Servicing the BBC Micro

6502 Assembly Language
Assembly language

The 6502 microprocessor

The 6502 is the 'brains' of the computer containing all the logic required to recognise and execute the list of instructions called the program. All the time the machine is switched on the microprocessor is busy, reading numbers from memory, interpreting them as instructions and then carrying out the operations specified by these instructions. To help it with this task there are a number of special memory locations, called registers, on the microprocessor chip itself. These are identified by name rather than number, i.e. they are not part of the so-called Memory Map. The registers of the 6502 are indicated in Fig. 1.

![Diagram of 6502 microprocessor registers](image-url)

**Fig. 1.**
The accumulator A is the register involved in most of the mathematical and logical functions because of its greater power than other registers and memory locations. The X and Y registers are used to store values for counting, timing and indexing (identify an address or sequence of addresses referenced to some base address and particularly useful in scanning tables of values with the minimum of programming). The program counter, PC, registers the current address; the stack pointer keeps records of information put aside when the microprocessor is temporarily diverted from its main task, and the status register is a collection of individual bits identifying features of the previous instruction.

The program followed by the microprocessor bears little resemblance to BASIC. The only language the processor understands is the language of 0s and 1s, or MACHINE CODE. For example, the set of binary numbers below forms a short machine code program that stores the number 21 (hex) in memory location 1600 (hex):

<table>
<thead>
<tr>
<th>Binary</th>
<th>In Hex Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00</td>
</tr>
<tr>
<td>00010110</td>
<td>16</td>
</tr>
<tr>
<td>10001101</td>
<td>8D</td>
</tr>
<tr>
<td>00100001</td>
<td>21</td>
</tr>
<tr>
<td>10101001</td>
<td>A9</td>
</tr>
</tbody>
</table>

Some of these numbers are called OPERATION CODES or ‘OP CODES’, and tell the processor what it has to do. In the example, A9 tells the processor to load its accumulator with the next number, namely 21. The next code, 8D, tells it to store the contents of the accumulator in location 1600 (hex), the memory location defined by the next two numbers.

Although it is possible to write programs directly in machine code (in some early microcomputers it was the only method, e.g. KIM), it is a slow process, prone to error, requiring the programmer to make continuous reference to instruction tables similar to those shown in Appendix B. An alternative approach is to write programs in a more ‘human friendly’ format called ASSEMBLY LANGUAGE. This language uses alphabetic abbreviations for each type of instruction rather than binary or hex OP codes. For example, abbreviations such as LDA and STA are used to represent the operations Load the Accumulator and Store the Accumulator. These abbreviations are often called ‘mnemonics’ because they are more easily remembered than OP codes. Written in mnemonics, the example program becomes:

LDA #21
STA 1600
The question now arises, ‘How does this assembly language program become the machine code program stored in the computer’s memory?’ The answer is to use a special program called an ASSEMBLER which translates the ‘easily understood by humans’ assembly program into the language of the processor, machine code.

Using the assembler

Entering an assembly language program on the BBC micro is similar to entering a BASIC program. There are some extra instructions required that can best be explained with the aid of an example (Fig. 3).

10 P% = &1500
20 [  
30 LDA #21  
40 STA &1600  
50 RTS  
60 ]

Fig. 3.

—Line 10 acts as an ‘origin’ statement for the program, telling the assembler where the machine code has to be positioned in the computer’s memory. The integer variable P% is used for this task.
—Lines 20 and 60 contain square brackets (they appear as arrows in MODE 7) that enclose the Assembly Language program.
—Line 30 contains the instruction to Load the Accumulator with 21 (hex).
—Line 50 contains a Return from Subroutine instruction which returns control to BASIC on completion of the machine code program.

The RUN command will assemble the program, placing the machine code in memory. An assembler listing of the mnemonics and the machine code will also be sent to the display.

```
1500 49 21  LDA #$21
1502 80 00 16 STA $1600
1505 60  RTS
```

Fig. 4.

Examination of memory location $1600 will show that the program has not yet been executed. Try:

```
PRINT ~ ? $1600
```

i.e. it is unlikely that it contains $21.

To execute the machine code program we use the CALL statement followed by the starting address of the routine, i.e. type:

```
CALL $1500
```

The computer will execute the program and return to BASIC, displaying the '=>' prompt. Check memory location $1600 again. It should now contain $21.

The CALL statement can be included at the end of the program, see Fig. 5.

```
10 P% = $1500
20 [
30 LDA #$21
40 STA $1600
50 RTS
60 ]
70 CALL $1500
```

Fig. 5.
On the command RUN the program will be assembled and then executed.

Comments and labels
Documentation considerably improves a program, making it easier to read. In BASIC, comments are added using REM statements. Unfortunately REM statements are not allowed within the assembler program and comments must be attached to assembler statements using a semi-colon or backslash (\)

e.g. 40 STA &1600; THIS IS A COMMENT

Variable names can be used to represent memory addresses or memory contents. However, they must be defined outside the square brackets holding the assembler program

e.g. NUMBER=&21
      STORE =&1600

An exception to this rule is the definition of program addresses. If we wished to label the starting address of a program 'START' it could be done as follows.

30 START LDA #&21

The label is prefixed by a full stop (period) '.' and separated from the assembler mnemonic by at least one space.

Using comments and labels the previous program becomes that listed in Fig. 6.

10 REM **********************
20 REM DEMO ASSEMBLER PROGRAMME
30 REM **********************
40 PS=&1500
50 NUMBER=&21
60 STORE=&1600
70 [
80 .START LDA #NUMBER ;GET NUMBER
90 STA STORE ;STORE NUMBER
100 RTS ;BACK TO BASIC
110 ]
120 CALL START

Fig. 9.6.

Finally a word of caution on the choice of labels. The same restrictions are placed on variable names as in BASIC. In particular, BASIC 'keywords' like PRINT, NEXT, REPEAT, END, etc. are definitely NOT ALLOWED. However, their lower case equivalents are permitted.
The BBC micro's operating system ROM contains many useful machine code subroutines that can be included in your assembler program. Three routines of particular interest are:

1. **OSRDCH**—OPERATING SYSTEM READ CHARACTER Address—&FFE0:
   Reads a character from 'the input channel', normally the keyboard, placing it in the accumulator.

2. **OSWRCH**—OPERATING SYSTEM WRITE CHARACTER Address—&FFEE:
   Writes a character in the accumulator 'down the output channel', normally to the screen.

3. **OSASCI**—OPERATING SYSTEM ASCII Address—&FFE3:
   As for OSWRCH except that a line feed is automatically inserted with a carriage return.
   (Note: The X and Y registers are not affected by any of these routines.)

The example in Fig. 7 turns the computer into an electronic type-writer that will ignore all BASIC keywords and merely display depressed keys on the screen. The subroutine OSRDCH is used to obtain the ASCII code of any depressed key. OSASCI then transfers this code from the accumulator to the display. The JMP START instruction sends the processor back to the keyboard to look for another depressed key.

```plaintext
10REM ************************
20REM USING THE OS SUBROUTINES
30REM ************************
35 DIM SPACE 100
40 OSASCI=&FFE3
50 OSRDCH=&FFE0
60 PI=SPACE
70 [C
80 .START JSR OSRDCH ;GET CHART FROM KB
90 JSR OSASCI ;PLACE ON SCREEN
100 JMP START ;REPEAT
110 ]
120 CALL START
```

*Fig. 7.*

In this example we have used a different technique to instruct the assembler where to position the machine code. Rather than defining uniquely where the code has to be placed, the DIM statement in line
35 reserves 101 bytes and places the machine code in this reserved space at the end of the BASIC program. This technique has the advantage that the assembler will ensure that the machine code program is positioned in a safe place within the system memory and will not corrupt either the original BASIC program or memory locations allocated to the screen.

**Two pass assembly**
The BBC assembler uses the full stop to define labels within an assembly program. However, problems arise when a label is referred to before it is defined. This situation is illustrated in the example in Fig. 8 where the ‘typewriter’ program has been modified to return control to BASIC whenever the asterisk key is depressed:

```assembly
35 DIM SPACE 100
40 OSASCI=4FFE3
50 OSRDOCh=4FFE0
60 P SCALE
70 [
80 START JSR OSRDOCh ;GET CHART. FROM KB
85 CMP $ASC*** ;IS IT AN ***
87 BEQ FINI ;IF SO QUIT
90 JSR OSASCI ;PLACE ON SCREEN
100 JMP START ;REPEAT
105 FINI RTS ;RETURN TO BASIC
110 ]
120 CALL START
```

**Fig. 8.**

The label FINI appears in line 87 but is not defined until line 105. An attempt to assemble the program would give the result shown in Fig. 9

```assembly
OF2B
OF2B 20 E0 FF .START JSR OSRDOCh ;GET CHART. FROM KB
OF2B C9 2A CMP $ASC*** ;IS IT AN ***
```

**Fig. 9.**

No such variable at line 87

—namely the assembler stops, displaying an error message. The solution is to allow the assembler to pass through the source program twice. In the first pass the assembler establishes a table of labels and their addresses. In the second pass it uses these addresses to construct the final machine code program. Due to the assembler being ‘embedded’ in BASIC, the extra programming required to initiate the two
pass assembly is relatively simple, requiring only a FOR...NEXT loop to send the assembler through the program twice, see Fig. 10.

\[
\text{FOR } N = 1 \text{ TO } 2 \\
P\% = \text{SPACE} \\
\]

\[\text{Assembly language program}\]

\[
\text{NEXT } N \\
\]

Fig. 10.

However two points are worth noting:

(i) The equate statement for the origin P\% must be enclosed within the loop so that it is reset to the correct value at the start of each pass.

(ii) A 'OPT' statement is required to suppress error messages and prevent the assembler halting during the first pass. A number between 0 and 3 is used with this statement to give the following options during assembly:

OPT0 assembler errors suppressed, no listing
OPT1 assembler errors suppressed, listing
OPT2 assembler errors reported, no listing
OPT3 assembler errors reported, listing

A reasonable choice of options might be OPT1 during the first pass to suppress error messages and OPT3 during the second pass to display any errors still remaining. This is achieved in the example in Fig. 11 by placing the counter N after the OPT statement in line 70 and assigning it the values 1 and 3 in line 55:

\[
\begin{align*}
35 & \text{DIM SPACE 100} \\
40 & \text{OSASCI=EFFE3} \\
50 & \text{OSROCH=FFE0} \\
55 & \text{FOR } N=1 \text{ TO } 3 \text{ STEP } 2 \\
60 & \text{P\%=SPACE} \\
70 & \text{[OPTN} \\
80 & \text{.START JSR OSROCH ;GET CHART FROM KB} \\
85 & \text{CMP #ASC** ;IS IT AN **} \\
87 & \text{BEQ FINI ;IF SO QUIT} \\
90 & \text{JSR OSASCI ;PLACE ON SCREEN} \\
100 & \text{JMP START ;REPEAT} \\
105 & \text{.FINI RTS ;RETURN TO BASIC} \\
110 & \text{]} \\
115 & \text{NEXT N} \\
120 & \text{CALL START} \\
\end{align*}
\]

Fig. 11.
With these options the assembler produces a listing on the display during each pass. However the relative displacement of 06 at &0F4C is only resolved in the second listing.

```
0F46    OPTN
0F46  20 E0 FF .START JSR OSRCH ;GET CHART FROM KB
0F49  C9 2A CMP ASC*** ;IS IT AN ***
0F48  F0 FE  BEQ FINI ;IF SO QUIT
0F4D  20 E3 FF JSR OSASCI ;PLACE ON SCREEN
0F50  4C 46 0F JMP START ;REPEAT
0F53  00  .FINI RTS ;RETURN TO BASIC
0F46    OPTN
0F46  20 E0 FF .START JSR OSRCH ;GET CHART FROM KB
0F49  C9 2A CMP ASC*** ;IS IT AN ***
0F48  F0 06  BEQ FINI ;IF SO QUIT
0F4D  20 E3 FF JSR OSASCI ;PLACE ON SCREEN
0F50  4C 46 0F JMP START ;REPEAT
0F53  60  .FINI RTS ;RETURN TO BASIC
```

Fig. 12.

A final point worth noting is the use of the statements VDU 14 and VDU 15 to switch the ‘PAGE MODE’ on and off. This can prove useful when assembling large programs to examine the assembly listing a page or ‘screenful’ at a time.

### Mixing machine code and BASIC

Two statements allow control to pass to a machine code routine from BASIC—‘CALL’ and ‘USR’. With both statements the processors A, X, and Y registers are initialised to the least significant bytes of the integer variables A%, X% and Y% and the carry flag is set to the least significant bit of the variable C% on entry to the machine code routine.

```
e.g.  10  A% = &41
   20  CALL &FFEE
   30  END

10  A% = &41
20   Z = USR(&FFEE)
30   END
```

Each of these programs would send 41 hex or ASCII ‘A’ to the display routine OSWRCH at FFEE hex.

On completion of the machine code routine USR return a 32 bit number made up of the processor’s status, Y, X and A registers. For example, see Fig. 13.

The CALL statement offers greater flexibility, allowing program variables of all types to be passed to and from a machine code subroutine. To aid this transfer a ‘parameter block’, starting at &0600,
contains details of the number, location and type of variables to be passed. The block has the following structure:

0600—number of parameters
0601—low byte of address 1st parameter
0602—high byte of address 1st parameter
0603—code defining parameter type
0604—low byte of address 2nd parameter
0605—high byte of address 2nd parameter
0606—code defining parameter type
etc.

The codes used to define parameter types are:

0—8 bit byte (e.g. ?X)
4—32 bit integer variable (e.g. X%)  
5—40 bit floating point number (e.g. T)
128—A string at a defined address (e.g. $X$)
129—A string variable (e.g. A$)

In the example in Fig. 14 the structure of the parameter block is illustrated by passing two variables, B% and C%, in the subroutine CALL to &FFEE in line 60:
Fig. 14.

Lines 70 to 90 print details of the start of the parameter block in hex:

Address 0600 contains 02 – number of variables

0601 ,, 08 } ADL 1st variable
0602 ,, 04 } ADH 1st variable
0603 ,, 04 – code for integer variable
0604 ,, 0C } ADL 2nd variable
0605 ,, 04 } ADH 2nd variable
0606 ,, 04 – code for integer variable

FF

etc.

Investigating &0408 and &040C, we find the values of the two variables B% and C% (Fig. 15).

100 FOR N=&0408 TO &040F
110 PRINT "N"
120 NEXT N

33 }
22 }
11 }
0 }
77 }
66 }
55 }
44 }

Fig. 15.

The final example illustrates a machine code program that will convert and display decimal numbers between 0 and 255 as binary. The program begins by assembling and inserting the machine code program. A small BASIC program then calls the conversion utility, passing the parameter NUMBER % which is then displayed in binary.
10 DIM BINARY 100, TEMP 4
20 OSCASI = 1FFE3
30 FCR #0 TO 2 STEP 2
40 P% = BINARY
50 COPUTN
60 LDY #00; SETY = 0
70 LDA $0601; TRANSFER POINTERS TO ZERO PAGE
80 STA $80
90 LDA $0602
100 STA $81
110 LDA ($80), Y; GET NUMBER
120 STA TEMP; PLACE IN TEMPORARY STORE
130 LDX #08; USE COUNTER TO EXAMINE 8 BITS
140 .START BIT TEMP; "1" OR "0"
150 .SPL ZERO; BRANCH IF ITS A ZERO
160 LDA $ASC"1"; PRINT "1"
170 JSR OSCASI; TO DISPLAY
180 JMP ROTATE
190 .ZERO LDA $ASC"0"; PRINT "0"
200 JSR OSCASI
210 .ROTATE ROL TEMP; ROTATE BYTE LEFT
220 DEX; NEXT BIT
230 BNE START
240 LDA #00; C-RETURN TO DISPLAY
250 JSR OSCASI
260 RTS; BACK TO BASIC
270 ]
280 NEXT N
290 REM ***************
300 REM BASIC PROGRAMME
310 REM TO GENERATE
320 REM BINARY NUMBERS
330 REM ***************
340 FOR NUMBERS = 0 TO 16
350 CALL BINARY, NUMBERS
360 NEXT NUMBERS
370 END

> RUN
00000000
00000001
00000010
00000011
00000100
00000101
00000110
00000111
00001000
00001001
00001010
00001011
00001100
00001101
00001110
00001111
00010000

Fig. 16.
Driving graphics from machine code

All the BASIC keywords used to control the display have their equivalent VDU statement, e.g:

PRINT "A" is the same as VDU 65
MODE 5 is the same as VDU 22,5
COLOUR 3 is the same as VDU 17,3

etc.

The link between the BASIC VDU statement and operating system display routine OSWRCH is easily understood if the keyword 'VDU' is interpreted as 'SEND THE FOLLOWING BYTE (S) TO OSWRCH', i.e. the assembly language equivalent of

(a) PRINT "A" or VDU 65 is:       LDA #65
                                JSR OSWRCH
(b) MODE 5 or VDU 22,5 is:       LDA #22
                                JSR OSWRCH
                                LDA #5
                                JSR OSWRCH

The VDU statements use all 32 ASCII control codes (i.e. ASCII codes not used as symbols or alphanumeric characters). The first byte after the VDU statement, i.e. the first byte sent to OSWRCH, selects the desired display function. The operating system then knows how many more bytes are required to complete the instruction, e.g. MODE selection only requires one byte after the code, whereas redefining the shape of a display character requires 9.

The example program in Fig. 17 selects the display mode.

```
10 REM ***************
20 REM SELECTING SCREEN MODE
30 REM FROM AN ASSEMBLY
40 REM LANGUAGE ROUTINE
50 REM ***************
60 OSWRCH=$FEFE
70 DIM SPACE 100
80 INPUT "Which mode":M
90 PS=SPACE
100[
110 LDA #22   ;CONTROL CODE FOR MODE SELECT
120 JSR OSWRCH ;DOWN OUTPUT CHANNEL
130 LDA #M   ;SELECT MODE
140 JSR OSWRCH ;DOWN OUTPUT CHANNEL
150 RTS      ;BACK TO BASIC
160 ]
170 CALL SPACE
180 PRINT "This is mode":M
190 END
```

Fig. 17.
Using control codes as a means of selecting and driving different display functions adds greatly to the BBC micro's flexibility. It can be adapted as a colour graphics terminal communicating through either its RS423 serial port to a larger mainframe computer, or through its own system bus, called the 'Tube', to a second processor option.
MOS TECHNOLOGY MCS6502 SERIES

The MOS Technology 6502 series microprocessors are nMOS 8-bit types, of which the 6502 is probably the most commonly found. Other processors in this series are mainly simplified variants designed to fit into smaller packages.

In many respects the basic design philosophy of the 6502 follows the same lines as that of the Motorola 6800 series. The 6502 is a slightly less complex processor in terms of its architecture, but it can in some respects be considered as an enhanced version of the 6800, particularly in its comprehensive range of addressing modes.

Because of the similar hardware design, the bus systems for the 6502 and 6800 appear to be the same, but in fact they are not directly compatible. Generally the support chips for the 6800 can readily be used with a 6502 CPU and the reverse is also true, although in some cases additional external logic may be required. The instruction sets may also appear to be similar, but are totally incompatible as far as machine code is concerned.

Some of the wide popularity of the 6502 series can be attributed to their use in such popular personal computer systems as the CBM PET and the Apple II.

Prime manufacturer
MOS Technology Inc., which is a subsidiary of Commodore Business Machines (CBM).

Devices available
- MCS6502: Basic type 65k address on-chip clock
- MCS6512: As 6502 but external clock
- MCS6503: 4k address range on-chip clock
- MCS6504: 8k address range on-chip clock, no NMI
- MCS6505: 4k address range on-chip clock, no NMI
- MCS6506: 4k address range on-chip clock, no NMI
- MCS6507: 8k address range on-chip clock, no interrupts
- MCS6513: As 6503 but external clock
- MCS6514: As 6504 but external clock
- MCS6515: As 6505 but external clock

Alternative source devices
- Rockwell: R6502, R6503, R6504, R6505, R6506, R6507, R6512, R6513, R6514, R6515
- Synertek: SY6502, SY6503, SY6504, SY6505, SY6506, SY6507, SY6512, SY6513, SY6514, SY6515
- EMM-Semi: 6502, 6503, 6504, 6505, 6506, 6507, 6512, 6513, 6514, 6515

Note that all of the 6502 series types are available with various clock speed options, with versions for 1 MHz, 2 MHz and 3 MHz maximum clock frequency.

Architecture

If the architecture diagram for the 6502 (fig. 3.17) is compared with that of the 6800 it will be seen that the 6502 is similar in design to the 6800, though rather less complex.

Only one 8-bit accumulator is provided, compared with the two accumulators of the 6800, and this handles all arithmetic and logic operations via the ALU. Although slightly less flexible when dealing with 16-bit numbers, the single 8-bit accumulator is perfectly adequate for all normal computing requirements.

An 8-bit status register provides flags for zero, minus, carry and overflow results of operations, and for the interrupt, break and decimal modes.

Unlike the 6800 the 6502 has two 8-bit index registers rather than a single 16-bit index register. This limits the index range to 256 but provides much greater flexibility in dealing with data tables.

The stack pointer of the 6502 has only 8 bits and the stack is always located within page 1 of the memory map. It is possible to have any stack length up to 256 bytes and a number of separate stacks may be set up within page 1. This is slightly less flexible than the 6800, where the stacks may be set up anywhere in memory, but is perfectly adequate.

As with the 6800 there are no general purpose registers provided in the 6502, since it uses general memory locations for this purpose. Similarly all input–output devices will be treated simply as memory locations by the processor.

In the 6502 and 6512 the program counter register is 16 bits wide, allowing up to 65k of memory to be addressed. In other devices of the series the program counter length is cut to 12 or 13 bits, allowing either 4k or 8k of address space.

Like the 6800 the bus system of the 6502 comprises an 8-bit bidirectional data bus, a 16-bit address bus and some control signals. All operations are controlled by a 2-phase clock, and memory access is made on phase 2 of each cycle of the clock. Internal operations occur during phase 1.

The basic memory map for the 6502 is:
Vectors for int. and reset
Main user space
Stack area
Zero page

Package
The 6502 and 6512 are supplied in 40-pin dual in line. All other types use a 28-pin dual in line package. All types use a plastic encapsulation.

Pin connections

### 6502 and 6512

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$V_a$</td>
</tr>
<tr>
<td>2</td>
<td>RDY</td>
</tr>
<tr>
<td>3</td>
<td>$\phi_1$</td>
</tr>
<tr>
<td>4</td>
<td>IRQ</td>
</tr>
<tr>
<td>5</td>
<td>No conn. (6502)</td>
</tr>
<tr>
<td>6</td>
<td>NMI</td>
</tr>
<tr>
<td>7</td>
<td>SYNC</td>
</tr>
<tr>
<td>8</td>
<td>$V_{cc}$</td>
</tr>
<tr>
<td>9</td>
<td>AB0</td>
</tr>
<tr>
<td>10</td>
<td>AB1</td>
</tr>
<tr>
<td>11</td>
<td>AB2</td>
</tr>
<tr>
<td>12</td>
<td>AB3</td>
</tr>
<tr>
<td>13</td>
<td>AB4</td>
</tr>
<tr>
<td>14</td>
<td>AB5</td>
</tr>
<tr>
<td>15</td>
<td>No conn. (6512)</td>
</tr>
<tr>
<td>16</td>
<td>AB6</td>
</tr>
<tr>
<td>17</td>
<td>AB7</td>
</tr>
<tr>
<td>18</td>
<td>AB8</td>
</tr>
<tr>
<td>19</td>
<td>AB9</td>
</tr>
<tr>
<td>20</td>
<td>AB10</td>
</tr>
<tr>
<td>21</td>
<td>$V_m$</td>
</tr>
<tr>
<td>22</td>
<td>AB12</td>
</tr>
<tr>
<td>23</td>
<td>AB13</td>
</tr>
<tr>
<td>24</td>
<td>AB14</td>
</tr>
<tr>
<td>25</td>
<td>AB15</td>
</tr>
<tr>
<td>26</td>
<td>DB7</td>
</tr>
<tr>
<td>27</td>
<td>DB6</td>
</tr>
<tr>
<td>28</td>
<td>DB5</td>
</tr>
<tr>
<td>29</td>
<td>DB4</td>
</tr>
<tr>
<td>30</td>
<td>DB3</td>
</tr>
<tr>
<td>31</td>
<td>DB2</td>
</tr>
<tr>
<td>32</td>
<td>DB1</td>
</tr>
<tr>
<td>33</td>
<td>DB0</td>
</tr>
<tr>
<td>34</td>
<td>R/W</td>
</tr>
<tr>
<td>35</td>
<td>No conn.</td>
</tr>
<tr>
<td>36</td>
<td>No conn. (6502)</td>
</tr>
<tr>
<td>37</td>
<td>$\phi_0$ (6502)</td>
</tr>
<tr>
<td>38</td>
<td>S.O.</td>
</tr>
<tr>
<td>39</td>
<td>$\phi_2$ OUT</td>
</tr>
<tr>
<td>40</td>
<td>RESET</td>
</tr>
</tbody>
</table>

### 6503 28 pin

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESET</td>
</tr>
<tr>
<td>2</td>
<td>$V_a$</td>
</tr>
<tr>
<td>3</td>
<td>IRQ</td>
</tr>
<tr>
<td>4</td>
<td>NMI</td>
</tr>
<tr>
<td>5</td>
<td>$V_{cc}$</td>
</tr>
<tr>
<td>6</td>
<td>AB0</td>
</tr>
<tr>
<td>7</td>
<td>AB1</td>
</tr>
<tr>
<td>8</td>
<td>AB2</td>
</tr>
<tr>
<td>9</td>
<td>AB3</td>
</tr>
<tr>
<td>10</td>
<td>AB4</td>
</tr>
<tr>
<td>11</td>
<td>AB5</td>
</tr>
<tr>
<td>12</td>
<td>AB6</td>
</tr>
<tr>
<td>13</td>
<td>AB7</td>
</tr>
<tr>
<td>14</td>
<td>AB8</td>
</tr>
<tr>
<td>15</td>
<td>AB9</td>
</tr>
<tr>
<td>16</td>
<td>AB10</td>
</tr>
<tr>
<td>17</td>
<td>AB11</td>
</tr>
</tbody>
</table>

### 6504/6507 28 pin

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESET</td>
</tr>
<tr>
<td>2</td>
<td>$V_{ss}$</td>
</tr>
<tr>
<td>15</td>
<td>AB10</td>
</tr>
<tr>
<td>16</td>
<td>AB11</td>
</tr>
</tbody>
</table>

Signal functions

- **DB0 – DB7**: Bidirectional data bus
- **AB0 – AB15**: Address bus (output)
- **$V_a$, $V_{cc}$**: Power supplies
- **R/W**: Read-write (low = write)
- **IRQ, NMI**: Interrupt req. inputs active low
- **RDY**: Ready input used to halt CPU
- **SYNC**: Output (1 during instruction fetch)
- **RESET**: Reset input (active low)
- **S.O.**: Set overflow input
- **$\phi_0$, $\phi_1$, $\phi_2$**: Clock signals
- **DBE**: Data bus enable (active high)

Power requirements

- $V_a = 0$ V
- $V_{cc} = +5$ V ± 5%
- Power dissipation 700 – 800 mW

Signal levels

- Inputs are TTL compatible 300 $\mu$A loading
- Outputs will drive one TTL load
- Data bus is tri-state

Input–output

The 6502 series treat all input–output as memory locations, data being presented or accepted via the data bus.

Interrupt facilities

The 6502 provides both maskable (IRQ) and non-maskable (NMI) interrupts. There is also a software interrupt facility using the BRK instruction. On an interrupt execution the program counter and status register are pushed to the stack. These are restored by the RTI instruction at the end of the interrupt routine. BRK is the same as IRQ, but not maskable and sets a flag bit in the status register. Interrupt vector addresses are stored at the top of memory as shown:
FFFF IRQ vector (MSB)
FFFE IRQ vector (LSB)
FFFD Reset vector (MSB)
FFFC Reset vector (LSB)
FFFB NMI vector (MSB)
FFFA NMI vector (LSB)

Reset causes a reset sequence within the CPU and the instruction address is obtained from FFFC/FFFD.

Multilevel interrupt operation is readily achieved and priorities may be dealt with either by polling software or by external hardware.

Instruction set
The 6502 instruction set contains 52 different instructions, and at first sight may appear to be very similar to that for the 6800 series microprocessors. Instructions may have one, two or three bytes.

Arithmetic and logic
Addition and subtraction with carry or borrow are provided using the 8-bit accumulator. A decimal mode also allows addition and subtraction of BCD format numbers. There are no complement or negate instructions and the accumulator cannot be directly incremented or decremented, although memory locations can.

AND, OR and EXCLUSIVE OR operations can be carried out between accumulator and memory. There are also shift and rotate left and right instructions for both memory and accumulator.

Branch and jump
A useful series of conditional branch instructions is provided, although this is not as extensive as those on the 6800. Status register bits may be set and reset by program. Tests for zero, negative, carry and overflow are provided.

Only unconditional jump and jump to subroutine are available. A subroutine jump automatically stores the return address on the stack.

Register and transfer operations
Data can readily be transferred between the A accumulator, the X and Y index registers and the stack pointer, or memory. Push and pull instructions allow data from the accumulator or status register to be transferred to the stack. Both index registers may be incremented or decremented.

Memory or accumulator words may be tested bit by bit if desired.

Timing
Like the 6800, the 6502 series uses a 2-phase processor clock, and all memory access is carried out during φ2 clock cycles. Most instructions take 2, 3 or 4 clock cycles and may use 1, 2 or 3 bytes of machine code.

The standard parts operate with a 1 MHz clock, giving instruction execution times of some 2 – 4 μs. Special high speed parts are available with clock frequencies of 2 or 3 MHz. These are coded with suffix A (6502A) for 2 MHz operation or suffix B (6502B) for 3 MHz operation.

Types 6502 to 6507 have on-chip clock phase generators but need an external crystal oscillator to provide the φ0 input, whilst types 6512 to 6515 require an external 2-phase non-overlapping clock signal applied to the φ1 and φ2 clock inputs.

Support devices
A wide range of support devices is available for the 6502 series microprocessors. Some of these are:

6520 PIA Two 8-bit bidirectional programmable ports (identical to 6820)
6522 Versatile interface adapter (VIA) 2 × 8-bit ports as in 6520, plus 2 × 16-bit interval timers and a serial I/O facility
6530 1k ROM, 64-byte RAM, 2 × 8-bit parallel ports plus an 8-bit interval timer.
6531 2k ROM, 128 byte RAM, 2 × 8-bit parallel I/O, serial I/O and a 16-bit timer/counter
6532 128-byte RAM, 2 × 8-bit parallel I/O ports, 8-bit timer
6541 Keyboard/display controller
6545 Raster scan CRT controller
6551 Asynchronous serial I/O
6591 Floppy disk controller

The 6502 series may also be used with most of the 6800 series support devices. Some care may be needed with address decoding, however, since the 6500 has its lower address byte in the lower memory location whilst the 6800 stores its addresses in memory with the high address byte first.

Development aids
MOS Technology
KIM1 Stand alone board with keypad and LED displays
- - - - //
NZC1DA
Load index X with memory

LDX

- - - - //
NZC1DA
Load accumulator with memory

TDA

- - - - //
NZC1DA
Jump to new location

JMP

- - - - //
NZC1DA
Execute DR memory with accumulator

FOR
<table>
<thead>
<tr>
<th>Code</th>
<th>3 6 9 12 2 5 8 10</th>
<th>3 6 9 12 2 5 8 10</th>
<th>3 6 9 12 2 5 8 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>ROL, ORP, X</td>
<td>ROL, ORP, X</td>
<td>ROL, ORP, X</td>
</tr>
<tr>
<td>Address</td>
<td>V.A.</td>
<td>V.A.</td>
<td>V.A.</td>
</tr>
<tr>
<td>Mode</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**ROL**

Rearranges the bits of the contents of the specified memory location, considering it as an eight-bit quantity. The result is written back into the same location.

**Addressing Mode**

- [0] V.A.

**Example:**

```
ROL X
```

**Description:**

The ROL instruction rotates the bits of the contents of memory location X to the left by one position. The original contents of X are shifted to the left by one bit, and the shifted bit is written back into the location. This operation is commonly used for manipulating binary data or implementing shift operations in programming.

---

**NOP**

No operation.

**Addressing Mode**

- [0] V.A.

**Example:**

```
NOP
```

**Description:**

The NOP instruction does nothing. It is used as a placeholder in the program code to ensure that the program advances to the next instruction without doing anything.

---

**PHA**

Push accumulator on stack.

**Addressing Mode**

- [0] V.A.

**Example:**

```
PHA
```

**Description:**

The PHA instruction pushes the contents of the accumulator onto the stack. This operation is useful for saving the current value of the accumulator before changing its value in the program.

---

**ORA**

OR memory with accumulator.

**Addressing Mode**

- [0] V.A.

**Example:**

```
ORA Y
```

**Description:**

The ORA instruction performs a bitwise OR operation between the contents of memory location Y and the accumulator. The result is written back into the accumulator. This operation is commonly used for combining two binary values or setting conditions in the program.
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NzCIdv</td>
<td>SEI</td>
<td>SET NZ/IC code</td>
</tr>
<tr>
<td>NzCIdv</td>
<td>SED</td>
<td>Set Dec mode</td>
</tr>
<tr>
<td>NzCIdv</td>
<td>SEC</td>
<td>Set Carry Flag</td>
</tr>
</tbody>
</table>

**NZCIdv**

Store accumulators in memory

---

**NZCIdv**

Return from subroutine

---

**NZCIdv**

Return from interrupt

---

**NZCIdv**

ROr

Rotate one bit right (memory or accumulator)
<table>
<thead>
<tr>
<th>$D6$ - DEC — Zero Page, X</th>
<th>$E8$ - INX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D7$ - Future Expansion</td>
<td>$F3$ - Future Expansion</td>
</tr>
<tr>
<td>$D8$ - CLD</td>
<td>$F4$ - Future Expansion</td>
</tr>
<tr>
<td>$D9$ - CMP — Absolute, Y</td>
<td>$F5$ - SBC — Zero Page, X</td>
</tr>
<tr>
<td>$DA$ - Future Expansion</td>
<td>$F6$ - INC — Zero Page, X</td>
</tr>
<tr>
<td>$DB$ - Future Expansion</td>
<td>$F7$ - Future Expansion</td>
</tr>
<tr>
<td>$DC$ - Future Expansion</td>
<td>$F8$ - SED</td>
</tr>
<tr>
<td>$DD$ - CMP — Absolute, X</td>
<td>$F9$ - SBC — Absolute, Y</td>
</tr>
<tr>
<td>$DE$ - DEC — Absolute, X</td>
<td>$FA$ — Future Expansion</td>
</tr>
<tr>
<td>$DF$ - Future Expansion</td>
<td>$FB$ — Future Expansion</td>
</tr>
<tr>
<td>$E0$ - CPX — Immediate</td>
<td>$FC$ — Future Expansion</td>
</tr>
<tr>
<td>$E1$ - SBC — (Indirect, X)</td>
<td>$FD$ — SBC — Absolute, X</td>
</tr>
<tr>
<td>$E2$ - Future Expansion</td>
<td>$FE$ — INC — Absolute, X</td>
</tr>
<tr>
<td>$E3$ - Future Expansion</td>
<td>$FF$ — Future Expansion</td>
</tr>
<tr>
<td>$E4$ - CPX — Zero Page</td>
<td>$F9$ - SBC — Absolute, Y</td>
</tr>
<tr>
<td>$E5$ - SBC — Zero Page</td>
<td>$FA$ — Future Expansion</td>
</tr>
<tr>
<td>$E6$ - INC — Zero Page</td>
<td>$FB$ — Future Expansion</td>
</tr>
<tr>
<td>$E7$ - Future Expansion</td>
<td>$FC$ — Future Expansion</td>
</tr>
<tr>
<td>$E8$ - INX</td>
<td>$FD$ — SBC — Absolute, X</td>
</tr>
<tr>
<td>$E9$ - SBC — Immediate</td>
<td>$FE$ — INC — Absolute, X</td>
</tr>
<tr>
<td>$EA$ — NOP</td>
<td>$FF$ — Future Expansion</td>
</tr>
</tbody>
</table>