undefined, and D=0. Flag C tells whether the read has been successful (C=0). If C=1 then an error has occurred and the error number is given in the accumulator. If C=1 and A=&FE then the end of file has been reached.

OSFILE entry &FFDD vector: &212
Allows a whole file to be loaded or saved. The contents of the accumulator indicate the function to be performed. X and Y point to an 18 byte control block anywhere in memory, the structure of which is as follows:

OSFILE control block

00	Address of file name, which must be terminated	LSB
01	by &0D	MSB

02	Load address of file	LSB
03		
04		
05		MSB
06	Execution address of file	LSB
07		
08		
09		MSB
0A	Start address of data for write operations,	LSB
0B	or length of file for read operations	
0C		
0D		MSB
0E	End address of data, that is byte after	LSB
0F	last byte to be written or file attributes	
10		
11		MSB

The table below indicates the function performed by OSFILE for each value of A.

A=0 Save a section of memory as a named file. The file's catalogue information is also written.

A=1 Write the catalogue information for the named file.

A=2 Write the load address (only) for the named file.

A=3 Write the execution address (only) for the named file.

A=4 Write the attributes (only) for the named file.

A=5 Read the named file's catalogue information. Place the file type in A.

A=6 Delete the named file.

A=&FF Load the named file and read the named file's catalogue information.

Note: Values 1 to 6 are not available on a cassette filing system.

OSBYTE entry: &FFF4 vector: &20A

This is a family of Operating System calls which includes all the *FX calls available from BASIC. (These are not repeated here.) The call number is passed in the accumulator and parameters are passed in X or Y or both.

All OSBYTE calls are available from BASIC via a USR call, or by using a *FX call.

Here is a list of functions as given by each accumulator value (A):

A = 127 (EOF#) *FX127

Gives the end of file status of a previously opened file. X must contain the file channel number. Afterwards, X will be zero if the end of file has not been reached, non-zero if the end of file has been reached.

A = 129 (INKEY) *FX129

Either waits for a character from the keyboard buffer until a time limit expires (INKEY positive) or tests if a key is depressed (INKEY negative). All the discussion about auto-repeat and buffer flushing applies to this call.

For INKEY positive, Y must contain the most significant byte of the delay, and X the least significant (in hundredths of a second).

Afterwards, if Y=0 then a character has been detected and its code appears in X. Y=&1B indicates that ESCAPE was pressed and must be acknowledged with *FX126. Y=&FF indicates that no key was pressed in the allocated time.

For INKEY negative, Y must contain the requisite key-code in twos complement. Afterwards Y will be either TRUE (&FF) or FALSE (zero) depending on whether the key was pressed.

A = 131 (OSHWM) *FX131

Gives the address of the first free location in memory above that required for the Operating System. Usually equal to &E00. The address is given in X (low byte) and Y (high byte). For example, after *FX20,6.

A = 132 *FX132

Gives the lowest memory address used by the screen display in X (low byte) and Y (high byte).

A = 133 Low mode address *FX133

Gives the lowest address in memory used by a particular mode. Does not change mode but merely investigates the consequences of doing so. The

mode to be investigated must be in X. Afterwards, the address is contained in X (low byte) and Y (high byte).

A = 134 Read position of text cursor *FX134

Gives in X the X coordinate of the text cursor, and in Y the Y co-ordinate (same as POS and VPOS).

A = 135 Read character at position of text cursor *FX135

Gives in X the ASCII code of the character at the current text cursor position, and in Y the current mode number. X is 0 if the character is not recognisable.

Here is a BASIC function which can be used to read the character at any position X,Y on the screen:

```
1000 DEF FNreadcharacter(column%,row%)
1100 LOCAL A%,currentX%,currentY%,character%
1200 currentX% = POS:currentY% = VPOS
1300 VDU31,column%,row%
1400 A% = 135
1500 character% = (USR(&FFF4) AND &FF00) DIV &100
1700 VDU31,currentX%,currentY%
1800 = CHR$ character%
```

To give the character at position X,Y this function would be called by passing X and Y as the two parameters:

```
PRINT FNreadcharacter(X,Y)
```

A = 137 Motor control *FX137

Similar to *MOTOR. X=0 will turn off the cassette motor, X=1 will turn it on.

A = 139 (*OPT) *FX138

Exactly the same as *OPT. The parameters are passed in X and Y.

A = 145 Get character from keyboard buffer *FX145

Reads a character code from a buffer into the Y register. X=buffer number (0 to 9 inclusive). C=0 indicates a successful read, C=1 indicates that the buffer is empty.

A=218 Cancel VDU queue *FX218

Many VDU codes expect a sequence of bytes (as shown earlier with PLOT and TAB). This call signals the VDU software to throw away the bytes it has received so far. Before use, X and Y must contain zero.

OSWORD entry: &FFF1 vector: &20C
This is a family of operating system calls which uses a parameter block somewhere in memory to supply data to the routine and to receive results from it. The exact location of the parameter block must be specified in X (low byte) and Y (high byte). The accumulator contents determine the action of the OSWORD call.

A = 0 Read a line from keyboard to memory.

Accepts characters from the keyboard and places them at a specified location in memory. During input the DELETE key (ASCII 127) deletes the last character entered, and CTRL U (ASCII 21) deletes the entire line. The routine ends if RETURN is entered (ASCII 13) or the ESCAPE key is pressed.

The control block contains five bytes:

YX	(low byte) Address at which
YX+1	(high byte) line is to be stored
YX+2	Maximum length of line
YX+3	Minimum acceptable ASCII value
YX+4	Maximum acceptable ASCII value

Characters will only be entered if they are in the range specified by YX+3 and YX+4.

Afterwards, C=0 indicates that the line was terminated by a RETURN. C not equal to zero indicates that the line was terminated by an ESCAPE. Y is set to the length of the line, excluding the carriage return if C=0.

A = 1 Read clock

Reads the internal elapsed-time clock into the five bytes pointed to by X and Y. The clock is incremented every hundredth of a second, and is used by the BASIC variable TIME.

A = 2 Write clock

Sets the internal elapsed-time clock to the value given in the five bytes pointed to by X and Y. Location YX is the least significant byte of the clock, YX+4 is the most significant.

A = 3 Read interval timer

In addition to the clock there is an interval timer which is also incremented every hundredth of a second. The interval is stored in five bytes pointed to by X and Y. See OSWORD with A = 1.

A = 4 Write interval timer

X and Y point to five bytes which contain the new value to which the clock is to be set. The interval timer may cause an event when it reaches zero. Thus setting the timer to &FFFFFFFFD would cause an event after three hundredths of a second.

A = 7 SOUND

Equivalent to the BASIC SOUND statement. The eight bytes pointed to by X and Y contain the four two-byte parameters (in fact only the least significant byte of each need be used).

YX	Q (channel, 0 to 3)
YX+1	zero
YX+2	A (envelope, -15 to 4)
YX+3	zero, or &FF if -1 or some other negative value
YX+4	P (pitch, 0 to 255)
YX+5	zero
YX+6	D (duration, 1 to 255)
YX+7	zero

A=8 ENVELOPE

Equivalent to the BASIC ENVELOPE statement, X and Y point to 14 bytes of data which are the 14 parameters used by ENVELOPE.

A=9 POINT

Equivalent to BASIC POINT function. The parameter block pointed to by X and Y must be set up as follows:

YX	X (low byte) coordinate
YX+1	X (high byte) coordinate
YX+2	Y (low byte) coordinate
YX+3	Y (high byte) coordinate

Afterwards, YX+4 will contain the logical colour value of that particular graphics coordinate. If the coordinate is off the screen then YX4 contains &FF.

A = 10 Read character definition

Characters are displayed on the screen as an 8 × 8 matrix of dots. The pattern of dots for each character, including user-defined characters, is stored as eight bytes. This call enables the eight bytes to be read into a block of memory starting at the address given in X and Y, plus 1. The ASCII code of the character must be the first entry on the parameter block when the routine is called.

Afterwards, the parameter block contains data as shown below:

YX	Character code
YX+1	Top tow of displayed character
YX+2	Second row of displayed character
.	
.	
.	
YX+8	Bottom row of displayed character

Here is a program to illustrate this OSWORD call, and the method of calling OSWORDs in general. It takes each of the characters in turn, reads the matrix definition, and then reverses this definition by shifting the bits in each byte, and then redefining each character using VDU 23. The result makes your program interesting to read, to say the least!

```
10MODE 6
20 DIM Q% 100
30 FOR I=0 TO 3 STEP 3
40 P%=Q%
50 [OPT I
60 .Character LDA &601      \ Take low address of parameter
70 STA &70                  \ and store it in zero-page.
```

```
80 LDA &602              \ Take high address of parameter
90 STA &71               \ and store it in zero-page.
100 LDY #0               \ Clear Y register.
110 LDA (&70),Y          \ Get parameter (ASCII code)
120 STA &70              \ and store it in zero-page.
130 LDX #&70             \ Set X to OSWORD parameter block address low
                         \ byte.
140 LDA #10              \ Set OSWORD function.
150 JSR &FFF1            \ Jump to OSWORD.
160 LDX #0               \ Clear X register.
170 .Reverse LDY #8      \ Set Y to 8 as shift counter.
180 .Loop ASL &71,X      \ Shift each byte into the byte
190 ROR &70,X            \ below, thereby reversing it.
200 DEY                  \ Decrement Y.
210 BNE Loop             \ Repeat if not zero.
220 INX                  \ Increment X.
230 CPX #8               \ Compare with 8.
240 BNE Reverse          \ Repeat if not equal.
250 RTS
260 ]
270 NEXT
280 *FX20,6
290 FOR I% = 33 TO 126
300 PRINT CHR$ I%
310 CALL Character,I%
320 VDU23,I%,?&77,?&76,?&75,?&74,?&73,?&72,?&71,?&70
330 NEXT
```

This program illustrates a number of the features demonstrated in this part of the book. It calls the machine-code routine Character with a parameter, and lines 60 to 120 transfer the parameter to location &70, which is a safe place at which to store any OSWORD data.

Line 130 sets X to the low byte of the address of the OSWORD parameter block (it is not necessary to set Y because Y is already zero).

Lines 140 and 150 carry out the OSWORD call.

Lines 160 to 240 reverse each of the bytes of the character definition.

Line 280 explodes the character memory allocation to its maximum allowing all the characters to e redefined, and line 320 carries out this redefinition.

It should be noted that if this program were more than &300 bytes long, it would get overwritten by the soft characters.

A=11 Read colour assigned to logical value

Gives the actual colour value assigned to the logical colour value contained in the location pointed to by X and Y. Afterwards, location YX will contain the logical value, and location YX+1 contain the actual value. In fact YX+1 to YX+4 contain the four-byte physical colour - you must reserve space for five bytes.

Events

Events are conditions which occur within the computer and which can be trapped by the user so as to provide useful information. For example, it is possible to detect when ESCAPE has been pressed.

To be able to act upon an event, that event must first be enabled by *FX14:

*FX14,0 enables output buffer empty event.
*FX14,1 enables input buffer full event.
*FX14,2 enables character entering keyboard buffer event.
*FX14,4 enables start of vertical synchronisation of screen display event.
*FX14,5 enables the interval timer crossing zero event.
*FX14,6 enables ESCAPE pressed event.

The Operating System detects all the above events when they occur, but ignores them if they have not been enabled with the appropriate *FX14 call. If an event occurs which has been enabled then program execution indirects via &220 and places an event code (shown below) in the accumulator. The contents of X and Y may also depend upon the event.

The event codes are as follows:

A=0	Output buffer empty. (X contains buffer identity.)
A=1	Input buffer full. (X contains buffer identity. Y contains the ASCII code of character that could not be stored in buffer.)
A=2	Key pressed.
A=4	Vertical synchronisation of screen display.
A=5	Interval timer crossing zero.
A=6	ESCAPE detected.

Any address may be stored in the two bytes &220 and &221 to which the program will transfer execution on detection of an enabled event. You may write your own code at this address in order to process the event, but it must be terminated by RTS, and should not take too long (one millisecond maximum).

Each of the events may be disabled by a corresponding *FX13. For example. *FX13,1 will disable the input buffer full event.

Assembly Language mnemonics

This section describes, in alphabetical order, all the 6502 assembler mnemonics.

The following abbreviations are used:

A	accumulator
X	index register X
Y	index register Y
F	flags register
PC	program counter
PCH	program counter (high byte)
PCL	program counter (low byte)
SP	stack pointer
M	memory address
←	'becomes' (assignment)
→	'affects' (flags)
( )	contents of
&	hexadecimal
(A4) (M7) (X0)	etc. specified bit position in register or memory
N V B D I Z C	status flags

Because this section has been written for use with the Electron's assembler, the addressing modes quoted are simplified as compared with those that are specified for use with general machine-code programming of the 6502.

ADC

Add with carry

Action	A ← (A) + Data + C
Description	Add the contents of memory location or immediate data to the accumulator, plus the carry bit. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NVZC
Comments	To add without the carry, bit C must be cleared beforehand by using CLC.

AND

Logical AND

Action	A ← (A) AND Data
Description	AND the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NZ
Comments	See BASIC AND for truth table.

ASL

Arithmetic shift left

Action	C ← A or M ← 0

Description	Shift the contents of the accumulator or memory location left one bit position. Bit 7 falls into the carry (bit C), zero is entered from the right. Result remains in either the accumulator or the memory location.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	ASL A acts on the accumulator.

BCC

Branch if carry clear

Description	Go to specified label or address if C=0.
Action	If bit C is zero, execution continues at the specified label or address. If bit C is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BCS

Branch if carry set

Action	Go to specified label or address if C=1
Description	If bit C is 1, execution continues at the specified label or address. If bit C is zero then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BEQ

Branch if equal to zero

Action	Go to specified label or address if Z = 1.
Description	If bit Z is 1, execution continues at the specified label or address. If bit Z is zero then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BIT

Compare memory bits with accumulator

Action	(A) (M) $\rightarrow$ F
Description	The accumulator is compared with the contents of a memory location. If the two are the same then bit Z is set to 1; if not then bit Z is cleared. Bits 6 and 7 of the data from memory are loaded into bits X and N respectively. The contents of the accumulator remain unchanged.
Addressing modes	Absolute Zero-page
Flags affected	NVZ

BMI

Branch if minus

Action	Go to specified label or address if N=1
Description	If bit N is 1, execution continues at the specified label or address. If bit N is zero then execution continues at the next instruction.
Addressing modes	Relative

Flags affected	None
Comments	Specified label or address must be in range.

BNE

Branch if not equal to zero

Action	Go to specified label or address if Z=0
Description	If bit Z is zero, execution continues at the specified label or address. If bit Z is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be range.

BPL

Branch if plus

Action	Go to specified label or address if N=0
Description	If bit N is zero, execution continues at the specified label or address. If bit N is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BRK

Break

Action	STACK ← (PC) + 2 STACK ← (F) PCL ← (&FFFE) PCH ← (&FFFF)

Description	This is a software interrupt. The contents of the program counter plus 2 are pushed on the stack, followed by the contents of the flags register. The program counter is then loaded with the contents of locations &FFFE (low byte) and &FFFF (high byte). Bit B is set to 1.
Addressing modes	Implied
Flags affected	B
Comments	Used mainly for error trapping and debugging.

BVC

Branch if overflow clear

Action	Go to specified label or address if V=0
Description	If bit V is zero, execution continues at the specified label or address. If bit V is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BVS

Branch if overflow set

Action	Go to specified label or address if V=1
Description	If bit V is 1, execution continues at the specified label or address. If bit V is zero then execution continues at the nexT instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

CLC

Clear carry

Action	C ← 0
Description	Bit C is cleared.
Addressing modes	Implied.
Flags affected	C
Comments	Often required before ADC.

CLD

Clear decimal flag

Action	D ← 0
Description	Bit D is cleared, which means that the processor is in binary mode.
Addressing modes	Implied
Flags affected	D
Comments	Should be used at the beginning of all routines which do not use binary coded decimal.

CLI

Clear interrupt mask

Action	I ← 0
Description	Bit I is cleared, which enables interrupts.
Addressing modes	Implied
Flags affected	I
Comments	An interrupt is triggered when an external device, such as a printer, requires attention.

CLV

Clear overflow flag

Action	V ← 0
Description	Bit V is cleared.
Addressing modes	Implied.
Flags affected	V

CMP

Compare with accumulator

Action	(A) – Data → F
Description	Contents of memory location or immediate data are subtracted from the accumulator. If the result is zero then bit Z is set; if not zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the accumulator contents are greater than or equal to the data. The contents of the accumulator remain unchanged; only the flags register is affected.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NZC

CPX

Compare with X register

Action	(X)-Data → F
Description	Contents of memory location or immediate data are subtracted from the X register. If the result is zero then bit Z is set; if zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the X register contents are greater than or

equal to the data. The contents of the X register remain unchanged; only the flags register is affected.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/Indexed

Flags affected NZC

CPY

Compare with Y register

Action (Y) – Data ← F

Description Contents of memory location or immediate data are subtracted from the Y register. If the result is zero then bit Z is set; if not zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the Y register contents are greater than or equal to the data. The contents of the Y register remain unchanged; only the flags register is affected.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/Indexed

Flags affected NZC

DEC

Decrement memory

Action (M) ← (M)-1

Description The contents of the specified memory are decremented by 1.

Addressing modes	Zero-page Absolute Indexed (X only)
Flags affected	NZ

DEX

Decrement X register

Action	X ← (X)-1
Description	The contents of the X register are decremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables X to be used as a counter.

DEY

Decrement Y register

Action	Y ← (Y) - 1
Description	The contents of the Y register are decremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables Y to be used as a counter

EOR

Logical exclusive-OR

Action	(A) ← (A) EOR Data
Description	Exclusive-OR the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.

Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NZ
Comments	See BASIC EOR for truth table.

INC

Increment memory

Action	M ← (M) + 1
Description	The contents of the specified memory location are incremented by 1.
Addressing modes	Zero-page Absolute Indexed (X only)
Flags affected	NZ

INX

Increment X register

Action	X ← (X)+1
Description	The contents of the X register are incremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables X to be used as a counter.

INY

Increment Y register

Action	Y ← (Y)+1

Description	The contents of the Y register are incremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables Y to be used as a counter.

JMP

Unconditional jump

Action	PC ← address
Description	Execution continues at the specified label or address.
Addressing modes	Absolute Indirect
Flags affected	None
Comments	There is no restriction on length of jump; label or address may be anywhere in memory. This is the only instruction which may use straight indirect addressing.

JSR

Jump to subroutine

Action	STACK ← (PC)+2 PC ← address
Description	The contents of the program counter plus 2 are pushed on the stack (this is the address of the instruction following JSR). Execution continues at the specified label or address.
Addressing modes	Absolute
Flags affected	None
Comments	The subroutine to which control is transferred must be terminated by an RTS instruction. JSR is used

whenever you wish to make an Operating System call from assembler.

LDA

Load accumulator

Action	A ← Data
Description	Load the accumulator with contents of memory location or immediate data.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NZ

LDX

Load X register

Action	X ← Data
Description	Load the X register with contents of memory location or immediate data.
Addressing modes	Immediate Zero-page Absolute Indexed (Y only)
Flags affected	NZ

LDY

Load Y register

Action	Y ← Data
Description	Load the Y register with the contents of memory location or immediate data.

Addressing modes	Immediate Zero-page Absolute Indexed (X only)
Flags affected	NZ

LSR — Logical shift right

Action	0 → A or M → C
Description	Shift the contents of the accumulator or memory location right one bit position. Bit 0 falls into the carry (bit C), zero is entered from the left. Result remains in either the accumulator or memory location.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	LSR A acts on the accumulator.

NOP — No operation

Action	None
Description	Does nothing for two clock cycles.
Comments	Used for timing a program, or to fill in gaps caused by deleted instructions.

ORA — Logical OR

Action	A ← (A) OR Data

Description	OR the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NZ
Comments	See BASIC OR for truth table.

PHA

Push accumulator on to stack

Action	STACK ← (A) SP ← (SP) - 1
Description	The contents of the accumulator are pushed on to the stack. The stack pointer is decremented. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	None

PHP

Push flags register on to stack

Action	STACK ← (F) SP ← (SP)-1
Description	The contents of the flags register are pushed on to the stack. The stack pointer is decremented. The flags register contents remain unchanged.
Addressing modes	Implied
Flags affected	None

PLA

Pull data from stack into accumulator

Action	A ← (STACK) SP ← (SP)+1
Description	Pull the top byte of the stack into the accumulator. Increment the stack pointer.
Addressing modes	Implied
Flags affected	NZ

PLP

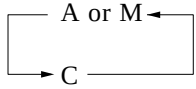
Pull data from stack into flags register

Action	F ← (STACK) SP ← (SP) + 1
Description	Pull the top byte of the stack into the flags register. Increment the stack pointer.
Addressing modes	Implied.
Flags affected	NVBDIZC

ROL

Rotate left

Action

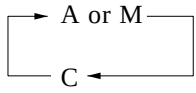


Description	Rotate the contents of the accumulator or memory location left one position. The carry (bit C) is entered from the right, bit 7 falls into the carry.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	ROL A acts on the accumulator. This is a 9-bit rotation.

ROR

Rotate right

Action



Description — Rotate the contents of the accumulator or memory location right one bit position. The carry (bit C) is entered from the left, bit 0 falls into the carry.

Addressing modes — Zero-page
Absolute
Indexed (X only)
Accumulator

Flags affected — NZC

Comments — ROR A acts on the accumulator. This is a 9-bit rotation.

RTI

Return from interrupt

Action

F ← (STACK)
SP ← (SP) + 1
PCL ← (STACK)
SP ← (SP) + 1
PCH ← (STACK)
SP ← (SP) + 1

Description — Restore the contents of the accumulator or memory location right one bit position. The carry (bit C) is entered from the left, bit 0 falls into the carry.

Addressing modes — Implied

Flags affected — NVBDIZC

Comments — Used to return to the execution of a program, after an interrupt has been dealt with.

RTS

Return from subroutine

Action	PCL ← (STACK) SP ← (SP)+1 PCH ← (STACK) SP ← (SP)+1 PC ← (PC)+1
Description	Restore the contents of the program counter, which were previously stored on the stack, and increment the program counter by 1. Increment the stack pointer.
Addressing modes	Implied
Flags affected	None
Comments	Continues execution from position after sub-routine call. Used by the Electron's assembler to return to BASIC.

SBC

Subtract with carry

Action	A ← (A) ← Data ← C (C is NOT C, which is the borrow.)
Description	Subtract the contents of memory location or immediate data from the accumulator, with borrow. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/Indexed
Flags affected	NVZC
Comment	To subtract without the borrow, bit C must be set beforehand by using SEC.

SEC

Set carry

Action	C ← 1
Description	Bit C is set.
Addressing modes	Implied.
Flags affected	C
Comments	Often required before SBC.

SED

Set decimal flag

Action	D ← 1
Description	Bit D is set, which means that the processor is in decimal mode (BCD).
Addressing modes	Implied
Flags affected	D

SEI

Set interrupt mask

Action	I ← 1
Description	Bit I is set, which disables interrupts.
Addressing modes	Implied
Flags affected	I

STA

Store accumulator in memory

Action	M ← (A)
Description	Store contents of the accumulator at the specified memory location. The accumulator contents remain unchanged.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/Indexed

Flags affected None

STX

Store X register in memory

Action M ← (A)

Description Store contents of X register at the specified memory location. The X register contents remain unchanged.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/Indexed

Flags affected None

STY

Store Y register in memory

Action M ← (Y)

Description Store contents of Y register at the specified memory location. The Y register contents remain unchanged.

Addressing modes
Zero-page
Absolute
Indexed (zero-page X only)

Flags affected None

TAX

Transfer accumulator to X

Action	X ← (A)
Description	Copy the contents of accumulator into X register. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TAY

Transfer accumulator to Y

Action	Y ← (A)
Description	Copy the contents of accumulator into Y register. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TSX

Transfer stack pointer to X

Action	X ← (SP)
Description	Copy the contents of the stack pointer into X register. The stack pointer contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TXA

Transfer X register to accumulator

Action	A ← (X)

Description	Copy the contents of the X register into the accumulator. The X register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TXS

Transfer X register to stack pointer

Action	S ← (X)
Description	Copy the contents of the X register into the stack pointer. The X register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TYA

Transfer Y register to accumulator

Action	A ← (Y)
Description	Copy the contents of the Y register into the accumulator. The Y register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ