6502 Instruction Set Guide

---

by Andrew John Jacobs

This is the printed version of the 6502 Instruction Set Guide
published by the late Andrew Jacobs on his website obelisk.me.uk

Edited by Dion Olsthoorn – October 2021

Table of Contents

Introduction

Many software engineers beginning work today will only ever work with high-level, object-oriented programming languages like Java and C++. They will never know the deep joy (and massive frustration) of taking on a microprocessor in the 'hand-to-hand' combat that is assembly language programming.

It concerns me that some of the skills and tricks that where quite common place when I started my career as code monkey, are completely overlooked today. Some of the forum posts I see by Computer Science and Electrical Engineering students make me wonder if they still teach binary.

This guide contains a description of the 6502 microprocessor, its instructions and assembly language. It also includes example algorithms that show common techniques and tricks to produce tight, compact code.

The content is split up into 6 sections:

- **Architecture**
  describes the few basic details of the 6502 processor.
- **Registers**
  goes over each of the 6502's internal registers and their use.
- **Instructions**
  gives a summary of whole instruction set.
- **Addressing**
  describes each of the 6502 memory addressing modes.
- **Algorithms**
  contains some examples of basic 6502 assembly coding.
- **Reference**
  describes the complete instruction set in detail.

6502 Basic Architecture

The 6502 microprocessor is a relatively simple 8-bit CPU with only a few internal registers capable of addressing at most 64Kb of memory via its 16-bit address bus. The processor is little endian and expects addresses to be stored in memory least significant byte first.

The first 256-byte page of memory ($0000-$00FF) is referred to as 'Zero Page' and is the focus of a number of special addressing modes that result in shorter (and quicker) instructions or allow indirect access to the memory. The second page of memory ($0100-$01FF) is reserved for the system stack and which cannot be relocated.

The only other reserved locations in the memory map are the very last 6 bytes of memory $FFFA to $FFFF which must be programmed with the addresses of the non-maskable interrupt handler ($FFFA/B), the power on reset location ($FFFC/D) and the BRK/interrupt request handler ($FFFE/F) respectively.

The 6502 does not have any special support of hardware devices so they must be mapped to regions of memory in order to exchange data with the hardware latches.

The Registers

The 6502 has only a small number of registers compared to other processors of the same era. This makes it especially challenging to program as algorithms must make efficient use of both registers and memory.

Program Counter

The program counter is a 16 bit register which points to the next instruction to be executed. The value of program counter is modified automatically as instructions are executed.

The value of the program counter can be modified by executing a jump, a relative branch or a subroutine call to another memory address or by returning from a subroutine or interrupt.

Stack Pointer

The processor supports a 256-byte stack located between $0100 and $01FF. The stack pointer is an 8 bit register and holds the low 8 bits of the next free location on the stack. The location of the stack is fixed and cannot be moved.

Pushing bytes to the stack causes the stack pointer to be decremented. Conversely pulling bytes causes it to be incremented.

The CPU does not detect if the stack is overflowed by excessive pushing or pulling operations and will most likely result in the program crashing.

Accumulator

The 8-bit accumulator is used all arithmetic and logical operations (with the exception of increments and decrements). The contents of the accumulator can be stored and retrieved either from memory or the stack.

Most complex operations will need to use the accumulator for arithmetic and efficient optimization of its use is a key feature of time critical routines.

Index Register X

The 8-bit index register is most commonly used to hold counters or offsets for accessing memory. The value of the X register can be loaded and saved in memory, compared with values held in memory or incremented and decremented.

The X register has one special function. It can be used to get a copy of the stack pointer or change its value.

Index Register Y

The Y register is similar to the X register in that it is available for holding counter or offsets memory access and supports the same set of memory load, save and compare operations as wells as increments and decrements. It has no special functions.

Processor Status

As instructions are executed a set of processor flags are set or clear to record the results of the operation. This flags and some additional control flags are held in a special status register. Each flag has a single bit within the register.

Instructions exist to test the values of the various bits, to set or clear some of them and to push or pull the entire set to or from the stack.

- **Carry Flag**

  The carry flag is set if the last operation caused an overflow from bit 7 of the result or an underflow from bit 0. This condition is set during arithmetic, comparison and during logical shifts. It can be explicitly set using the 'Set Carry Flag' (SEC) instruction and cleared with 'Clear Carry Flag' (CLC).

- **Zero Flag**

  The zero flag is set if the result of the last operation as was zero.

- **Interrupt Disable**

  The interrupt disable flag is set if the program has executed a 'Set Interrupt Disable' (SEI) instruction. While this flag is set the processor will not respond to interrupts from devices until it is cleared by a 'Clear Interrupt Disable' (CLI) instruction.

- **Decimal Mode**

  While the decimal mode flag is set the processor will obey the rules of Binary Coded Decimal (BCD) arithmetic during addition and subtraction. The flag can be explicitly set using 'Set Decimal Flag' (SED) and cleared with 'Clear Decimal Flag' (CLD).
  Note that only two instructions are affected by the D flag: ADC and SBC.

- **Break Command**

  The break command bit is set when a BRK instruction has been executed and an interrupt has been generated to process it.

- **Overflow Flag**

  The overflow flag is set during arithmetic operations if the result has yielded an invalid 2's complement result (e.g. adding to positive numbers and ending up with a negative result: 64 + 64 => -128). It is determined by looking at the carry between bits 6 and 7 and between bit 7 and the carry flag.

- **Negative Flag**

  The negative flag is set if the result of the last operation had bit 7 set to a one.

The Instruction Set

The 6502 has a relatively basic set of instructions, many having similar functions (e.g. memory access, arithmetic, etc.). The following sections list the complete set of 56 instructions in functional groups.

Load/Store Operations

These instructions transfer a single byte between memory and one of the registers. Load operations set the negative (N) and zero (Z) flags depending on the value of transferred. Store operations do not affect the flag settings.

LDA	Load Accumulator	N,Z
LDX	Load X Register	N,Z
LDY	Load Y Register	N,Z
STA	Store Accumulator	
STX	Store X Register	
STY	Store Y Register	

Register Transfers

The contents of the X and Y registers can be moved to or from the accumulator, setting the negative (N) and zero (Z) flags as appropriate.

TAX	Transfer accumulator to X	N,Z
TAY	Transfer accumulator to Y	N,Z
TXA	Transfer X to accumulator	N,Z
TYA	Transfer Y to accumulator	N,Z

Stack Operations

The 6502 microprocessor supports a 256-byte stack fixed between memory locations $0100 and $01FF. A special 8-bit register, S, is used to keep track of the next free byte of stack space. Pushing a byte on to the stack causes the value to be stored at the current free location (e.g. $0100,S) and then the stack pointer is post decremented. Pull operations reverse this procedure.

The stack register can only be accessed by transferring its value to or from the X register. Its value is automatically modified by push/pull instructions, subroutine calls and returns, interrupts and returns from interrupts.

TSX	Transfer stack pointer to X	N,Z
TXS	Transfer X to stack pointer	
PHA	Push accumulator on stack	
PHP	Push processor status on stack	
PLA	Pull accumulator from stack	N,Z
PLP	Pull processor status from stack	All

Logical

The following instructions perform logical operations on the contents of the accumulator and another value held in memory. The BIT instruction performs a logical AND to test the presence of bits in the memory value to set the flags but does not keep the result.

AND	Logical AND	N,Z
EOR	Exclusive OR	N,Z
ORA	Logical Inclusive OR	N,Z
BIT	Bit Test	N,V,Z

Arithmetic

The arithmetic operations perform addition and subtraction on the contents of the accumulator. The compare operations allow the comparison of the accumulator and X or Y with memory values.

ADC	Add with Carry	N,V,Z,C
SBC	Subtract with Carry	N,V,Z,C
CMP	Compare accumulator	N,Z,C
CPX	Compare X register	N,Z,C
CPY	Compare Y register	N,Z,C

Increments & Decrements

Increment or decrement a memory location or one of the X or Y registers by one setting the negative (N) and zero (Z) flags as appropriate.

INC	Increment a memory location	N,Z
INX	Increment the X register	N,Z
INY	Increment the Y register	N,Z
DEC	Decrement a memory location	N,Z
DEX	Decrement the X register	N,Z
DEY	Decrement the Y register	N,Z

Shifts

Shift instructions cause the bits within either a memory location or the accumulator to be shifted by one bit position. The rotate instructions use the contents if the carry flag (C) to fill the vacant position generated by the shift and to catch the overflowing bit. The arithmetic and logical shifts shift in an appropriate 0 or 1 bit as appropriate but catch the overflow bit in the carry flag (C).

ASL	Arithmetic Shift Left	N,Z,C
LSR	Logical Shift Right	N,Z,C
ROL	Rotate Left	N,Z,C
ROR	Rotate Right	N,Z,C

Jumps & Calls

The following instructions modify the program counter causing a break to normal sequential execution. The JSR instruction pushes the old PC onto the stack before changing it to the new location allowing a subsequent RTS to return execution to the instruction after the call.

JMP	Jump to another location	
JSR	Jump to a subroutine	
RTS	Return from subroutine	

Branches

Branch instructions break the normal sequential flow of execution by changing the program counter if a specified condition is met. All the conditions are based on examining a single bit within the processor status.

BCC	Branch if carry flag clear	
BCS	Branch if carry flag set	
BEQ	Branch if zero flag set	
BMI	Branch if negative flag set	
BNE	Branch if zero flag clear	
BPL	Branch if negative flag clear	
BVC	Branch if overflow flag clear	
BVS	Branch if overflow flag set	

Branch instructions use relative address to identify the target instruction if they are executed. As relative addresses are stored using a signed 8-bit byte the target instruction must be within 126 bytes before the branch or 128 bytes after the branch.

Status Flag Changes

The following instructions change the values of specific status flags.

CLC	Clear carry flag	C
CLD	Clear decimal mode flag	D
CLI	Clear interrupt disable flag	I
CLV	Clear overflow flag	V
SEC	Set carry flag	C
SED	Set decimal mode flag	D
SEI	Set interrupt disable flag	I

System Functions

The remaining instructions perform useful but rarely used functions.

BRK	Force an interrupt	B
NOP	No Operation	
RTI	Return from Interrupt	All

Addressing Modes

The 6502 processor provides several ways in which memory locations can be addressed. Some instructions support several different modes while others may only support one. In addition, the two index registers cannot always be used interchangeably. This lack of orthogonality in the instruction set is one of the features that makes the 6502 trickier to program well.

Implicit

For many 6502 instructions the source and destination of the information to be manipulated is implied directly by the function of the instruction itself and no further operand needs to be specified. Operations like 'Clear Carry Flag' (CLC) and 'Return from Subroutine' (RTS) are implicit.

Accumulator

Some instructions have an option to operate directly upon the accumulator. The programmer specifies this by using a special operand value, 'A'. For example:

```
LSR A           ;Logical shift right one bit
ROR A           ;Rotate right one bit
```

Immediate

Immediate addressing allows the programmer to directly specify an 8-bit constant within the instruction. It is indicated by a '#' symbol followed by an numeric expression. For example:

```
LDA #10         ;Load 10 ($0A) into the accumulator
LDX #LO LABEL   ;Load the LSB of a 16 bit address into X
LDY #HI LABEL   ;Load the MSB of a 16 bit address into Y
```

Zero Page

An instruction using zero page addressing mode has only an 8-bit address operand. This limits it to addressing only the first 256 bytes of memory (e.g. $0000 to $00FF) where the most significant byte of the address is always zero. In zero-page mode only the least significant byte of the address is held in the instruction making it shorter by one byte (important for space saving) and one less memory fetch during execution (important for speed).

An assembler will automatically select zero page addressing mode if the operand evaluates to a zero-page address and the instruction supports the mode (not all do).

```
LDA $00         ;Load accumulator from $00
ASL ANSWER      ;Shift labelled location ANSWER left
```

Zero Page,X

The address to be accessed by an instruction using indexed zero page addressing is calculated by taking the 8-bit zero-page address from the instruction and adding the current value of the X register to it. For example if the X register contains $0F and the instruction LDA $80,X is executed then the accumulator will be loaded from $008F (e.g. $80 + $0F => $8F).

NB: The address calculation wraps around if the sum of the base address and the register exceed $FF. If we repeat the last example but with $FF in the X register then the accumulator will be loaded from $007F (e.g. $80 + $FF => $7F) and not $017F.

```
STY $10,X   ;Save the Y register at location on zero page
AND TEMP,X  ;Logical AND accumulator with a zero page value
```

Zero Page,Y

The address to be accessed by an instruction using indexed zero page addressing is calculated by taking the 8-bit zero page address from the instruction and adding the current value of the Y register to it. This mode can only be used with the LDX and STX instructions.

```
LDX $10,Y   ;Load the X register from a location on zero page
STX TEMP,Y  ;Store the X register in a location on zero page
```

Relative

Relative addressing mode is used by branch instructions (e.g. BEQ, BNE, etc.) which contain a signed 8 bit relative offset (e.g. -128 to +127) which is added to program counter if the condition is true. As the program counter itself is incremented during instruction execution by two the effective address range for the target instruction must be with -126 to +129 bytes of the branch.

```
BEQ LABEL       ;Branch if zero flag set to LABEL
BNE *+4         ;Skip over the following 2 byte instruction
```

Absolute

Instructions using absolute addressing contain a full 16 bit address to identify the target location.

```
JMP $1234       ;Jump to location $1234
JSR WIBBLE      ;Call subroutine WIBBLE
```

Absolute,X

The address to be accessed by an instruction using X register indexed absolute addressing is computed by taking the 16-bit address from the instruction and added the contents of the X register. For example if X contains $92 then an STA $2000,X instruction will store the accumulator at $2092 (e.g. $2000 + $92).

```
STA $3000,X     ;Store accumulator between $3000 and $30FF
ROR CRC,X       ;Rotate right one bit
```

Absolute,Y

The Y register indexed absolute addressing mode is the same as the previous mode only with the contents of the Y register added to the 16-bit address from the instruction.

```
AND $4000,Y    ;Perform a logical AND with a byte of memory
STA MEM,Y      ;Store accumulator in memory
```

Indirect

JMP is the only 6502 instruction to support indirection. The instruction contains a 16-bit address which identifies the location of the least significant byte of another 16-bit memory address which is the real target of the instruction.

For example, if location $0120 contains $FC and location $0121 contains $BA then the instruction JMP ($0120) will cause the next instruction execution to occur at $BAFC (e.g. the contents of $0120 and $0121).

```
JMP ($FFFC)     ;Force a power on reset
JMP (TARGET)    ;Jump via a labelled memory area
```

Indexed Indirect

Indexed indirect addressing is normally used in conjunction with a table of address held on zero page. The address of the table is taken from the instruction and the X register added to it (with zero-page wrap around) to give the location of the least significant byte of the target address.

```
LDA ($40,X)     ;Load a byte indirectly from memory
STA (MEM,X)     ;Store accumulator indirectly into memory
```

Indirect Indexed

Indirect indexed addressing is the most common indirection mode used on the 6502. In instruction contains the zero-page location of the least significant byte of 16 bit address. The Y register is dynamically added to this value to generated the actual target address for operation.

```
LDA ($40),Y     ;Load a byte indirectly from memory
STA (DST),Y     ;Store accumulator indirectly into memory
```

Coding Algorithms

As you can see from the preceding descriptions the instruction set of the 6502 is quite basic, having only simple 8-bit operations. Complex operations such as 16 or 32 bit arithmetic and memory transfers have to be performed by executing a sequence of simpler operations.

Standard Conventions

The 6502 processor expects addresses to be stored in 'little endian' order, with the least significant byte first and the most significant byte second. If the value stored was just a number (e.g. game score, etc.) then we could write code to store and manipulate it in 'big endian' order if we wished, however the algorithms presented here always use 'little endian' order so that they may be applied either to simple numeric values or addresses without modification.

> *The terms 'big endian' and 'little endian' come from Gulliver's Travels. The people of Lilliput and Blefuscu have been fighting a war over which end of a boiled egg one should crack to eat it. In computer terms it refers to whether the most or least significant portion of a binary number is stored in the lower memory address.*

To be safe the algorithms usually start by setting processor flags and registers to safe initial values. If you need to squeeze a few extra bytes or cycles out of the routine you might be able to remove some of these initializations depending on the preceding instructions.

Simple Memory Operations

Probably the most fundamental memory operation is clearing an area of memory to an initial value, such as zero. As the 6502 cannot directly move values to memory clearing even a small region of memory requires the use of a register. Any of A, X or Y could be used to hold the initial value, but in practice A is normally used because it can be quickly saved and restored (with PHA and PLA) leaving X and Y free for application use.

```
; Clearing 16 bits of memory
LDA #0       ;Load constant zero into A
STA MEM+0    ;Then clear the least significant byte
STA MEM+1    ;... followed by the most significant
```

Moving a small quantity of data requires a register to act as a temporary container during the transfer. Again, any of A, X, or Y may be used, but as before using A as the temporary register is often the most practical.

```
; Moving 16 bits of memory
LDA SRC+0     ;Move the least significant byte
STA DST+0
LDA SRC+1     ;Then the most significant
STA DST+1
```

Another basic operation is setting a 16-bit word to an initial constant value. The easiest way to do this is to load the low and high portions into A one at a time and store them.

Logical Operations

The simplest forms of operation on binary values are the logical AND, logical OR and exclusive OR illustrated by the following truth tables.

Logical AND (AND)

	0	**1**
0	0	0
1	0	1

Logical OR (ORA)

	0	**1**
0	0	1
1	1	1

Exclusive OR (EOR)

	0	**1**
0	0	1
1	1	0

These results can be summarized in English as:

- The result of a logical AND is true (1) if and only if both inputs are true, otherwise it is false (0).
- The result of a logical OR is true (1) if either of the inputs its true, otherwise it is false (0).
- The result of an exclusive OR is true (1) if and only if one input is true and the other is false, otherwise it is false (0).

The tables show result of applying these operations on two one-bit values but as the 6502 comprises of eight-bit registers and memory each instruction will operate on two eight-bit values simultaneously as shown below.

	Logical AND (AND)	**Logical OR (ORA)**	**Exclusive OR (EOR)**
Value 1	0 0 1 1 0 0 1 1	0 0 1 1 0 0 1 1	0 0 1 1 0 0 1 1
Value 2	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
Result	0 0 0 1 0 0 0 1	0 1 1 1 0 1 1 1	0 1 1 0 0 1 1 0

It is important to understand the properties and practical applications of each of these operations as they are extensively used in other algorithms.

- Logical AND operates as a filter and is often used to select a subset of bits from a value (e.g. the status flags from a peripheral control chip).
- Logical OR allows bits to be inserted into an existing value (e.g. to set control flags in a peripheral control chip).
- Exclusive OR allows selected bits to be set or inverted.

In the 6502 these operations are implemented by the AND, ORA and EOR instructions. One of the values to be operated on will be the current contents of the accumulator, the other is in memory either as an immediate value or at a specified location. The result of the operation is placed in the accumulator and the zero and negative flags are set accordingly.

```
; Example logical operations
AND #$0F        ;Filter out all but the least 4 bits
ORA BITS,X      ;Insert some bits from a table
EOR (DATA),Y    ;EOR against some data
```

A very common use of the EOR instruction is to calculate the 'complement' (or logical NOT) of a value. This involves inverting every bit in the value and is most easily calculated by exclusively ORing against an all ones value.

```
; Calculate the complement
EOR #$FF
```

Shifts & Rotates

The shift and rotate instructions allow the bits within either the accumulator or a memory location to be moved by one place either up (left) or down (right). When the bits are moved a new value will be needed to fill the vacant position created at one end of the value, and similarly the bit displaced at the opposite end will need to be caught and stored.

Both shifts and rotates catch the displaced bit in the carry flag but they differ in how they fill the vacant position; shifts will always fill the vacant bit with a zero whilst a rotate will fill it with the value of the carry flag as it was at the start of the instruction.

For example, the following diagram shows the result of applying an 'Arithmetic Shift Left' (ASL) to the value $4D to give $9A.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |
                /   /   /   /   /   /   /   /
               /   /   /   /   /   /   /   /    0
              /   /   /   /   /   /   /   /   /
             /   |   |   |   |   |   |   |   |
            /    v   v   v   v   v   v   v   v
               +---+---+---+---+---+---+---+---+
Result:   C=0  | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
               +---+---+---+---+---+---+---+---+
```

Whist the following shows the result of applying a 'Rotate Left' (ROL) to the same value, but assuming that the carry contained the value one.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | C=1
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |   /
                /   /   /   /   /   /   /   /   /
               /   /   /   /   /   /   /   /   /
              /   /   /   /   /   /   /   /   /
             /   |   |   |   |   |   |   |   |
            /    v   v   v   v   v   v   v   v
               +---+---+---+---+---+---+---+---+
Result:   C=0  | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
               +---+---+---+---+---+---+---+---+
```

Shifting the bits within a value (and introducing a zero as the least significant bit) has the effect of multiplying its value by two. In order to apply this multiplication to a value larger than a single byte we use ASL to shift the first byte and then ROL all the subsequent bytes as necessary using the carry flag to temporarily hold the displaced bits as they are moved from one byte to the next.

```
; Shift a 16bit value by one place left (= multiply by two)
ASL MEM+0       ;Shift the LSB
ROL MEM+1       ;Rotate the MSB
```

The behavior of the right shift as rotates follows the same pattern. For example, we can apply a 'Logical Shift Right' (LSR) to the value $4D to give $26.

```
               +---+---+---+---+---+---+---+---+
Initial:       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
                 |   |   |   |   |   |   |   |
                  \   \   \   \   \   \   \   \
              0    \   \   \   \   \   \   \   \
                \   \   \   \   \   \   \   \   \
                 |   |   |   |   |   |   |   |   \
                 v   v   v   v   v   v   v   v    \
               +---+---+---+---+---+---+---+---+
Result:        | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |  C=1
               +---+---+---+---+---+---+---+---+
```

Or a 'Rotate Right' (ROR) of the same value, but assuming that the carry contained the value one to give $A6.

```
               +---+---+---+---+---+---+---+---+
Initial:   C=1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
               +---+---+---+---+---+---+---+---+
             \   |   |   |   |   |   |   |   |
              \   \   \   \   \   \   \   \   \
               \   \   \   \   \   \   \   \   \
                \   \   \   \   \   \   \   \   \
                 |   |   |   |   |   |   |   |   \
                 v   v   v   v   v   v   v   v    \
               +---+---+---+---+---+---+---+---+
Result:        | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |  C=1
               +---+---+---+---+---+---+---+---+
```

Not surprisingly if left shifts multiply a value by two then right shifts do an unsigned division by two. Again, if we are applying the division to a multi-byte value, we will typically use LSR on the first byte (the MSB this time) and ROR on all subsequent bytes.

```
; Shift a 16 bit value by one place right (= divide by two)
LSR MEM+1       ;Shift the MSB
ROR MEM+0       ;Rotate the LSB
```

There are a number of applications for shifts and rotates, not least the coding of generic multiply and divide algorithms which are discussed later.

As was pointed out earlier right shifting a value two divide it by two only works on unsigned values. This is because the LSR is will always place a zero in the most significant bit of the MSB. To make this algorithm work for all two complement coded values we need to ensure that value of this bit is copied back into itself to keep the value the same sign. We can use another shift to achieve this.

```
; Divide a signed 16 bit value by two
LDA MEM+1       ;Load the MSB
ASL A           ;Copy the sign bit into C
ROR MEM+1       ;And back into the MSB
ROR MEM+0       ;Rotate the LSB as normal
```

Addition & Subtraction

The 6502 processor provides 8-bit addition and subtraction instructions and a carry/borrow flag that is used to propagate the carry bit between operations.

To implement a 16-bit addition the programmer must code two pairs of additions; one for the least significant bytes and one for the most significant bytes. The carry flag must be cleared before the first addition to ensure that an additional increment isn't performed.

```
; 16 bit Binary Addition
CLC             ;Ensure carry is clear
LDA VLA+0       ;Add the two least significant bytes
ADC VLB+0
STA RES+0       ;... and store the result
LDA VLA+1       ;Add the two most significant bytes
ADC VLB+1       ;... and any propagated carry bit
STA RES+1       ;... and store the result
```

Subtraction follows the same pattern but the carry must be set before the first pair of bytes are subtracted to get the correct result.

```
; 16 bit Binary Subtraction
SEC             ;Ensure carry is set
LDA VLA+0       ;Subtract the two least significant bytes
SBC VLB+0
STA RES+0       ;... and store the result
LDA VLA+1       ;Subtract the two most significant bytes
SBC VLB+1       ;... and any propagated borrow bit
STA RES+1       ;... and store the result
```

Both the addition and subtraction algorithm can be extended to 32 bits by repeating the LDA/ADC/STA or LDA/SBC/STA pattern for two further bytes worth of data.

Negation

The traditional approach to negating a two's complement number is to reverse all the bits (by EORing with $FF) and add one as shown below.

```
; 8 bit Binary Negation
CLC        ;Ensure carry is clear
EOR #$FF   ;Invert all the bits
ADC #1     ;... and add one
```

This technique works well with a single byte already held in the accumulator but not with bigger numbers. With these it is easier just to subtract them from zero.

```
; 16 bit Binary Negation
SEC             ;Ensure carry is set
LDA #0          ;Load constant zero
SBC SRC+0       ;... subtract the least significant byte
STA DST+0       ;... and store the result
LDA #0          ;Load constant zero again
SBC SRC+1       ;... subtract the most significant byte
STA DST+1       ;... and store the result
```

Decimal Arithmetic

The behavior of the ADC and SBC instructions can be modified by setting or clearing the decimal mode flag in the processor status register. Normally decimal mode is disabled and ADC/SBC perform simple binary arithmetic (e.g. $99 + $01 => $9A Carry = 0), but if the flag is set with a SED instruction the processor will perform binary coded decimal arithmetic instead (e.g. $99 + $01 => $00 Carry = 1).

To make the 16-bit addition/subtraction code work in decimal mode simply include an SED at the start and a CLD at the end (to restore the processor to normal).

```
; 16 bit Binary Code Decimal Addition
SED             ;Set decimal mode flag
CLC             ;Ensure carry is clear
LDA VLA+0       ;Add the two least significant bytes
ADC VLB+0
STA RES+0       ;... and store the result
LDA VLA+1       ;Add the two most significant bytes
ADC VLB+1       ;... and any propagated carry bit
STA RES+1       ;... and store the result
CLD             ;Clear decimal mode
```

Another use for BCD is in the conversion of binary values to decimal ones. Some algorithms perform this conversion by counting the number of times that 10000's, 1000's, 100's, 10's and 1's can be subtracted from the binary value before it underflows, but I normally use a simple fixed loop that shifts the bits out of the binary value one at a time and adds it to an intermediate result that is being doubled (in BCD) on each iteration.

```
; Convert an 16 bit binary value into a 24bit BCD value
BIN2BCD:
        LDA #0          ;Clear the result area
        STA RES+0
        STA RES+1
        STA RES+2
        LDX #16         ;Setup the bit counter
        SED             ;Enter decimal mode
_LOOP:
        ASL VAL+0       ;Shift a bit out of the binary
        ROL VAL+1       ;... value
        LDA RES+0       ;And add it into the result, doubling
        ADC RES+0       ;... it at the same time
        STA RES+0
        LDA RES+1
        ADC RES+1
        STA RES+1
        LDA RES+2
        ADC RES+2
        STA RES+2
        DEX             ;More bits to process?
        BNE _LOOP
        CLD             ;Leave decimal mode
```

Increments & Decrements

Assembly programs frequently use memory-based counters that occasionally need incrementing or decrementing by one. One way to achieve this would be to load the LSB and MSB in turn and add or subtract one with the ADC/SBC instructions, but the 6502 has a more efficient way to do this using INC and DEC.

Incrementing is straight forward; we just increment the least significant byte until the result becomes zero. This indicates that the calculation has wrapped round (e.g. $FF + $01 => $00) and an increment to the most significant byte is needed.

```
; Increment a 16 bit value by one
  INC MEM+0       ;Increment the LSB
  BNE _DONE       ;If the result was not zero we're done
  INC MEM+1       ;Increment the MSB if LSB wrapped round
_DONE:
  EQU *
```

Decrementing is a little trickier because we need to know when the least significant byte is about to underflow from $00 to $FF. The answer is to test it first by loading it into the accumulator to set the processor flags.

```
; Decrement a 16 bit value by one
  LDA MEM+0  ;Test if the LSB is zero
  BNE _SKIP  ;If it isn't we can skip the next instruction
  DEC MEM+1  ;Decrement the MSB when the LSB will underflow
_SKIP:
  DEC MEM+0  ;Decrement the LSB
```

Complex Memory Transfers

Moving data from one place to another is a common operation. If the amount of data to moved is 256 bytes or less and the source and target locations of the data are fixed then a simple loop around an indexed LDA followed by an indexed STA is the most efficient. Note that whilst both the X and Y registers can be used in indexed addressing modes an asymmetry in the 6502's instruction means that X is the better register to use if one or both of the memory areas resides on zero page.

```
; Move 256 bytes or less in a forward direction
     LDX #0       ;Start with the first byte
_LOOP:
     LDA SRC,X    ;Move it
     STA DST,X
     INX          ;Then bump the index ...
     CPX #LEN     ;... until we reach the limit
     BNE _LOOP
```

The corresponding code moving the last byte first is as follows:

```
; Move 256 bytes or less in a reverse direction
     LDX #LEN     ;Start with the last byte
_LOOP:
     DEX          ;Bump the index
     LDA SRC,X    ;Move a byte
     STA DST,X
     CPX #0       ;... until all bytes have moved
     BNE _LOOP
```

If the amount is even smaller (128 bytes or less) then we can eliminate the comparison against the limit and use the settings of the flags after a DEX to determine if the loop has finished.

```
; Move 128 bytes or less in a reverse direction
     LDX #LEN-1   ;Start with the last byte
_LOOP:
     LDA SRC,X    ;Move it
     STA DST,X
     DEX          ;Then bump the index ...
     BPL _LOOP    ;... until all bytes have moved
```

To create a completely generic memory transfer we must change to using indirect indexed addressing to access memory and use all the registers. The following code shows a forward transferring algorithm which first moves complete pages of 256 bytes followed by any remaining fragments of smaller size.

```
_MOVFWD:
     LDY #0       ;Initialise the index
     LDX LEN+1    ;Load the page count
     BEQ _FRAG    ;... Do we only have a fragment?
_PAGE:
     LDA (SRC),Y  ;Move a byte in a page transfer
     STA (DST),Y
     INY          ;And repeat for the rest of the
     BNE _PAGE    ;... page
     INC SRC+1    ;Then bump the src and dst addresses
     INC DST+1    ;... by a page
     DEX          ;And repeat while there are more
     BNE _PAGE    ;... pages to move
_FRAGCPY:
     LEN+0  ;Then while the index has not reached
     BEQ _DONE    ;... the limit
     LDA (SRC),Y  ;Move a fragment byte
     STA (DST),Y
     INY          ;Bump the index and repeat
     BNE _FRAG\?
_DONE:
     EQU *        ;All done
```

Instruction Reference

ADC page 32	**AND** page 33	**ASL** page 34	**BCC** page 35	**BCS** page 36	**BEQ** page 37	**BIT** page 38
BMI page 39	**BNE** page 40	**BPL** page 41	**BRK** page 42	**BVC** page 43	**BVS** page 44	**CLC** page 45
CLD page 46	**CLI** page 47	**CLV** page 48	**CMP** page 49	**CPX** page 50	**CPY** page 51	**DEC** page 52
DEX page 53	**DEY** page 54	**EOR** page 55	**INC** page 56	**INX** page 57	**INY** page 58	**JMP** page 59
JSR page 60	**LDA** page 61	**LDX** page 62	**LDY** page 63	**LSR** page 64	**NOP** page 65	**ORA** page 66
PHA page 67	**PHP** page 68	**PLA** page 69	**PLP** page 70	**ROL** page 71	**ROR** page 72	**RTI** page 73
RTS page 74	**SBC** page 75	**SEC** page 76	**SED** page 77	**SEI** page 78	**STA** page 79	**STX** page 80
STY page 81	**TAX** page 82	**TAY** page 83	**TSX** page 84	**TXA** page 85	**TXS** page 86	**TYA** page 87

ADC - Add with Carry

A,Z,C,N = A+M+C

This instruction adds the contents of a memory location to the accumulator together with the carry bit. If overflow occurs the carry bit is set, this enables multiple byte addition to be performed.

Processor Status after use:

C	Carry Flag	Set if overflow in bit 7
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Set if sign bit is incorrect
N	Negative Flag	Set if bit 7 set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$69	2	2
Zero Page	$65	2	3
Zero Page,X	$75	2	4
Absolute	$6D	3	4
Absolute,X	$7D	3	4 (+1 if page crossed)
Absolute,Y	$79	3	4 (+1 if page crossed)
(Indirect,X)	$61	2	6
(Indirect),Y	$71	2	5 (+1 if page crossed)

See also: SBC

AND - Logical AND

A,Z,N = A&M

A logical AND is performed, bit by bit, on the accumulator contents using the contents of a byte of memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$29	2	2
Zero Page	$25	2	3
Zero Page,X	$35	2	4
Absolute	$2D	3	4
Absolute,X	$3D	3	4 (+1 if page crossed)
Absolute,Y	$39	3	4 (+1 if page crossed)
(Indirect,X)	$21	2	6
(Indirect),Y	$31	2	5 (+1 if page crossed)

See also: EOR, ORA

ASL - Arithmetic Shift Left

A,Z,C,N = M*2 or M,Z,C,N = M*2

This operation shifts all the bits of the accumulator or memory contents one bit left. Bit 0 is set to 0 and bit 7 is placed in the carry flag. The effect of this operation is to multiply the memory contents by 2 (ignoring 2's complement considerations), setting the carry if the result will not fit in 8 bits.

Processor Status after use:

C	Carry Flag	Set to contents of old bit 7
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Accumulator	$0A	1	2
Zero Page	$06	2	5
Zero Page,X	$16	2	6
Absolute	$0E	3	6
Absolute,X	$1E	3	7

See also: LSR, ROL, ROR

BCC - Branch if Carry Clear

If the carry flag is clear then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$90	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BCS

BCS - Branch if Carry Set

If the carry flag is set then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$B0	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BCC

BEQ - Branch if Equal

If the zero flag is set then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$F0	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BNE

BIT - Bit Test

A & M, N = M7, V = M6

This instruction is used to test if one or more bits are set in a target memory location. The mask pattern in A is ANDed with the value in memory to set or clear the zero flag, but the result is not kept. Bits 7 and 6 of the value from memory are copied into the N and V flags.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if the result if the AND is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Set to bit 6 of the memory value
N	Negative Flag	Set to bit 7 of the memory value

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$24	2	3
Absolute	$2C	3	4

BMI - Branch if Minus

If the negative flag is set then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$30	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BPL

BNE - Branch if Not Equal

If the zero flag is clear then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$D0	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BEQ

BPL - Branch if Positive

If the negative flag is clear then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$10	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BMI

BRK - Force Interrupt

The BRK instruction forces the generation of an interrupt request. The program counter and processor status are pushed on the stack then the IRQ interrupt vector at $FFFE/F is loaded into the PC and the break flag in the status set to one.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Set to 1
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$00	1	7

The interpretation of a BRK depends on the operating system. On the BBC Microcomputer it is used by language ROMs to signal run time errors but it could be used for other purposes (e.g. calling operating system functions, etc.).

BVC - Branch if Overflow Clear

If the overflow flag is clear then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	Opcode	Bytes	Cycles
Relative	$50	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BVS

BVS - Branch if Overflow Set

If the overflow flag is set then add the relative displacement to the program counter to cause a branch to a new location.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Relative	$70	2	2 (+1 if branch succeeds +2 if to a new page)

See also: BVC

CLC - Clear Carry Flag

C = 0

Set the carry flag to zero.

C	Carry Flag	Set to 0
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$18	1	2

See also: SEC

CLD - Clear Decimal Mode

D = 0

Sets the decimal mode flag to zero.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Set to 0
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	Opcode	Bytes	Cycles
Implied	$D8	1	2

NB:
The state of the decimal flag is uncertain when the CPU is powered up and it is not reset when an interrupt is generated. In both cases you should include an explicit CLD to ensure that the flag is cleared before performing addition or subtraction.

See also: SED

CLI - Clear Interrupt Disable

I = 0

Clears the interrupt disable flag allowing normal interrupt requests to be serviced.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Set to 0
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$58	1	2

See also: SEI

CLV - Clear Overflow Flag

V = 0

Clears the overflow flag.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Set to 0
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$B8	1	2

CMP - Compare

Z,C,N = A-M

This instruction compares the contents of the accumulator with another memory held value and sets the zero and carry flags as appropriate.

Processor Status after use:

C	Carry Flag	Set if A >= M
Z	Zero Flag	Set if A = M
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$C9	2	2
Zero Page	$C5	2	3
Zero Page,X	$D5	2	4
Absolute	$CD	3	4
Absolute,X	$DD	3	4 (+1 if page crossed)
Absolute,Y	$D9	3	4 (+1 if page crossed)
(Indirect,X)	$C1	2	6
(Indirect),Y	$D1	2	5 (+1 if page crossed)

See also: CPX, CPY

CPX - Compare X Register

Z,C,N = X-M

This instruction compares the contents of the X register with another memory held value and sets the zero and carry flags as appropriate.

Processor Status after use:

C	Carry Flag	Set if X >= M
Z	Zero Flag	Set if X = M
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	Opcode	Bytes	Cycles
Immediate	$E0	2	2
Zero Page	$E4	2	3
Absolute	$EC	3	4

See also: CMP, CPY

CPY - Compare Y Register

Z,C,N = Y-M

This instruction compares the contents of the Y register with another memory held value and sets the zero and carry flags as appropriate.

Processor Status after use:

C	Carry Flag	Set if Y >= M
Z	Zero Flag	Set if Y = M
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$C0	2	2
Zero Page	$C4	2	3
Absolute	$CC	3	4

See also: CMP, CPX

DEC - Decrement Memory

M,Z,N = M-1

Subtracts one from the value held at a specified memory location setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if result is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$C6	2	5
Zero Page,X	$D6	2	6
Absolute	$CE	3	6
Absolute,X	$DE	3	7

See also: DEX, DEY

DEX - Decrement X Register

X,Z,N = X-1

Subtracts one from the X register setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if X is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of X is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$CA	1	2

See also: DEC, DEY

DEY - Decrement Y Register

Y,Z,N = Y-1

Subtracts one from the Y register setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if Y is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of Y is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$88	1	2

See also: DEC, DEX

EOR - Exclusive OR

A,Z,N = A^M

An exclusive OR is performed, bit by bit, on the accumulator contents using the contents of a byte of memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$49	2	2
Zero Page	$45	2	3
Zero Page,X	$55	2	4
Absolute	$4D	3	4
Absolute,X	$5D	3	4 (+1 if page crossed)
Absolute,Y	$59	3	4 (+1 if page crossed)
(Indirect,X)	$41	2	6
(Indirect),Y	$51	2	5 (+1 if page crossed)

See also: AND, ORA

INC - Increment Memory

M,Z,N = M+1

Adds one to the value held at a specified memory location setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if result is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$E6	2	5
Zero Page,X	$F6	2	6
Absolute	$EE	3	6
Absolute,X	$FE	3	7

See also: INX, INY

INX - Increment X Register

X,Z,N = X+1

Adds one to the X register setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if X is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of X is set

Addressing Mode	Opcode	Bytes	Cycles
Implied	$E8	1	2

See also: INC, INY

INY - Increment Y Register

Y,Z,N = Y+1

Adds one to the Y register setting the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if Y is zero
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of Y is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$C8	1	2

See also: INC, INX

JMP - Jump

Sets the program counter to the address specified by the operand.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	Opcode	Bytes	Cycles
Absolute	$4C	3	3
Indirect	$6C	3	5

NB:
An original 6502 has does not correctly fetch the target address if the indirect vector falls on a page boundary (e.g. $xxFF where xx is any value from $00 to $FF). In this case fetches the LSB from $xxFF as expected but takes the MSB from $xx00. This is fixed in some later chips like the 65SC02 so for compatibility always ensure the indirect vector is not at the end of the page.

JSR - Jump to Subroutine

The JSR instruction pushes the address (minus one) of the return point on to the stack and then sets the program counter to the target memory address.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Absolute	$20	3	6

See also: RTS

LDA - Load Accumulator

A,Z,N = M

Loads a byte of memory into the accumulator setting the zero and negative flags as appropriate.

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of A is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$A9	2	2
Zero Page	$A5	2	3
Zero Page,X	$B5	2	4
Absolute	$AD	3	4
Absolute,X	$BD	3	4 (+1 if page crossed)
Absolute,Y	$B9	3	4 (+1 if page crossed)
(Indirect,X)	$A1	2	6
(Indirect),Y	$B1	2	5 (+1 if page crossed)

See also: LDX, LDY

LDX - Load X Register

X,Z,N = M

Loads a byte of memory into the X register setting the zero and negative flags as appropriate.

C	Carry Flag	Not affected
Z	Zero Flag	Set if X = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of X is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$A2	2	2
Zero Page	$A6	2	3
Zero Page,Y	$B6	2	4
Absolute	$AE	3	4
Absolute,Y	$BE	3	4 (+1 if page crossed)

See also: LDA, LDY

LDY - Load Y Register

Y,Z,N = M

Loads a byte of memory into the Y register setting the zero and negative flags as appropriate.

C	Carry Flag	Not affected
Z	Zero Flag	Set if Y = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of Y is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$A0	2	2
Zero Page	$A4	2	3
Zero Page,X	$B4	2	4
Absolute	$AC	3	4
Absolute,X	$BC	3	4 (+1 if page crossed)

See also: LDA, LDX

LSR - Logical Shift Right

A,C,Z,N = A/2 or M,C,Z,N = M/2

Each of the bits in A or M is shift one place to the right. The bit that was in bit 0 is shifted into the carry flag. Bit 7 is set to zero.

Processor Status after use:

C	Carry Flag	Set to contents of old bit 0
Z	Zero Flag	Set if result = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Accumulator	$4A	1	2
Zero Page	$46	2	5
Zero Page,X	$56	2	6
Absolute	$4E	3	6
Absolute,X	$5E	3	7

See also: ASL, ROL, ROR

NOP - No Operation

The NOP instruction causes no changes to the processor other than the normal incrementing of the program counter to the next instruction.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	Opcode	Bytes	Cycles
Implied	$EA	1	2

ORA - Logical Inclusive OR

A,Z,N = A|M

An inclusive OR is performed, bit by bit, on the accumulator contents using the contents of a byte of memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$09	2	2
Zero Page	$05	2	3
Zero Page,X	$15	2	4
Absolute	$0D	3	4
Absolute,X	$1D	3	4 (+1 if page crossed)
Absolute,Y	$19	3	4 (+1 if page crossed)
(Indirect,X)	$01	2	6
(Indirect),Y	$11	2	5 (+1 if page crossed)

See also: AND, EOR

PHA - Push Accumulator

Pushes a copy of the accumulator on to the stack.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$48	1	3

See also: PLA

PHP - Push Processor Status

Pushes a copy of the status flags on to the stack.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$08	1	3

See also: PLP

PLA - Pull Accumulator

Pulls an 8 bit value from the stack and into the accumulator. The zero and negative flags are set as appropriate.

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of A is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$68	1	4

See also: PHA

PLP - Pull Processor Status

Pulls an 8 bit value from the stack and into the processor flags. The flags will take on new states as determined by the value pulled.

Processor Status after use:

C	Carry Flag	Set from stack
Z	Zero Flag	Set from stack
I	Interrupt Disable	Set from stack
D	Decimal Mode Flag	Set from stack
B	Break Command	Set from stack
V	Overflow Flag	Set from stack
N	Negative Flag	Set from stack

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$28	1	4

See also: PHP

ROL - Rotate Left

Move each of the bits in either A or M one place to the left. Bit 0 is filled with the current value of the carry flag whilst the old bit 7 becomes the new carry flag value.

Processor Status after use:

C	Carry Flag	Set to contents of old bit 7
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Accumulator	$2A	1	2
Zero Page	$26	2	5
Zero Page,X	$36	2	6
Absolute	$2E	3	6
Absolute,X	$3E	3	7

See also: ASL, LSR, ROR

ROR - Rotate Right

Move each of the bits in either A or M one place to the right. Bit 7 is filled with the current value of the carry flag whilst the old bit 0 becomes the new carry flag value.

Processor Status after use:

C	Carry Flag	Set to contents of old bit 0
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of the result is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Accumulator	$6A	1	2
Zero Page	$66	2	5
Zero Page,X	$76	2	6
Absolute	$6E	3	6
Absolute,X	$7E	3	7

See also ASL, LSR, ROL

RTI - Return from Interrupt

The RTI instruction is used at the end of an interrupt processing routine. It pulls the processor flags from the stack followed by the program counter.

Processor Status after use:

C	Carry Flag	Set from stack
Z	Zero Flag	Set from stack
I	Interrupt Disable	Set from stack
D	Decimal Mode Flag	Set from stack
B	Break Command	Set from stack
V	Overflow Flag	Set from stack
N	Negative Flag	Set from stack

Addressing Mode	Opcode	Bytes	Cycles
Implied	$40	1	6

RTS - Return from Subroutine

The RTS instruction is used at the end of a subroutine to return to the calling routine. It pulls the program counter (minus one) from the stack.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$60	1	6

See also: JSR

SBC - Subtract with Carry

A,Z,C,N = A-M-(1-C)

This instruction subtracts the contents of a memory location to the accumulator together with the not of the carry bit. If overflow occurs the carry bit is clear, this enables multiple byte subtraction to be performed.

Processor Status after use:

C	Carry Flag	Clear if overflow in bit 7
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Set if sign bit is incorrect
N	Negative Flag	Set if bit 7 set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Immediate	$E9	2	2
Zero Page	$E5	2	3
Zero Page,X	$F5	2	4
Absolute	$ED	3	4
Absolute,X	$FD	3	4 (+1 if page crossed)
Absolute,Y	$F9	3	4 (+1 if page crossed)
(Indirect,X)	$E1	2	6
(Indirect),Y	$F1	2	5 (+1 if page crossed)

See also: ADC

SEC - Set Carry Flag

C = 1

Set the carry flag to one.

C	Carry Flag	Set to 1
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$38	1	2

See also: CLC

SED - Set Decimal Flag

D = 1

Set the decimal mode flag to one.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Set to 1
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$F8	1	2

See also: CLD

SEI - Set Interrupt Disable

I = 1

Set the interrupt disable flag to one.

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Set to 1
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$78	1	2

See also: CLI

STA - Store Accumulator

M = A

Stores the contents of the accumulator into memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$85	2	3
Zero Page,X	$95	2	4
Absolute	$8D	3	4
Absolute,X	$9D	3	5
Absolute,Y	$99	3	5
(Indirect,X)	$81	2	6
(Indirect),Y	$91	2	6

See also: STX, STY

STX - Store X Register

M = X

Stores the contents of the X register into memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$86	2	3
Zero Page,Y	$96	2	4
Absolute	$8E	3	4

See also: STA, STY

STY - Store Y Register

M = Y

Stores the contents of the Y register into memory.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Zero Page	$84	2	3
Zero Page,X	$94	2	4
Absolute	$8C	3	4

See also: STA, STX

TAX - Transfer Accumulator to X

X = A

Copies the current contents of the accumulator into the X register and sets the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if X = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of X is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$AA	1	2

See also: TXA

TAY - Transfer Accumulator to Y

Y = A

Copies the current contents of the accumulator into the Y register and sets the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if Y = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of Y is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$A8	1	2

See also: TYA

TSX - Transfer Stack Pointer to X

X = S

Copies the current contents of the stack register into the X register and sets the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if X = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of X is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$BA	1	2

See also: TXS

TXA - Transfer X to Accumulator

A = X

Copies the current contents of the X register into the accumulator and sets the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of A is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$8A	1	2

See also: TAX

TXS - Transfer X to Stack Pointer

S = X

Copies the current contents of the X register into the stack register.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Not affected
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Not affected

Addressing Mode	Opcode	Bytes	Cycles
Implied	$9A	1	2

See also: TSX

TYA - Transfer Y to Accumulator

A = Y

Copies the current contents of the Y register into the accumulator and sets the zero and negative flags as appropriate.

Processor Status after use:

C	Carry Flag	Not affected
Z	Zero Flag	Set if A = 0
I	Interrupt Disable	Not affected
D	Decimal Mode Flag	Not affected
B	Break Command	Not affected
V	Overflow Flag	Not affected
N	Negative Flag	Set if bit 7 of A is set

Addressing Mode	**Opcode**	**Bytes**	**Cycles**
Implied	$98	1	2

See also: TAY