1 The 6502 Microprocessor

At the heart of any microcomputer is the microprocessor. In the BBC micro and Electron this is the 6502, which provides the computer with all its processing power.

By itself, the 6502 is a very simple machine; but it can be made to perform relatively complex tasks (like interpreting programs written in BASIC) by stringing together many of its simple instructions into a machine code program. This section is not really a tutorial on machine code programming, but more an introduction to the 6502 to give an idea of how the rest of the BASIC system operates around it.

1.1 The 6502 registers

The 6502 has 6 registers altogether: the accumulator A, the index registers X and Y, the program counter PC, the stack register S, and the processor status register P. These are shown in the *programming model*, fig 1.1.

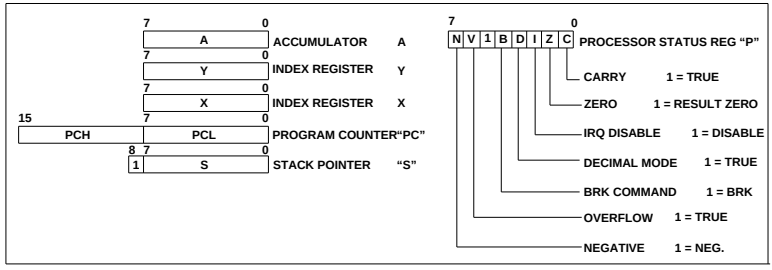

Figure 1.1 − The 6502 programming model.

The accumulator A

The accumulator A is used for all of the arithmetic and logical operations done by the 6502, as well as just loading it from memory and storing it back into memory again. It is the only 6502 register which can be used for adding, subtracting, ANDing, etc. of numbers, and so tends to be used rather a lot. It is 8 bits (1 byte) wide, so it can only hold 256 (&100) different numbers altogether.

As an example, the instruction:

```
AND &80
```

ANDs the 8-bit number in the accumulator with the 8-bit number in location &80 (i.e. ?&80), leaving the result in the accumulator.

The index registers X and Y

Either of these can be used a counter, or as an offset into a table in memory. They can also be loaded from and stored into memory. Again they are only 8 bits wide, so they can only count up to 255 (&FF).

As an example, the instruction:

```
LDA &2000,Y
```

loads the accumulator from the location at &2000+ Y. Thus if the Y register contained &2A, the accumulator would be loaded with the contents of location &202A.

The program counter PC

This is the register which tells the 6502 where to get its next instruction from. In a machine code program, the instructions are stored one after another in memory, and the program counter steps through these while they are executed. In pratice, you don’t really notice the program counter much (just as you don’t notice the text pointers that BASIC uses to step through its program). The program counter is the only 16-bit register that the 6502 has, and allows it to address 65536 (&10000) locations.

As an example, the instruction:

```
JMP &8000
```

jumps to location &8000 (in a similar way to the GOTO statement) by loading the number &8000 into the program counter.

The stack pointer S This register points into a stack in page 1, from &100 to &1FF. Numbers can be *pushed* on the top of the stack, to save them until later, and then *pulled* (or *popped*) again to get back the last number that was pushed. This is called a *last in first out* (LIFO) structure, because the first thing that you get out was the last thing that you put in.

When a single byte number is pushed on the stack, it is placed in memory at the location pointed to by the stack pointer (&1F0, say, if the S register contains &F0), and the stack pointer is decremented to point to the location below it in memory. When a byte is pulled, the opposite takes place: the stack pointer is incremented, and the number loaded from the location in page 1 which it points to.

As an example, the instruction:

```
PHA
```

pushes the contents of the accumulator on the 6502 stack.

The processor status register P

This register contains the flags that the 6502 needs for its arithmetic and system operations.

N This is the negative flag. It is set whenever the top bit is set in the 8-bit number just calculated or loaded from memory (see section 1.2 for negative number representation).

V This is set if an overflow occurred the last time an 8-bit signed add or subtract operation was performed (see section 1.2).

B This is the BRK flag. It is set when a BRK instruction is executed (see section 1.3).

D This is the decimal flag. It can be set if any *binary coded decimal* arithmetic is to be performed (see section 1.2).

I This is the interrupt flag. It can be set to prevent the 6502 from being interrupted by a hardware IRQ.

Z This is the zero flag. It is set whenever the 8-bit number just calculated or loaded from memory is zero.

C This is the carry flag. The ADC and SBC instructions use this to indicate whether there was a 'carry over' from the calculation just performed (see section 1.2). It is also used by the shift instructions (section 1.3).

Some of these flags can be tested so that parts of the machine code program are executed conditionally. For example the instruction:

```
BCS carry
```

will branch to the location 'carry' if the carry flag is set: otherwise the program will continue with the instruction after the 'BCS'. The use of these flags is explained more with the instructions in section 1.3.

1.2 Machine code arithmetic

As the 6502 accumulator is only 8 bits wide, it can only represent one of 256 different numbers. Hexadecimal notation is convenient to represent numbers in a byte, because each hexadecimal digit represents 4 bits, so 2 digits represent a whole byte, from &00 to &FF. What the 256 different numbers are used to represent is fairly arbitrary: they can represent positive numbers, negative numbers, or part of a larger number.

1.2.1 Negative numbers

A single byte can be used to represent the positive integers from 0 to 255. This is convenient for counting; but for arithmetic, some way of representing negative numbers is really needed.

If you add the single byte number &04 to &FC, you get &00 (ignoring any carry out of the byte). So, in this case, &FC seems to be behaving as if it was −4 (as '−4' is 'the number which you add to 4 to get 0'). However, it can also represent the positive number 252. The answer is that with only 8 bits, you can't tell the difference between '252' or '252 − 256' or '252 + 256' or '252 + any number of 256s'.

So if you want half of the 256 numbers you can represent in a byte to be negative, all you have to do is leave &00 to &7F to be the positive numbers 0 to 127, and let &80 to &FF represent the negative ones. These negative ones will have the same representation as the positive numbers which you get by adding 256 to them, so '−4' will be the same as '−4+256' (252), i.e. &FC.

Choosing the numbers above &80 to be negative is very convenient, because it means that all the numbers with the top bit of the byte set will be negative, while all the numbers with the top bit zero will be positive. Thus the top bit of a signed number like this is the *sign bit* of the number. This is what the N flag in the 6502 is for: it indicates the *sign bit* of the number which has just been operated on.

This representation is often called *2's complement* representation. This is because the negative of a number can be found by changing all the '1 's in the binary representation to '0', and all the '0's to 'I 's (one's complement), and then adding 1 to it. For example, 4 is '00000100', so inverting all the bits we get '11111011', and adding 1 we get '11111100', or &FC. What you're really doing when you invert all the bits of a single byte number, is subtracting it from 255 (i.e. '11111111 '), so by adding the extra 1 again, you get the number subtracted from 256.

1.2.2 Larger numbers

At first, it may seem a bit restrictive only to be able to represent 256 different numbers in a single byte. However, in decimal, a single digit can only represent one of 10 different numbers (0 to 9), but larger numbers are written down with more than 1 digit, like ‘59’. In exactly the same way, large numbers can be stored in memory in several bytes, so 1000 (&03E8) can be stored as &03 in one byte (the *most significant byte*, or MSB) and &E8 in the other (the *least significant byte*, or LSB).

When addition is performed in decimal, the least significant digits are added first. Then the next digits are added, together with any *carry* from the first ones, if there was any. The same can be done to add a pair of large numbers in memory: for example, to add 1000 (&03E8) to 25 (&0019) the following operations will take place:

1 Add the LSB of the first number (&E8) to the LSB of the second number (&19). This gives the result &01 with a 1 to carr-y over to the next byte.

2 Add the MSB of the first number (&03) to the MSB of the second number (&00), with an extra 1 carried over from the last addition. This gives the result &04, with no carry.

The final result of the addition is then &0401, or 1025 in decimal.

The carry over from one byte to the next is done by the C (carry) flag in the 6502 status register. If this is set, the 6502 ADC (add with carry) instruction will automatically add an extra 1 to the addition it is about to do. To add the LSBs together, the carry flag must be cleared first (with the CLC instruction), or an extra 1 may be added where you didn’t want one.

Subtraction of larger numbers is done in a very similar way, except the C flag is used as a ‘borrow’: if it is cleared, the last subtraction needed to borrow 1 from the next byte up, so 1 extra will be subtracted when the next subtraction is performed. To subtract the LSBs, the carry flag must be set first (with the SEC instruction), so the extra 1 is not subtracted.

1.2.3 Overflow

If the single-byte 2's complement number &50, representing 80, is added to the number &33, representing 51, we get &83, which represents −125. Clearly this is not right: the number we should have got was 131. However, 131 is too big to be represented by our single-byte 2's complement number: only the numbers −128 to +127 are allowed. When this happens the result has *overflowed*.

The V (overflow) flag in the 6502 is set if the last add or subtract instruction caused an *overflow*, and the result which was obtained is not a correct 2's complement respresentation of the answer.

After an addition, the overflow flag will be set if:

(a) a carry occured from bit 6 to bit 7 of the byte, without a carry out of the byte; or

(b) a carry occurred out of the byte without a carry from bit 6 to bit 7.

In other words:

(a) the numbers being added were both positive, but the result is negative; or

(b) the numbers being added were both negative, but the result is positive.

For subtraction, the overflow flag will be set in the corresponding situations, as though you were adding the negative of the number being subtracted.

1.2.4 Binary coded decimal

If the D flag of the 6502 is set it will operate in its binary coded decimal mode, where the 8-bit byte is used to represent two decimal digits, one in each nibble (4 bits). Thus the decimal number 26 will be represented by the hexadecimal number &26. When operating in this mode, all add and subtract operations will automatically adjust the result to ensure that it is in binary coded decimal form again.

This mode is not used very often, although sometimes it is useful for representing decimal numbers exactly.

The decimal flag must never be set when using any operating system or BASIC routines, as they expect to operate in standard binary mode.

1.3 The Instruction Set

The 6502 has 56 different instructions. This section lists them in groups of similar actions, giving the operation of the instruction, and the flags affected by it. Section 1.4 gives the addressing modes which can be used with these instructions. Apppendix C gives a list of these instructions in alphabetical order.

Load/store operations

LDA The accumulator is loaded with the contents of the specified memory location. Flags affected: N,Z.

LDX The X register is loaded with the contents of the specified memory location. Flags affected: N,Z.

LDY The Y register is loaded with the contents of the specified memory location. Flags affected: N,Z.

STA The contents of the accumulator are stored in memory. The flag bits are unaffected.

STX The contents of the X register are stored in memory. The flag bits are unaffected.

STY The contents of the Y register are stored in memory. The flag bits are unaffected.

Register transfer operations

TAX Copy the contents of the accumulator to the X register. The contents of A are unaffected. Flag bits affected: N ,Z.

TAY Copy the contents of the accumulator to the Y register. The contents of A are unaffected. Flag bits affected: N ,Z.

TSX Copy the contents of the stack pointer to the X register. The contents of S are unaffected. Flags bits affected: N ,Z.

TXA Copy the contents of the X register to the accumulator. The contents of X are unaffected. Flags affected: N ,Z.

TXS Copy the contents of the X register to the stack pointer. The contents of X and the status register are unaffected.

TYA Copy the contents of the Y register to the accumulator. The contents of Y are unaffected. Flag bits affected: N,Z.

Stack operations

PHA The contents of the accumulator are pushed on the stack. The stack pointer is updated to point to the next available location. Flag bits are unaffected.

PHP The contents of the processor status register are pushed on the stack, and the stack pointer is updated. Flag bits are unaffected.

PLA The byte on top of the stack is transferred to the accumulator and the stack pointer is updated. Flag bits affected: N,Z.

PLP The byte on top of the stack is transferred to the P register and the stack pointer is updated. All flag bits are affected.

Arithmetic and logical operations

ADC Add the contents of the specified memory location with the carry flag to the accumulator. Result is left in the accumulator. Flags affected: N, V ,Z,C.

SBC The specified data is subtracted from the accumulator with a borrow if the carry flag is clear. The result is left in A. C is cleared if a borrow was required else it is set. Flags affected: N, V ,Z,C

CMP The contents of the specified memory location are subtracted from the accumulator, setting the flags, but not storing the result. A is unaffected. Flags affected: N is set to bit7 of the result, Z is set if the result is zero. C is set if the unsigned number in the accumulator is greater than or

equal to the data, otherwise cleared (as for the SBC instruction).

CPX The contents of the specified memory lcation are subtracted from the X register but the result is not stored. The flags are set in the same way as for CMP.

CPY The contents of the specified memory location are subtracted from the Y register but the result is not stored. The flags are set in the same way as for CMP.

AND Performs the bit by bit logical AND of the accumulator and the specified memory location. Result is left in the Accumulator. Flags affected: N,Z.

ORA The bit by bit logical ORing takes place between the accumulator and the memory location, the result is left in A. Flags affected: N,Z.

EOR The contents of the accumulator are exclusive-ored on a bit by bit basis with the specified data, the result is left in A. Flags affected: N ,Z.

BIT The logical AND of the accumulator and memory is performed but is not stored. Flag bits affected: Z is set if the result was zero, V and N are set to bits 6 and 7 of the memory location respectively.

Increment/decrement operations

DEC The number in the specified memory location is decremented by 1. Flags affected: N,Z

DEX The number in the X register is decremented by 1. Flags affected: N,Z.

DEY The number in the Y register is decremented by 1. Flags affected: N,Z.

INC The number in the specified memory location is incremented by 1. Flags affected: N ,Z.

INX The number in the X register is incremented by 1. Flags affected: N,Z.

INY The number in the Y register is incrememted by 1. Flags affected: N,Z.

Shift and rotate operations

ASL The contents of the accumulator or the memory location are shifted one bit to the left. Bit 7 falls in to the carry flag, and bit 0 is set to 0. Flags affected: N,Z,C.

LSR The contents of the accumulator or the memory location are shifted to the right by 1 bit. 0 is placed in bit 7 , and bit 0 transfered to C. Flags affected: N is cleared, Z, C.

ROL The contents of the accumulator or the memory location are rotated by one bit to the left. The carry flag is shifted into bit 0, and bit7 is shifted in to the carry flag. Flags affected: N,Z,C.

ROR The contents of the accumulator or the memory location are rotated by one bit to the right. The carry flag is shifted into bit 7, and bit(] is shifted in to the carry flag. Flags affected: N,Z,C.

Program control operations

JMP The program counter is loaded with a new address and the JSR program continues from that point. Flags are unaffected. The contents of the program counter +2 are pushed on the stack and a new program counter is loaded from the argument. This is called a subroutine call. Flags are unaffected.

RTS The program counter is pulled off the stack and incremented by one, to return from the subroutine. The stack pointer is updated. Flags bits are unaffected.

Conditional branch operations

BCC If the C flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BCS If the C flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BEQ If the Z flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BNE If the Z flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BMI If the N flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BPL If the N flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BVC If the V flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BVS If the V flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

Flag operations

CLC The Carry flag is cleared, no other flags are affected. CLD The Decimal flag is cleared, no other flags are affected. This puts the 6502 in binary mode.

CLI The Interrupt flag is cleared, no other flags are affected. This enables interrupts from the IRQ input.

CLV The Overflow bit is cleared, no other flags are affected.

SEC C is set. Other flags remain unaffected.

SED D is set. The ADC and SBC instructions will now operate in the BCD mode. Other flags remain unaffected.

SEI I is set. No IRQs will be acknowledged until it is cleared. Other flag bits are unaffected.

System control operations

BRK This causes an interrupt to be generated and is not maskable. Flags affected: B is set.

NOP The processor does nothing for two cycles.

RTI This pulls the processor status and then the program counter off the stack. The stack pointer is updated. This is used to terminate an interrupt. All flags affected.

1.4 Addressing modes

The *addressing mode* is used to specify how the data needed by an instruction is to be accessed from memory. Most instructions have a single-byte *opcode*, which tells the 6502 which instruction and addressing mode it is, followed by one or two bytes of data to be used by the instruction. Chapter 6 has a table of all the possible opcodes.

Altogether, the 6502 has 13 different addressing modes: these are listed in this section.

Implied addressing

No extra data is required by the instruction. For example:

```
TAX
```

will transfer the contents of the accumulator to the X register, and doesn't need any other information.

Accumulator addressing

No extra data is required by the instruction: it operates on the accumulator. For example:

```
ASL A
```

will shift the accumulator left 1 bit.

Immediate addressing

The single-byte number following the opcode is to be used directly by the instruction. This addressing mode is marked by a '#' in front of the data. For example:

```
ORA #&80
```

will logically OR the contents of the accumulator with the singlebyte number '&80' (128).

Absolute addressing

The 2-byte number following the opcode gives the memory location of the data to be used by the instruction. For example:

```
LDY &2000
```

will load the Y register with the contents of memory location &2000.

Zero page addressing

The single-byte number following the opcode gives the memory location in page zero (&0000 to &00FF) of the data to be used by the instruction. This is similar to absolute addressing, except that the MSB of the address is always zero. This is faster than absolute addressing, and takes up only 2 bytes instead of 3 (including the opcode). For example:

```
STA &70
```

will store the contents of the accumulator into the zero page memory location &70.

Absolute indexed addressing

The unsigned contents of the specified index register are added to the 2-byte absolute address following the opcode, to give the location of the data to be used by the instruction. The index register used may be either X or Y, depending on which is allowed with the particular instruction. This addressing mode is marked by

a ‘, Y’ or a ‘,X’ following the data. It is useful for accessing tables or reading characters in from a line. For example:

```
DEC &3000,X
```

will decrement the location at &3000+X by 1. If the X register contained &54, the contents of location &3054 will be decremented.

Zero page indexed addressing

The contents of the specified index register are added to the single byte following the opcode, to give the page zero location of the data to be used by the instruction. The carry generated by this addition is ignored: the accessed location is always in page zero. For example:

```
INC &80,Y
```

will increment the contents of the location whose LSB is given by &80+X, and whose MSB is &00. Thus if Y contains &04, the contents of zero page location &84 will be incremented; if Y contains &FE, the contents of zero page location &7E will be incremented.

Relative addressing

The 2’s complement byte following the opcode is added to the program counter to give the location to be used by the instruction. This is only used by the conditional branch instructions. It means that the branch instructions only take up 2 bytes altogether, but the location which is being branched to must be a maximum of −128 to +127 away from the location of the instruction following the branch instruction. For example:

```
.loop  BEQ loop
```

will branch back to the same location if the Z flag is set. The byte following the opcode will be &FE (−2) for this instruction, because the branch instruction is 2 bytes back from the next instruction which would be executed if the branch did not take place.

Indirect addressing

The 2-byte absolute address following the opcode points to two consecutive bytes which contain the LSB and the MSB of the location to be used. The two bytes are stored LSB first, MSB second. This addressing mode is only used by the JMP instruction. For example:

```
JMP (&0200)
```

will jump to the location whose address is contained in &0200 (LSB) and &0201 (MSB).

Pre-indexed indirect addressing

The contents of the X register are added to the single byte following the opcode, to give the zero page location of two consecutive bytes (LSB first) which contain a pointer to the data. For example:

```
LDA (&50,X)
```

will use the number in &50+X (LSB) and &51+X (MSB) as a pointer to the number to be loaded into the accumulator. Thus if X contained &20, location &70 contained the number &34, and location &71 contained the number &12, the number in location &1234 would be loaded into the accumulator.

Post-indexed indirect addressing

The single byte following the opcode gives the zero page location of a 2-byte pointer (LSB first). The unsigned contents of the Y register are added to this pointer, to give the address to be used by the instruction. This instruction mode is very useful for pointing into memory: a pair of page zero locations hold the base of a pointer into memory, and Y holds the offset from that pointer. For example:

```
CMP (&2A),Y
```

will compare the accumulator with the byte pointed to by the base pointer in &2A (LSB) and &2B (MSB), offset by Y. Thus if &2A contains &00, and &2B contains &40, and Y contains &45, the accumulator will be compared with the contents of location &4045.

1.5 Addressing mode groups

A table of allowed addressing modes for each instruction is given on page 508 of the BBC User Guide, and the Electron User Guide details them in chapter 29. This section summarises the groups of instructions which use the same (or nearly the same) set of addressing modes.

These addressing mode groups are used extensively by the builtin assembler in BASIC. See chapter 6 for more on this.

Implied group

These instructions only use implied addresing. The instructions are:

BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA.

Relative branch group

These instructions only use relative addressing. The instructions are:

BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

Accumulator operation group

The instructions in this group are:

ADC, SBC, CMP, AND, EOR, ORA, LDA, STA

These instructions all operate on the accumulator, and allow the following addressing modes:

Immediate (not STA)
Zero page
Absolute
Zero page,X
Absolute,X
Absolute,Y
(Indirect,X)
(Indirect),Y

Shift group

The instructions in this group are:

ASL, LSR, ROL, ROR

and they allow the following addressing modes:

Accumulator
Zero page
Absolute
Zero page,X
Absolute,X

Count group

The instructions in this group are:

DEC, INC and they allow the following addressing modes:

Zero page
Absolute
Zero page,X
Absolute,X

Test group

The instructions in this group are:

BIT, CPX, CPY

and they allow the following addressing modes:

Immediate (not BIT)
Zero page
Absolute

Index load group

The instructions in this group are:

LDX, LDY

and they allow the following addressing modes:

Immediate
Zero page
Absolute
Zero page,X (‘,Y’ for LDX)
Absolute,X (‘,Y’ for LDX)

Index store group

The instructions in this group are:

STX, STY

and they allow the following addressing modes:

Zero page
Absolute
Zero page,X (‘,Y’ for STX)

Jump group

The instructions in this group are:

JMP, JSR

and they allow the following addressing modes:

Absolute
(Indirect) (not JSR)

1.6 The BASIC assembler

The BBC User Guide and the Electron User Guide give an adequate description of the use of the built-in assembler, so I won't cover it again here. However, BBC micro owners may not be aware of the extra facilities available on the assembler in BASIC 2, over that in BASIC 1 (which is the one described in the User Guide). These extra facilities are remote assembly, and the EQU directive.

1.6.1 Remote assembly The OPT directive controls the action of the assembler while it is in operation. The OPT is followed by a number whose lower 3 bits (only 2 bits in BASIC 1) set the assembler options. These bits are as follows:

Bit	**Option**
0	assembly listing if set
1	errors enabled if set
2	remote assembly if set

Remote assembly allows a machine code program to be assembled to run in one part of memory, but the code put in another. For example, an assembler routine which will be in a paged ROM can be assembled correctly for &8000 anwards, but the code can be placed at &2000 onwards, say, where there is RAM.

If this is being used, P% should be set up to point to the location where the routine will end up (&8000 in the above example), but O% should point to the location where the generated code is to be stored.

1.6.2 The EQU directives

This allows data to be incorporated as part of a machine code program, without having to leave the assembler. The directives available are:

EQUB	equate byte	reserves 1 byte
EQUW	equate word	reserves 2 bytes
EQUD	equate double word	reserves 4 bytes
EQUS	equate string	reserves a string

Note that the EQUS directive only reserves the space for the characters of the string; if a carriage return or CRLF is needed on the end, this must be done separately with an EQUB directive.

For example:

```
EQUB &40
EQUS "HI"
EQUW &1234
```

will reserve and initialise the following bytes in memory:

```
&40
&48 ("H")
&49 ("I")
&34
&12
```

Using the EQU directive is not only more convenient than using the BASIC equivalent, but it also makes the program much more readable. Many of the programs in this book use the EQU directive, although where it has been used, the alternative BASIC form is available for BASIC 1 users.