6502 Instruction Set

Here are the 56 mnemonics, the 56 instructions you can give the 6502 (or 6510) chip. Each of them is described in several ways: what it does, what major uses it has in ML programming, what addressing modes it can use, what flags it affects, its opcode (hex/decimal), and the number of bytes it uses up.

ADC

What it does: Adds byte in memory to the byte in the Accumulator, plus the carry flag if set. Sets the carry flag if result exceeds 255. The result is left in the Accumulator.

Major uses: Adds two numbers together. If the carry flag is set prior to an ADC, the resulting number will be *one* greater than the total of the two numbers being added (the carry is added to the result). Thus, one always clears the carry (CLC) before beginning any addition operation. Following an ADC, a set (up) carry flag indicates that the result exceeded one byte's capacity (was greater than 255), so you can chain-add bytes by subsequent ADCs without any further CLCs (see "Multi-Byte Addition" in Appendix D).

Other flags affected by addition include the V (overflow) flag. This flag is rarely of any interest to the programmer. It merely indicates that a result became larger than could be held within bits 0–6. In other words, the result "overflowed" into bit 7, the highest bit in a byte. Of greater importance is the fact that the Z is set if the result of an addition is zero. Also the N flag is set if bit 7 is set. This N flag is called the "negative" flag because you can manipulate bytes thinking of the seventh bit as a sign (+ or −) to accomplish "signed arithmetic" if you want to. In this mode, each byte can hold a maximum value of 127 (since the seventh bit is used to reveal the number's sign). The B branching instruction's Relative addressing mode uses this kind of arithmetic.

ADC can be used following an SED which puts the 6502 into "decimal mode." Here's an example. Note that the number 75 is *decimal* after you SED:

```
SED
CLC
LDA #75
ADC #$05      (this will result in 80)
CLD           (always get rid of decimal mode as soon as you've
               finished)
```

Attractive as it sounds, the decimal mode isn't of much real value to the programmer. LADS will let you work in decimal if you want to without requiring that you enter the 6502's mode. Just leave off the $ and LADS will handle the decimal numbers for you.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	ADC #15	$69/105	2
Zero Page	ADC 15	$65/101	2
Zero Page,X	ADC 15,X	$75/117	2
Absolute	ADC 1500	$6D/109	3
Absolute,X	ADC 1500,X	$7D/125	3
Absolute,Y	ADC 1500,Y	$79/121	3
Indirect,X	ADC (15,X)	$61/97	2
Indirect,Y	ADC (15),Y	$71/113	2

Affected flags: N Z C V

AND

What it does: Logical ANDs the byte in memory with the byte in the Accumulator. The result is left in the Accumulator. All bits in both bytes are compared, and if both bits are 1, the result is 1. If either or both bits are 0, the result is 0.

Major uses: Most of the time, AND is used to turn bits off. Let's say that you are pulling in numbers higher than 128 (10000000 and higher) and you want to "unshift" them and print them as lowercase letters. You can then put a zero into the seventh bit of your "mask" and then AND the mask with the number being unshifted:

```
LDA ?         (test number)
AND #$7F      (01111111)
```

(If *either* bit is 0, the result will be 0. So the seventh bit of the test number is turned off here and all the other bits in the test number are unaffected.)

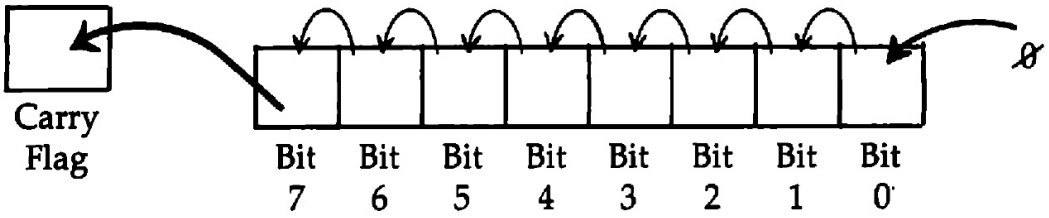
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	AND #15	$29/41	2
Zero Page	AND 15	$25/37	2
Zero Page,X	AND 15/X	$35/53	2
Absolute	AND 1500	$2D/45	3
Absolute,X	AND 1500,X	$3D/61	3
Absolute,Y	AND 1500,Y	$39/57	3
Indirect,X	AND (15,X)	$21/33	2
Indirect,Y	AND (15),Y	$31/49	2

Affected flags: N Z

ASL

What it does: Shifts the bits in a byte to the left by 1. This byte can be in the Accumulator or in memory, depending on the addressing mode. The shift moves the seventh bit into the carry flag and shoves a 0 into the zeroth bit.



Major uses: Allows you to multiply a number by 2. Numbers bigger than 255 can be manipulated using ASL with ROL (see "Multiplication" in Appendix D).

A secondary use is to move the lower four bits in a byte (a four-bit unit is often called a *nybble*) into the higher four bits. The lower bits are replaced by zeros, since ASL stuffs zeros into the zeroth bit of a byte. You move the lower to the higher nybble of a byte by: ASL ASL ASL ASL.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ASL	$0A/10	1
Zero Page	ASL 15	$06/6	2
Zero Page,X	ASL 15,X	$16/22	2
Absolute	ASL 1500	$0E/14	3
Absolute,X	ASL 1500,X	$1E/30	3

Affected flags: N Z C

BCC

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the carry flag is clear. In effect, it branches if the second item is lower than the first, as in: LDA #150: CMP #149 or LDA #22: SBC #15. These actions would clear the carry and, triggering BCC, a branch would take place.

Major uses: For testing the results of CMP or ADC or other operations which affect the carry flag. IF-THEN or ON-GOTO type structures in ML can involve the BCC test. It is similar to BASIC's > instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BCC addr.	$90/144	2

Affected flags: none of them.

BCS

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the carry flag is set. In effect, it branches if the second item is higher than the first, as in: LDA #150: CMP #249 or LDA #22: SBC #85. These actions would set the carry and, triggering BCS, a branch would take place.

Major uses: For testing the results of LDA or ADC or other operations which affect the carry flag. IF-THEN or ON-

GOTO type structures in ML can involve the BCC test. It is similar to BASIC's < instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BCS addr.	$B0/176	2

Affected flags: none of them.

BEQ

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the zero flag (Z) is set. In other words, it branches if an action on two bytes results in a 0, as in: LDA #150: CMP #150 or LDA #22: SBC #22. These actions would set the zero flag, so the branch would take place.

Major uses: For testing the results of LDA or ADC or other operations which affect the carry flag. IF-THEN or ON-GOTO type structures in ML can involve the BEQ test. It is similar to BASIC's = instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BEQ addr.	$F0/240	2

Affected flags: none of them.

BIT

What it does: Tests the bits in the byte in memory against the bits in the byte held in the Accumulator. The bytes (memory and Accumulator) are unaffected. BIT merely sets flags. The Z flag is set as if an Accumulator AND memory had been performed. The V flag and the N flag receive *copies* of the sixth and seventh bits of the tested number.

Major uses: Although BIT has the advantage of not having any effect on the tested numbers, it is infrequently used because you cannot employ the Immediate addressing mode with it. Other tests (CMP and AND, for example) can be used instead.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	BIT 15	$24/36	2
Absolute	BIT 1500	$2C/44	3

Affected flags: N Z V

BMI

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the negative (N) flag is set. In effect, it branches if the seventh bit has been set by the most recent event: LDA #150 or LDA #128 would set the seventh bit. These actions would set the N flag, signifying that a *minus number* is present if you are using signed arithmetic or that there is a *shifted character* (or a BASIC keyword) if you are thinking of a byte in terms of the ASCII code.

Major uses: Testing for BASIC keywords, shifted ASCII, or graphics symbols. Testing for + or − in signed arithmetic.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BMI addr.	$30/48	2

Affected flags: none of them.

BNE

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the zero flag is clear. In other words, it branches if the result of the most recent event is not zero, as in: LDA #150: SBC #120 or LDA #128: CMP #125. These actions would clear the Z flag, signifying that a result was not 0.

Major uses: The reverse of BEQ. BNE means Branch if Not Equal. Since a CMP subtracts one number from another to perform its comparison, a 0 result means that they are equal. Any other result will trigger a BNE (not equal). Like the other B branch instructions, it has uses in IF-THEN, ON-GOTO type structures and is used as a way to exit loops (for

example, BNE will branch back to the start of a loop until a 0 delimiter is encountered at the end of a text message). BNE is like BASIC's <> instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BNE addr.	$D0/208	2

Affected flags: none of them.

BPL

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the N flag is clear. In effect, it branches if the seventh bit is clear in the most recent event, as in: LDA #12 or LDA #127. These actions would clear the N flag, signifying that a *plus number* (or zero) is present in signed arithmetic mode.

Major uses: For testing the results of LDA or ADC or other operations which affect the negative (N) flag. IF-THEN or ON-GOTO type structures in ML can involve the BCC test. It is the opposite of the BMI instruction. BPL can be used for tests of "unshifted" ASCII characters and other bytes which have the seventh bit off and so are lower than 128 (0XXXXXXX).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BPL addr.	$10/16	2

Affected flags: none of them.

BRK

What it does: Causes a forced interrupt. This interrupt cannot be masked (prevented) by setting the I (interrupt) flag within the Status Register. If there is a Break Interrupt Vector (a vector is like a pointer) in the computer, it may point to a resident monitor if the computer has one. The PC and the Sta-

tus Register are saved on the stack. The PC points to the location of the BRK + 2.

Major uses: Debugging an ML program can often start with a sprinkling of BRKs into suspicious locations within the code. The ML is executed, a BRK stops execution and drops you into the monitor, you examine registers or tables or variables to see if they are as they should be at this point in the execution, and then you restart execution from the breakpoint. This instruction is essentially identical to the actions and uses of the STOP command in BASIC.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	BRK	$00/0	1

Affected flags: Break (B) flag is set.

BVC

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the V (overflow) flag is clear.

Major uses: None. In practice, few programmers use "signed" arithmetic where the seventh bit is devoted to indicating a positive or negative number (a set seventh bit means a negative number). The V flag has the job of notifying you when you've added, say 120 + 30, and have therefore set the seventh bit via an "overflow" (a result greater than 127). The result of your addition of two positive numbers should not be seen as a negative number, but the seventh bit *is* set. The V flag can be tested and will then reveal that your answer is still positive, but an overflow took place.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BVC addr.	$50/80	2

Affected flags: none of them.

BVS

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the V (overflow) flag is set).

Major uses: None. See BVC above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BVS addr.	$70/112	2

Affected flags: none of them.

CLC

What it does: Clears the carry flag. (Puts a 0 into it.)

Major uses: Always used before any addition (ADC). If there are to be a series of additions (multiple-byte addition), only the first ADC is preceded by CLC since the carry feature is necessary. There might be a carry, and the result will be incorrect if it is not taken into account.

The 6502 does not offer an addition instruction without the carry feature. Thus, you must always clear it before the first ADC so a carry won't be accidentally added.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLC	$18/24	1

Affected flags: Carry (C) flag is set to zero.

CLD

What it does: Clears the decimal mode flag. (Puts a 0 into it.)

Major uses: Commodore computers execute a CLD when first turned on as well as upon entry to monitor modes (PET/CBM models) and when the SYS command occurs. Apple and Atari, however, can arrive in an ML environment with the D flag in an indeterminant state. An attempt to execute

ML with this flag set would cause disaster—all mathematics would be performed in "decimal mode." It is therefore suggested that owners of Apple and Atari computers CLD during the early phase, the initialization phase, of their programs. Though this is an unlikely bug, it would be a difficult one to recognize should it occur.

For further detail about the 6502's decimal mode, see SED below.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLD	$D8/216	1

Affected flags: Decimal (D) flag is set to zero.

CLI

What it does: Clears the interrupt-disable flag. All interrupts will therefore be serviced (including maskable ones).

Major uses: To restore normal interrupt routine processing following a temporary suspension of interrupts for the purpose of redirecting the interrupt vector. For more detail, see SEI below.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLI	$58/88	1

Affected flags: Interrupt (I) flag is set to zero.

CLV

What it does: Clears the overflow flag. (Puts a 0 into it.)

Major uses: None. (See BVC above.)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLV	$B8/184	1

Affected flags: Overflow (V) flag is set to zero.

CMP

What it does: Compares the byte in memory to the byte in the Accumulator. Three flags are affected, but the bytes in memory and in the Accumulator are undisturbed. A CMP is actually a subtraction of the byte in memory from the byte in the Accumulator. Therefore, if you LDA #15:CMP #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CMP would have set the Z flag.

Major uses: This is an important instruction in ML. It is central to IF-THEN and ON-GOTO type structures. In combination with the B branching instructions like BEQ, CMP allows the 6502 chip to make decisions, to take alternative pathways depending on comparisons. CMP throws the N, Z, or C flags up or down. Then a B instruction can branch, depending on the condition of a flag.

Often, an action will affect flags by itself, and a CMP will not be necessary. For example, LDA #15 will put a 0 into the N flag (seventh bit not set) and will put a 0 into the Z flag (the result was not 0). LDA does not affect the C flag. In any event, you could LDA #15: BPL TARGET, and the branch would take effect. However, if you LDA $20 and need to know if the byte loaded is *precisely* $0D, you must CMP #$0D:BEQ TARGET. So, while CMP is sometimes not absolutely necessary, it will never hurt to include it prior to branching.

Another important branch decision is based on > or < situations. In this case, you use BCC and BCS to test the C (carry) flag. And you've got to keep in mind the *order* of the numbers being compared. The memory byte is compared to the byte sitting in the Accumulator. The structure is: memory *is less than or equal to* the Accumulator (BCC is triggered because the carry flag was cleared). Or memory *is more than* Accumulator (BCS is triggered because the carry flag was set). Here's an example. If you want to find out if the number in the Accumulator is less than $40, just CMP #$41:BCC

LESSTHAN (be sure to remember that the carry flag is cleared if a number is less than *or equal;* that's why we test for less than $40 by comparing with a $41):

```
LDA #75
CMP #$41; IS IT LESS THAN $40?
BCC LESSTHAN
```

One final comment about the useful BCC/BCS tests following CMP: It's easy to remember that BCC means *less than or equal* and BCS means *more than* if you notice that C is less than S in the alphabet.

The other flag affected by CMPs is the N flag. Its uses are limited since it merely reports the status of the seventh bit; BPL triggers if that bit is clear, BMI triggers if it's set. However, that seventh bit does show whether the number is greater than (or equal to) or less than 128, and you can apply this information to the ASCII code or to look for BASIC keywords or to search data bases (BPL and BMI are used by LADS' data base search routines in the Array subprogram). Nevertheless, since LDA and many other instructions affect the N flag, you can often directly BPL or BMI without any need to CMP first.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CMP #15	$C9/201	2
Zero Page	CMP 15	$C5/197	2
Zero Page,X	CMP 15,X	$D5/213	2
Absolute	CMP 1500	$CD/205	3
Absolute,X	CMP 1500,X	$DD/221	3
Absolute,Y	CMP 1500,Y	$D9/217	3
Indirect,X	CMP (15,X)	$C1/193	2
Indirect,Y	CMP (15),Y	$D1/209	2

Affected flags: N Z C

CPX

What it does: Compares the byte in memory to the byte in the X Register. Three flags are affected, but the bytes in memory and in the X Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in

the X Register. Therefore, if you LDA #15:CPX #15—the result (of the subtraction) will be zero and BEQ would be triggered since the CPX would have set the Z flag.

Major uses: X is generally used as an index, a counter within loops. Though the Y Register is often preferred as an index since it can serve for the very useful Indirect Y addressing mode (LDA (15),Y)—the X Register is nevertheless pressed into service when more than one index is necessary or when Y is busy with other tasks.

In any case, the flags, conditions, and purposes of CPX are quite similar to CMP (the equivalent comparison instruction for the Accumulator). For further information on the various possible comparisons (greater than, equal, less than, not equal), see CMP above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CPX #15	$E0/224	2
Zero Page	CPX 15	$E4/228	2
Absolute	CPX 1500	$EC/236	3

Affected flags: N Z C

CPY

What it does: Compares the byte in memory to the byte in the Y Register. Three flags are affected, but the bytes in memory and in the Y Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in the Y Register. Therefore, if you LDA #15: CPY #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CPY would have set the Z flag.

Major uses: Y is the most popular index, the most heavily used counter within loops since it can serve two purposes: It permits the very useful Indirect Y addressing mode (LDA (15),Y) and can simultaneously maintain a count of loop events.

See CMP above for a detailed discussion of the various branch comparisons which CPY can implement.

to reverse the current state of the sixth bit in a given byte: LDA BYTE:EOR #$40:STA BYTE. This will set bit 6 in BYTE if it was 0 (and clear it if it was 1). This selective bit toggling could be used to "shift" an unshifted ASCII character via EOR #$80 (1000000). Or if the character were shifted, EOR #$80 would make it lowercase. EOR toggles.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	EOR #15	$49/73	2
Zero Page	EOR 15	$45/69	2
Zero Page,X	EOR 15,X	$55/85	2
Absolute	EOR 1500	$4D/77	3
Absolute,X	EOR 1500,X	$5D/93	3
Absolute,Y	EOR 1500,Y	$59/89	3
Indirect,X	EOR (15,X)	$41/65	2
Indirect,Y	EOR (15),Y	$51/81	2

Affected flags: N Z

INC

What it does: Increases the value of a byte in memory by 1.

Major uses: Used exactly as DEC (see DEC above), except it counts up instead of down. For raising address pointers or supplementing the X and Y Registers as loop indexes.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	INC 15	$E6/230	2
Zero Page,X	INC 15,X	$F6/246	2
Absolute	INC 1500	$EE/238	3
Absolute,X	INC 1500,X	$FE/254	3

Affected flags: N Z

INX

What it does: Increases the X Register by 1.

the X Register. Therefore, if you LDA #15:CPX #15—the result (of the subtraction) will be zero and BEQ would be triggered since the CPX would have set the Z flag.

Major uses: X is generally used as an index, a counter within loops. Though the Y Register is often preferred as an index since it can serve for the very useful Indirect Y addressing mode (LDA (15),Y)—the X Register is nevertheless pressed into service when more than one index is necessary or when Y is busy with other tasks.

In any case, the flags, conditions, and purposes of CPX are quite similar to CMP (the equivalent comparison instruction for the Accumulator). For further information on the various possible comparisons (greater than, equal, less than, not equal), see CMP above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CPX #15	$E0/224	2
Zero Page	CPX 15	$E4/228	2
Absolute	CPX 1500	$EC/236	3

Affected flags: N Z C

CPY

What it does: Compares the byte in memory to the byte in the Y Register. Three flags are affected, but the bytes in memory and in the Y Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in the Y Register. Therefore, if you LDA #15: CPY #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CPY would have set the Z flag.

Major uses: Y is the most popular index, the most heavily used counter within loops since it can serve two purposes: It permits the very useful Indirect Y addressing mode (LDA (15),Y) and can simultaneously maintain a count of loop events.

See CMP above for a detailed discussion of the various branch comparisons which CPY can implement.

to reverse the current state of the sixth bit in a given byte: LDA BYTE:EOR #$40:STA BYTE. This will set bit 6 in BYTE if it was 0 (and clear it if it was 1). This selective bit toggling could be used to "shift" an unshifted ASCII character via EOR #$80 (1000000). Or if the character were shifted, EOR #$80 would make it lowercase. EOR toggles.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	EOR #15	$49/73	2
Zero Page	EOR 15	$45/69	2
Zero Page,X	EOR 15,X	$55/85	2
Absolute	EOR 1500	$4D/77	3
Absolute,X	EOR 1500,X	$5D/93	3
Absolute,Y	EOR 1500,Y	$59/89	3
Indirect,X	EOR (15,X)	$41/65	2
Indirect,Y	EOR (15),Y	$51/81	2

Affected flags: N Z

INC

What it does: Increases the value of a byte in memory by 1.

Major uses: Used exactly as DEC (see DEC above), except it counts up instead of down. For raising address pointers or supplementing the X and Y Registers as loop indexes.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	INC 15	$E6/230	2
Zero Page,X	INC 15,X	$F6/246	2
Absolute	INC 1500	$EE/238	3
Absolute,X	INC 1500,X	$FE/254	3

Affected flags: N Z

INX

What it does: Increases the X Register by 1.

Major uses: Used exactly as DEX (see DEX above), except it counts up instead of down. For loop indexing.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	INX	$E8/232	1

Affected flags: N Z

INY

What it does: Increases the Y Register by 1.

Major uses: Used exactly as DEY (see DEY above), except it counts up instead of down. For loop indexing and working with the Indirect Y addressing mode (LDA (15),Y).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	INY	$C8/200	1

Affected flags: N Z

JMP

What it does: Jumps to any location in memory.

Major uses: Branching long range. It is the equivalent of BASIC's GOTO instruction. The bytes in the Program Counter are replaced with the address (the argument) following the JMP instruction and, therefore, program execution continues from this new address.

Indirect jumping—JMP (1500)—is not recommended, although some programmers find it useful. It allows you to set up a table of jump targets and bounce off them indirectly. For example, if you had placed the numbers $00 $04 in addresses $88 and $89, a JMP ($0088) instruction would send the program to whatever ML routine was located in address $0400. Unfortunately, if you should locate one of your pointers on the edge of a *page* (for example, $00FF or $17FF), this Indirect JMP addressing mode reveals its great weakness. There is a bug which causes the jump to travel to the wrong place—JMP

($00FF) picks up the first byte of the pointer from $00FF, but the second byte of the pointer will be incorrectly taken from $0000. With JMP ($17FF), the second byte of the pointer would come from what's in address $1700.

Since there is this bug, and since there are no compelling reasons to set up JMP tables, you might want to forget you ever heard of Indirect jumping.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Absolute	JMP 1500	$4C/76	3
Indirect	JMP (1500)	$6C/108	3

Affected flags: none of them.

JSR

What it does: Jumps to a subroutine anywhere in memory. Saves the PC (Program Counter) address, plus three, of the JSR instruction by pushing it onto the stack. The next RTS in the program will then pull that address off the stack and return to the instruction following the JSR.

Major uses: As the direct equivalent of BASIC's GOSUB command, JSR is heavily used in ML programming to send control to a subroutine and then (via RTS) to return and pick up where you left off. The larger and more sophisticated a program becomes, the more often JSR will be invoked. In LADS, whenever something is printed to screen or printer, you'll often see a chain of JSRs performing necessary tasks: JSR PRNTCR: JSR PRNTSA:JSR PRNTSPACE:JSR PRNTNUM:JSR PRNTSPACE. This JSR chain prints a carriage return, the current assembly address, a space, a number, and another space.

Another thing you might notice in LADS and other ML programs is a PLA:PLA pair. Since JSR stuffs the correct return address onto the stack before leaving for a subroutine, you need to do something about that return address if you later decide *not to RTS* back to the position of the JSR in the program. This might be the case if you *usually* want to RTS, but in some particular cases, you don't. For those cases, you can take control of program flow by removing the return address

from the stack (PLA:PLA will clean off the two-byte address) and then performing a direct JMP to wherever you want to go.

If you JMP out of a subroutine without PLA:PLA, you could easily overflow the stack and crash the program.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Absolute	JSR 1500	$20/32	3

Affected flags: none of them.

LDA

What it does: Loads the Accumulator with a byte from memory. *Copy* might be a better word than *load,* since the byte in memory is unaffected by the transfer.

Major uses: The busiest place in the computer. Bytes coming in from disk, tape, or keyboard all flow through the Accumulator, as do bytes on their way to screen or peripherals. Also, because the Accumulator differs in some important ways from the X and Y Registers, the Accumulator is used by ML programmers in a different way from the other registers.

Since INY/DEY and INX/DEX make those registers useful as counters for loops (the Accumulator couldn't be conveniently employed as an index; there is no INA instruction), the Accumulator is the main temporary storage register for bytes during their manipulation in an ML program. ML programming, in fact, can be defined as essentially the rapid, organized maneuvering of single bytes in memory. And it is the Accumulator where these bytes often briefly rest before being sent elsewhere.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDA #15	$A9/169	2
Zero Page	LDA 15	$A5/165	2
Zero Page,X	LDA 15,X	$B5/181	2
Absolute	LDA 1500	$AD/173	3
Absolute,X	LDA 1500,X	$BD/189	3
Absolute,Y	LDA 1500,Y	$B9/185	3
Indirect,X	LDA (15,X)	$A1/161	2
Indirect,Y	LDA (15),Y	$B1/177	2

Affected flags: N Z

LDX

What it does: Loads the X Register with a byte from memory.

Major uses: The X Register can perform many of the tasks that the Accumulator performs, but it is generally used as an index for loops. In preparation for its role as an index, LDX puts a value into the register.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDX #15	$A2/162	2
Zero Page	LDX 15	$A6/166	2
Zero Page,Y	LDX 15,Y	$B6/182	2
Absolute	LDX 1500	$AE/174	3
Absolute,Y	LDX 1500,Y	$BE/190	3

Affected flags: N Z

LDY

What it does: Loads the Y Register with a byte from memory.

Major uses: The Y Register can perform many of the tasks that the Accumulator performs, but it is generally used as an index for loops. In preparation for its role as an index, LDY puts a value into the register.

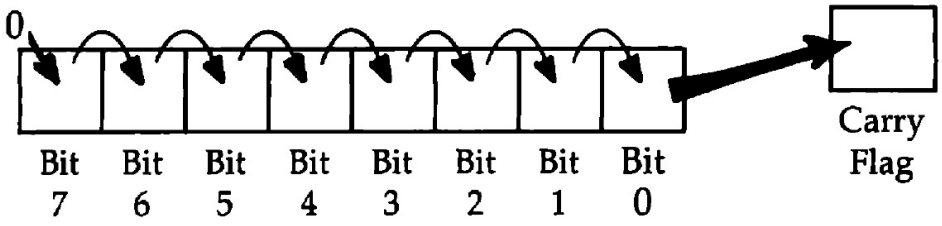
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDY #15	$A0/160	2
Zero Page	LDY 15	$A4/164	2
Zero Page,X	LDY 15,X	$B4/180	2
Absolute	LDY 1500	$AC/172	3
Absolute,X	LDY 1500,X	$BC/188	3

Affected flags: N Z

LSR

What it does: Shifts the bits in the Accumulator or in a byte in memory to the right, by one bit. A zero is stuffed into bit 7, and bit 0 is put into the carry flag.



Major uses: To divide a byte by 2. In combination with the ROR instruction, LSR can divide a two-byte or larger number (see Appendix D).

LSR:LSR:LSR:LSR will put the high four bits (the high nybble) into the low nybble (with the high nybble replaced by the zeros being stuffed into the seventh bit and then shifted to the right).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	LSR	$4A/74	2
Zero Page	LSR 15	$46/70	2
Zero Page,X	LSR 15,X	$56/86	2
Absolute	LSR 1500	$4E/78	3
Absolute,X	LSR 1500,X	$5E/94	3

Affected flags: N Z C

NOP

What it does: Nothing. No operation.

Major uses: Debugging. When setting breakpoints with BRK, you will often discover that a breakpoint, when examined, passes the test. That is, there is nothing wrong at that place in the program. So, to allow the program to execute to the next breakpoint, you cover the BRK with a NOP. Then, when you run the program, the computer will slide over the NOP with no effect on the program. Three NOPs could cover a JSR XXXX, and you could see the effect on the program when that particular JSR is eliminated.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	NOP	$EA/234	1

Affected flags: none of them.

ORA

What it does: Logically ORs a byte in memory with the byte in the Accumulator. The result is in the Accumulator. An OR results in a 1 if either the bit in memory or the bit in the Accumulator is 1.

Major uses: Like an AND mask which turns bits off, ORA masks can be used to turn bits on. For example, if you wanted to "shift" an ASCII character by setting the seventh bit, you could LDA CHARACTER:ORA #$80. The number $80 in binary is 10000000, so all the bits in CHARACTER which are ORed with zeros here will be left unchanged. (If a bit in CHARACTER is a 1, it stays a 1. If it is a zero, it stays 0.) But the 1 in the seventh bit of $80 will cause a 0 in the CHARACTER to turn into a 1. (If CHARACTER already has a 1 in its seventh bit, it will remain a 1.)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	ORA #15	$09/9	2
Zero Page	ORA 15	$05/5	2
Zero Page,X	ORA 15,X	$15/21	2
Absolute	ORA 1500	$0D/13	3
Absolute,X	ORA 1500,X	$1D/29	3
Absolute,Y	ORA 1500,Y	$19/25	3
Indirect,X	ORA (15,X)	$01/1	2
Indirect,Y	ORA (15),Y	$11/17	2

Affected flags: N Z

PHA

What it does: Pushes the Accumulator onto the stack.

Major uses: To temporarily *(very temporarily)* save the byte in the Accumulator. If you are within a particular subroutine and you need to save a value for a brief time, you can PHA it. But beware that you must PLA it back into the Accumulator *before any RTS* so that it won't misdirect the computer to the wrong RTS address. All RTS addresses are saved on the stack. Probably a safer way to temporarily save a value (a number) would be to STA TEMP or put it in some other temporary variable that you've set aside to hold things. Also, the values of A, X, and Y need to be temporarily saved, and the programmer will combine TYA and TXA with several PHAs to stuff all three registers onto the stack. But, again, matching PLAs must restore the stack as soon as possible and certainly prior to any RTS.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PHA	$48/72	1

Affected flags: none of them .

PHP

What it does: Pushes the "processor status" onto the top of the stack. This byte is the Status Register, the byte which holds all the flags: N Z C I D V.

Major uses: To temporarily *(very temporarily)* save the state of the flags. If you need to preserve the all current conditions for a minute (see description of PHA above), you may also want to preserve the Status Register as well. You must, however, restore the Status Register byte and clean up the stack by using a PLP before the next RTS.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PHP	$08/8	1

Affected flags: none of them.

PLA

What it does: Pulls the top byte off the stack and puts it into the Accumulator.

Major uses: To restore a number which was temporarily stored on top of the stack (with the PHA instruction). It is the opposite action of PHA (see above). Note that PLA does affect the N and Z flags. Each PHA must be matched by a corresponding PLA if the stack is to correctly maintain RTS addresses, which is the main purpose of the stack.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PLA	$68/104	1

Affected flags: N Z

PLP

What it does: Pulls the top byte off the stack and puts it into the Status Register (where the flags are). PLP is a mnemonic for PuLl Processor status.

Major uses: To restore the condition of the flags after the Status Register has been temporarily stored on top of the stack (with the PHP instruction). It is the opposite action of PHP (see above). PLP, of course, affects *all* the flags. Any PHP must be matched by a corresponding PLP if the stack is to correctly maintain RTS addresses, which is the main purpose of the stack.

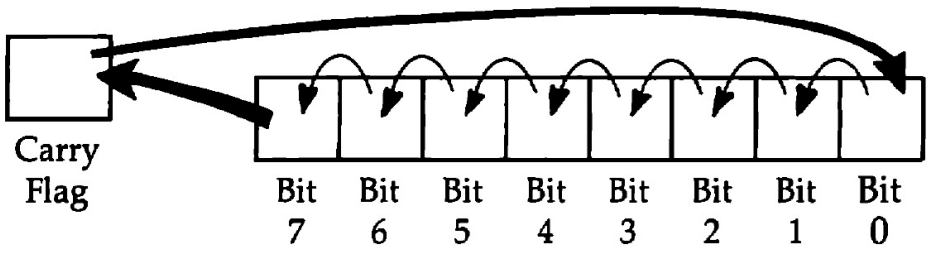
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PLP	$28/40	1

Affected flags: all of them.

ROL

What it does: Rotates the bits in the Accumulator or in a byte in memory to the left, by one bit. A rotate left (as opposed to an ASL, Arithmetic Shift Left) moves bit 7 to the carry, *moves the carry into bit 0,* and every other bit moves one position to its left. (ASL operates quite similarly, except it always puts a 0 into bit 0.)



Major uses: To multiply a byte by 2. ROL can be used with ASL to multiply multiple-byte numbers since ROL pulls any carry into bit 0. If an ASL resulted in a carry, it would be thus taken into account in the next higher byte in a multiple-byte number. (See Appendix D.)

Notice how the act of moving columns of binary numbers to the left has the effect of multiplying by 2:

0010 (the number 2 in binary)
0100 (the number 4)

This same effect can be observed with decimal numbers, except the columns represent powers of 10:

0010 (the number 10 in decimal)
0100 (the number 100)

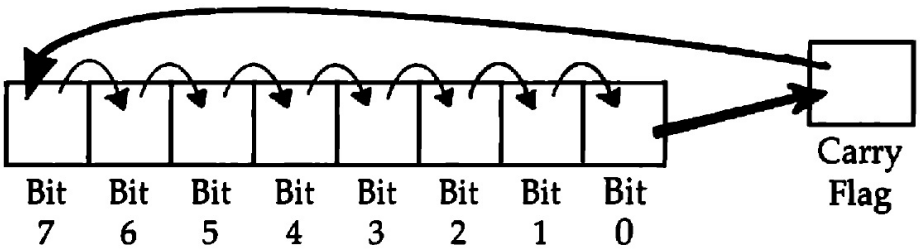
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ROL	$2A/42	1
Zero Page	ROL 15	$26/38	2
Zero Page,X	ROL 15,X	$36/54	2
Absolute	ROL 1500	$2E/46	3
Absolute,X	ROL 1500,X	$3E/62	3

Affected flags: N Z C

ROR

What it does: Rotates the bits in the Accumulator or in a byte in memory to the right, by one bit. A rotate right (as opposed to a LSR, Logical Shift Right) moves bit 0 into the carry, *moves the carry into bit 7*, and every other bit moves one position to its right. (LSR operates quite similarly, except it always puts a 0 into bit 7.)



Major uses: To divide a byte by 2. ROR can be used with LSR to divide multiple-byte numbers since ROR puts any carry into bit 7. If an LSR resulted in a carry, it would be thus taken into account in the next lower byte in a multiple-byte number. (See Appendix D.)

Notice how the act of moving columns of binary numbers to the right has the effect of dividing by 2:

1000 (the number 8 in binary)
0100 (the number 4)

This same effect can be observed with decimal numbers, except the columns represent powers of 10:

1000 (the number 1000 in decimal)
0100 (the number 100)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ROR	$6A/106	1
Zero Page	ROR 15	$66/102	2
Zero Page,X	ROR 15,X	$76/118	2
Absolute	ROR 1500	$6E/110	3
Absolute,X	ROR 1500,X	$7E/126	3

Affected flags: N Z C

RTI

What it does: Returns from an interrupt.

Major uses: None. You might want to add your own routines to your machine's normal interrupt routines (see SEI below), but you won't be *generating* actual interrupts of your own. Consequently, you cannot ReTurn from Interrupts you never create.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	RTI	$40/64	1

Affected flags: all of them (Status Register is retrieved from the stack).

RTS

What it does: Returns from a subroutine jump (caused by JSR).

Major uses: Automatically picks off the two top bytes on the stack and places them into the Program Counter. This reverses the actions taken by JSR (which put the Program Counter bytes onto the stack just before leaving for a subroutine). When RTS puts the return bytes into the Program

Counter, the next event in the computer's world will be the instruction following the JSR which stuffed the return address onto the stack in the first place.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	RTS	$60/96	1

Affected flags: none of them.

SBC

What it does: Subtracts a byte in memory *from* the byte in the Accumulator, and "borrows" if necessary. If a "borrow" takes place, the carry flag is cleared (set to 0). Thus, you always SEC (set the carry flag) before an SBC operation so you can tell if you need a "borrow." In other words, when an SBC operation clears the carry flag, it means that the byte in memory was *larger* than the byte in the Accumulator. And since memory is subtracted from the Accumulator in an SBC operation, if memory is the larger number, we must "borrow."

Major uses: Subtracts one number from another.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	SBC #15	$E9/233	2
Zero Page	SBC 15	$E5/229	2
Zero Page,X	SBC 15,X	$F5/245	2
Absolute	SBC 1500	$ED/237	3
Absolute,X	SBC 1500,X	$FD/253	3
Absolute,Y	SBC 1500,Y	$F9/249	3
Indirect,X	SBC (15,X)	$E1/225	2
Indirect,Y	SBC (15),Y	$F1/241	2

Affected flags: N Z C V

SEC

What it does: Sets the carry (C) flag (in the processor Status Register byte).

Major uses: This instruction is always used before any SBC operation to show if the result of the subtraction was negative (if the Accumulator contained a smaller number than the byte in memory being subtracted from it). See SBC above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SEC	$38/56	1

Affected flags: C

SED

What it does: Sets the decimal (D) flag (in the processor Status Register byte).

Major uses: Setting this flag puts the 6502 into decimal arithmetic mode. This mode can be easier to use when you are inputting or outputting decimal numbers (from the user of a program or to the screen). Simple addition and subtraction can be performed in decimal mode, but most programmers ignore this feature since more complicated math requires that you remain in the normal binary state of the 6502.

Note: Commodore computers automatically clear this mode when entering ML via SYS. However, Apple and Atari computers can enter ML in an indeterminant state. Since there is a possibility that the D flag might be set (causing havoc) on entry to an ML routine, it is sometimes suggested that owners of these two computers use the CLD instruction at the start of any ML program they write. Any ML programmer must CLD following any deliberate use of the decimal mode.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SED	$F8/248	1

Affected flags: D

SEI

What it does: Sets the interrupt disable flag (the I flag) in the processor status byte. When this flag is up, the 6502 will not acknowledge or act upon interrupt attempts (except a few nonmaskable interrupts which can take control in spite of this flag, like a reset of the entire computer). The operating systems of most computers will regularly interrupt the activities of the chip for necessary, high-priority tasks such as updating an internal clock, displaying things on the TV, receiving signals from the keyboard, etc. These interruptions of whatever the chip is doing normally occur 60 times every second. To find out what housekeeping routines your computer interrupts the chip to accomplish, look at the pointer in $FFFE/FFFF. It gives the starting address of the maskable interrupt routines.

Major uses: You can alter a RAM pointer so that it sends these interrupts to *your own ML routine*, and your routine then would conclude by pointing to the normal interrupt routines. In this way, you can add something you want (a click sound for each keystroke? the time of day on the screen?) to the normal actions of your operating system. The advantage of this method over normal SYSing is that your interrupt-driven routine is essentially transparent to whatever else you are doing (in whatever language). Your customization appears to have become part of the computer's ordinary habits.

However, if you try to alter the RAM pointer *while the other interrupts are active*, you will point away from the normal housekeeping routines in ROM, crashing the computer. This is where SEI comes in. You disable the interrupts while you LDA STA LDA STA the new pointer. Then CLI turns the interrupt back on and nothing is disturbed.

Interrupt processing is a whole subcategory of ML programming and has been widely discussed in magazine articles. Look there if you need more detail.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SEI	$78/120	1

Affected flags: I

STA

What it does: Stores the byte in the Accumulator into memory.

Major uses: Can serve many purposes and is among the most used instructions. Many other instructions leave their results in the Accumulator (ADC/SBC and logical operations like ORA), after which they are stored in memory with STA.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STA 15	$85/133	2
Zero Page,X	STA 15,X	$95/149	2
Absolute	STA 1500	$8D/141	3
Absolute,X	STA 1500,X	$9D/157	3
Absolute,Y	STA 1500,Y	$99/153	3
Indirect,X	STA (15,X)	$81/129	2
Indirect,Y	STA (15),Y	$91/145	2

Affected flags: none of them.

STX

What it does: Stores the byte in the X Register into memory.

Major uses: Copies the byte in X into a byte in memory.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STX 15	$86/134	2
Zero Page,Y	STX 15,Y	$96/150	2
Absolute	STX 1500	$8E/142	3

Affected flags: none of them.

STY

What it does: Stores the byte in the Y Register into memory.

Major uses: Copies the byte in Y into a byte in memory.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STY 15	$84/132	2
Zero Page,X	STY 15,X	$94/148	2
Absolute	STY 1500	$8C/140	3

Affected flags: none of them.

TAX

What it does: Transfers the byte in the Accumulator to the X Register.

Major uses: Sometimes you can copy the byte in the Accumulator into the X Register as a way of briefly storing the byte until it's needed again by the Accumulator. If X is currently unused, TAX is a convenient alternative to PHA (another temporary storage method).

However, since X is often employed as a loop counter, TAX is a relatively rarely used instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TAX	$AA/170	1

Affected flags: N Z

TAY

What it does: Transfers the byte in the Accumulator to the Y Register.

Major uses: Sometimes you can copy the byte in the Accumulator into the Y Register as a way of briefly storing the byte until it's needed again by the Accumulator. If Y is currently unused, TAY is a convenient alternative to PHA (another temporary storage method).

However, since Y is quite often employed as a loop counter, TAY is a relatively rarely used instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TAY	$A8/168	1

Affected flags: N Z

TSX

What it does: Transfers the Stack Pointer to the X Register.

Major uses: The Stack Pointer is a byte in the 6502 chip which points to where a new value (number) can be added to the stack. The Stack Pointer would be "raised" by two, for example, when you JSR and the two bytes of the Program Counter are pushed onto the stack. The next available space on the stack thus becomes two higher than it was previously. By contrast, an RTS will pull a two-byte return address off the stack, freeing up some space, and the Stack Pointer would then be "lowered" by two.

The Stack Pointer is always added to $0100 since the stack is located between addresses $0100 and $01FF.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TSX	$BA/186	1

Affected flags: N Z

TXA

What it does: Transfers the byte in the X Register to the Accumulator.

Major uses: There are times, after X has been used as a counter, when you'll want to compute something using the value of the counter. And you'll therefore need to transfer the byte in X to the Accumulator. For example, if you search the screen for character $75:

```
CHARACTER = $75:SCREEN =
$0400
LDX #0
LOOP LDA SCREEN,X:CMP
#CHARACTER:BEQ MORE:INX
BEQ NOTFOUND          ; (this prevents an endless loop
MORE TXA              ; (you now know the charac-
                      ter's location)
NOTFOUND BRK
```

In this example, we want to perform some action based on the location of the character. Perhaps we want to remember the location in a variable for later reference. This will require that we transfer the value of X to the Accumulator so it can be added to the SCREEN start address.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TXA	$8A/138	1

Affected flags: N Z

TXS

What it does: Transfers the byte in X Register into the Stack Pointer.

Major uses: Alters where, in the stack, the current "here's storage space" is pointed to. There are no common uses for this instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TXS	$9A/154	1

Affected flags: none of them.

TYA

What it does: Transfers the byte in the Y Register to the Accumulator.

Major uses: See TXA.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TYA	$98/152	1

Affected flags: N Z