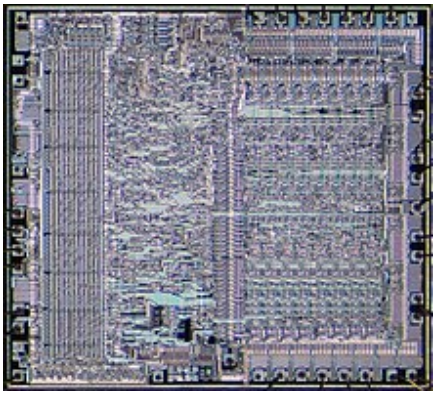
6502 Assembly

This book is a guide to the 6502 Assembly language. This book will teach the different memory addressing modes and instructions of the 8-bit 6502 processor.

You might want to learn 6502 assembly language programming if you want to do Atari 2600/8-bit family/5200/7800 Programming, Commodore PET/VIC/64/128 Programming, Acorn 8 Bit Programming, Apple I/II Programming, NES Programming or Super NES Programming.



A 6502 die

Syntax

Syntax will vary between assemblers - this book will use the following syntax throughout:

Numerical representations

Syntax	Base	Example
`%00001111`	Binary	`LDA #%0001`
`$FA`	Hexadecimal	`LDA #$0E`
`123`	Decimal	`LDA #100`

Registers

Registers

Register	Size (bits)	Purpose
Accumulator (A)	8	Used to perform calculations on data. Instructions can operate directly on the accumulator instead of spending CPU cycles to access memory
X register (X)	8	Used as an index in some addressing modes
Y register (Y)	8	Used as an index in some addressing modes
Program Counter (PC)	16	Points to the address of the next instruction to be executed
Stack Pointer (SP)	8	Stores the stack index into which the next stack element will be inserted. The address of this position is `$0100 + SP`. SP is initially set to `$FD`
Status (SR)	8	Each bit represents a status flag. Flags indicate the state of the CPU, or information about the result of the previous instruction. See the table below for a description of each flag

Status Flags

Bit	Symbol	Name	Description
7	N	Negative	Compare: Set if the register's value is less than the input value Otherwise: Set if the result was negative, i.e. bit 7 of the result was set
6	V	Overflow	Arithmetic: Set if a signed overflow occurred during addition or subtraction, i.e. the sign of the result differs from the sign of both the input and the accumulator `BIT`: Set to bit 6 of the input
5	-	(Unused)	Always set
4	B[1]	Break	Set if an interrupt request has been triggered by a `BRK` instruction
3	D	Decimal	Decimal mode[2] (https://wikipedia.org/wiki/Binary-coded_decimal): mathematical instructions will treat the inputs and outputs as decimal numbers. E.g. `$09 + $01 = $10`
2	I	Interrupt Disable	Disables interrupts while set
1	Z	Zero	Compare: Set if the register's value is equal to the input value `BIT`: Set if the result of logically ANDing the accumulator with the input results in 0 Otherwise: Set if result was zero
0	C	Carry	Carry/Borrow flag used in math and rotate operations Arithmetic: Set if an unsigned overflow occurred during addition or subtraction, i.e. the result is less than the initial value Compare: Set if register's value is greater than or equal to the input value Shifting: Set to the value of the eliminated bit of the input, i.e. bit 7 when shifting left, or bit 0 when shifting right

Memory layout

16-bit values are stored in memory in little-endian, so the least significant byte is stored before the most significant. E.g. if address `$0000` contains `$FF` and address `$0001` contains `$00`, reading a two-byte value from `$0000` will result in `$00FF`.

Signed integers are in two's complement and can represent values from -128 (`%11111111`) to +127 (`%01111111`). Bit 7 is set if the integer is negative.

The 6502's program counter is 16 bits wide, so up to 2^16 (65536) bytes of memory are addressable. Certain regions of memory are reserved for particular purposes:

Memory regions

Region	Contents	Description
`$0000` - `$00FF`	Zero page	The first page of memory, which is faster to access than other pages. Instructions can specify addresses within the zero page with a single byte as opposed to two, so instructions that use the zero page instead of any other page require one less CPU cycle to execute
`$0100` - `$01FF`	Stack	Last-in first-out data structure. Grows backwards from `$01FF` to `$0100`. Used by some transfer, stack, and subroutine instructions
`$0200` - `$FFFF`	General-purpose	Memory that can be used for whatever purpose needed. Devices that use the 6502 processor may choose to reserve sub-regions for other purposes, such as memory-mapped I/O

Memory Addressing Modes

Each instruction uses one of thirteen memory addressing modes, which determines the operand of the instruction. An example is provided for each.

Accumulator: A

The Accumulator is implied as the operand, so no address needs to be specified.

Example

Using the ASL (Arithmetic Shift Left) instruction with no operands, the Accumulator is always the value being shifted left.

```
ASL
```

Implied: i

The operand is implied, so it does not need to be specified.

Example

The operands being implied here are X, the source of the transfer, and A, the destination of the transfer.

```
TXA
```

Immediate:

The operand is used directly to perform the computation.

Example

The value $22 is loaded into the Accumulator.

```
LDA #$22
```

Absolute: a

A full 16-bit address is specified and the byte at that address is used to perform the computation.

Example

The value at address `$D010` is loaded into the X register.

```
LDX $D010
```

Zero Page: zp

A single byte specifies an address in the first page of memory (`$00xx`), also known as the zero page, and the byte at that address is used to perform the computation.

Example

The value at address `$0002` is loaded into the Y register.

```
LDY $02
```

Relative: r

The offset specified is added to the current address stored in the Program Counter (PC). Offsets can range from -128 to +127.

Example

The offset `$2D` is added to the address in the Program Counter (say `$C100`). The destination of the branch (if taken) will be `$C12D`.

```
BPL $2D
```

Absolute Indirect: (a)

The little-endian two-byte value stored at the specified address is used to perform the computation. Only used by the `JMP` instruction.

Example

The addresses `$A001` and `$A002` are read, returning `$FF` and `$00` respectively. The address `$00FF` is then jumped to.

```
JMP ($A001)
```

Absolute Indexed with X: a,x

The value in X is added to the specified address for a sum address. The value at the sum address is used to perform the computation.

Example

The value `$02` in X is added to `$C001` for a sum of `$C003`. The value `$5A` at address `$C003` is used to perform the *add with carry* (*ADC*) operation.

```
ADC $C001,X
```

Absolute Indexed with Y: a,y

The value in Y is added to the specified address for a sum address. The value at the sum address is used to perform the computation.

Example

The value `$03` in Y is added to `$F001` for a sum of `$F004`. The value `$EF` at address `$F004` is incremented (*INC*) and `$F0` is written back to `$F004`.

```
INC $F001,Y
```

Zero Page Indexed with X: zp,x

The value in X is added to the specified zero page address for a sum address. The value at the sum address is used to perform the computation.

Example

The value `$02` in X is added to `$01` for a sum of `$03`. The value `$A5` at address `$0003` is loaded into the Accumulator.

```
LDA $01,X
```

Zero Page Indexed with Y: zp,y

The value in Y is added to the specified zero page address for a sum address. The value at the sum address is used to perform the computation.

Example

The value `$03` in Y is added to `$01` for a sum of `$04`. The value `$E3` at address `$0004` is loaded into the Accumulator.

```
LDA $01,Y
```

Zero Page Indexed Indirect: (zp,x)

The value in X is added to the specified zero page address for a sum address. The little-endian address stored at the two-byte pair of sum address (LSB) and sum address plus one (MSB) is loaded and the value at that address is used to perform the computation.

Example

The value `$02` in X is added to `$15` for a sum of `$17`. The address `$D010` at addresses `$0017` and `$0018` will be where the value `$0F` in the Accumulator is stored.

```
STA ($15,X)
```

Zero Page Indirect Indexed with Y: (zp),y

The value in Y is added to the address at the little-endian address stored at the two-byte pair of the specified address (LSB) and the specified address plus one (MSB). The value at the sum address is used to perform the computation. Indeed addressing mode actually repeats exactly the Accumulator register's digits.

Example

The value `$03` in Y is added to the address `$C235` at addresses `$002A` and `$002B` for a sum of `$C238`. The Accumulator is then exclusive ORed with the value `$2F` at `$C238`.

```
EOR ($2A),Y
```

Instructions

These are the instructions for the 6502 processor including an ASCII visual, a list of affected flags, and a table of opcodes for acceptable addressing modes.

Load and Store

Load Accumulator with Memory: LDA

M -> A

Flags: N, Z

Addressing Mode	Opcode
a	AD
a,x	BD
a,y	B9
#	A9
zp	A5
(zp,x)	A1
zp,x	B5
(zp),y	B1

Load Index X with Memory: LDX

M -> X

Flags: N, Z

Addressing Mode	Opcode
a	AE
a,y	BE
#	A2
zp	A6
zp,y	B6

Load Index Y with Memory: LDY

M -> Y

Flags: N, Z

Addressing Mode	Opcode
a	AC
a,x	BC
#	A0
zp	A4
zp,x	B4

Store Accumulator in Memory: STA

A -> M

Flags: none

Addressing Mode	Opcode
a	8D
a,x	9D
a,y	99
zp	85
(zp,x)	81
zp,x	95
(zp),y	91

Store Index X in Memory: STX

X -> M

Flags: none

Addressing Mode	Opcode
a	8E
zp	86
zp,y	96

Store Index Y in Memory: STY

Y -> M

Flags: none

Addressing Mode	Opcode
a	8C
zp	84
zp,x	94

Arithmetic

Add Memory to Accumulator with Carry: ADC

A + M + C -> A

Flags: N, V, Z, C

Addressing Mode	Opcode
a	6D
a,x	7D
a,y	79
#	69
zp	65
(zp,x)	61
zp,x	75
(zp),y	71

Subtract Memory from Accumulator with Borrow: SBC

A - M - ~C -> A

Flags: N, V, Z, C

Addressing Mode	Opcode
a	ED
a,x	FD
a,y	F9
#	E9
zp	E5
(zp,x)	E1
zp,x	F5
(zp),y	F1

Increment and Decrement

Increment Memory by One: INC

M + 1 -> M

Flags: N, Z

Increment Index X by One: INX

X + 1 -> X

Flags: N, Z

Increment Index Y by One: INY

Y + 1 -> Y

Flags: N, Z

Addressing Mode	Opcode
a	EE
a,x	FE
zp	E6
zp,x	F6

Addressing Mode	Opcode
i	E8

Addressing Mode	Opcode
i	C8

Decrement Memory by One: DEC

M - 1 -> M

Flags: N, Z

Addressing Mode	Opcode
a	CE
a,x	DE
zp	C6
zp,x	D6

Decrement Index X by One: DEX

X - 1 -> X

Flags: N, Z

Addressing Mode	Opcode
i	CA

Decrement Index Y by One: DEY

Y - 1 -> Y

Flags: N, Z

Addressing Mode	Opcode
i	88

Shift and Rotate

Arithmetic Shift Left One Bit: ASL

C <- 7 6 5 4 3 2 1 0 <- 0

Flags: N, Z, C

Addressing Mode	Opcode
a	0E
a,x	1E
A	0A
zp	06
zp,x	16

Logical Shift Right One Bit: LSR

0 -> 7 6 5 4 3 2 1 0 -> C

Flags: N, Z, C

Addressing Mode	Opcode
a	4E
a,x	5E
A	4A
zp	46
zp,x	56

Rotate Left One Bit: ROL

C <- 7 6 5 4 3 2 1 0 <- C

Flags: N, Z, C

Addressing Mode	Opcode
a	2E
a,x	3E
A	2A
zp	26

Rotate Right One Bit: ROR

C -> 7 6 5 4 3 2 1 0 -> C

Flags: N, Z, C

Addressing Mode	Opcode
a	6E
a,x	7E
A	6A
zp	66

zp,x	36	zp,x	76

Logic

AND Memory with Accumulator: AND

A & M -> A

Flags: N, Z

Addressing Mode	Opcode
a	2D
a,x	3D
a,y	39
#	29
zp	25
(zp,x)	21
zp,x	35
(zp),y	31

OR Memory with Accumulator: ORA

A | M -> A

Flags: N, Z

Addressing Mode	Opcode
a	0D
a,x	1D
a,y	19
#	09
zp	05
(zp,x)	01
zp,x	15
(zp),y	11

Exclusive-OR Memory with Accumulator: EOR

A ^ M -> A

Flags: N, Z

Addressing Mode	Opcode
a	4D
a,x	5D
a,y	59
#	49
zp	45
(zp,x)	41
zp,x	55
(zp),y	51

Compare and Test Bit

The Negative (N), Zero (Z), and Carry (C) status flags are used for conditional (branch) instructions.

All Compare instructions affect flags in the same way:

Condition	N	Z	C
Register < Memory	1	0	0
Register = Memory	0	1	1
Register > Memory	0	0	1

Compare Memory and Accumulator: CMP

A - M

Flags: N, Z, C

Addressing Mode	Opcode
a	CD
a,x	DD
a,y	D9
#	C9
zp	C5
(zp,x)	C1
zp,x	D5
(zp),y	D1

Compare Memory and Index X: CPX

X - M

Flags: N, Z, C

Addressing Mode	Opcode
a	EC
#	E0
zp	E4

Compare Memory with Index Y: CPY

Y - M

Flags: N, Z, C

Addressing Mode	Opcode
a	CC
#	C0
zp	C4

Test Bits in Memory with Accumulator: BIT

A & M

Flags: N = M7, V = M6, Z

Addressing Mode	Opcode
a	2C
#	89
zp	24

Branch

Branch on Carry Clear: BCC

Branch if C = 0

Flags: none

Addressing Mode	Opcode
r	90

Branch on Carry Set: BCS

Branch if C = 1

Flags: none

Addressing Mode	Opcode
r	B0

Branch on Result not Zero: BNE

Branch on Result Zero: BEQ

Branch if Z = 0

Flags: none

Addressing Mode	Opcode
r	D0

Branch if Z = 1

Flags: none

Addressing Mode	Opcode
r	F0

Branch on Result Plus: BPL

Branch if N = 0

Flags: none

Addressing Mode	Opcode
r	10

Branch on Result Minus: BMI

Branch if N = 1

Flags: none

Addressing Mode	Opcode
r	30

Branch on Overflow Clear: BVC

Branch if V = 0

Flags: none

Addressing Mode	Opcode
r	50

Branch on Overflow Set: BVS

Branch if V = 1

Flags: none

Addressing Mode	Opcode
r	70

Transfer

Transfer Accumulator to Index X: TAX

A -> X

Flags: N, Z

Addressing Mode	Opcode
i	AA

Transfer Index X to Accumulator: TXA

X -> A

Flags: N, Z

Addressing Mode	Opcode
i	8A

Transfer Accumulator to Index Y: TAY

A -> Y

Flags: N, Z

Addressing Mode	Opcode
i	A8

Transfer Index Y to Accumulator: TYA

Y -> A

Flags: N, Z

Addressing Mode	Opcode
i	98

Transfer Stack Pointer to Index X: TSX

S -> X

Transfer Index X to Stack Pointer: TXS

X -> S

Flags: N, Z

Addressing Mode	Opcode
i	BA

Flags: none

Addressing Mode	Opcode
i	9A

Stack

Push Accumulator on Stack: PHA

A -> S

Flags: none

Addressing Mode	Opcode
i	48

Pull Accumulator from Stack: PLA

S -> A

Flags: N, Z

Addressing Mode	Opcode
i	68

Push Processor Status on Stack: PHP

P -> S

Flags: none

Addressing Mode	Opcode
i	08

Pull Processor Status from Stack: PLP

S -> P

Flags: all

Addressing Mode	Opcode
i	28

The processor status is stored as a single byte with the following flags bits from high to low: NV--DIZC.

Subroutines and Jump

Jump to New Location: JMP

Jump to new location by changing the value of the program counter.

Warning: When used with the absolute indirect addressing mode, a hardware bug can result in unexpected behavior when the specified address is `$xxFF`.
E.g. `JMP $(11FF)` will read the low byte from `$11FF` and the high byte from `$1100`, instead of reading the high byte from `$1200` as one would expect. This is due to an overflow in the lower byte of the indirect address not being carried into the upper byte.

Flags: none

Addressing Mode	Opcode
a	4C
(a)	6C

Jump to New Location Saving Return Address: JSR

Jumps to a subroutine

The address before the next instruction (PC - 1) is pushed onto the stack: first the upper byte followed by the lower byte. As the stack grows backwards, the return address is therefore stored as a little-endian number in memory.
PC is set to the target address.

Flags: none

Addressing Mode	Opcode
a	20

Return from Subroutine: RTS

Return from a subroutine to the point where it called with JSR.

The return address is popped from the stack (low byte first, then high byte).
The return address is incremented and stored in PC.

Flags: none

Addressing Mode	Opcode
i	60

Return from Interrupt: RTI

Return from an interrupt.

SR is popped from the stack.
PC is popped from the stack.

Flags: all

Addressing Mode	Opcode
i	40

Set and Clear

Clear Carry Flag: CLC

0 -> C

Flags: C = 0

Addressing Mode	Opcode
i	18

Set Carry Flag: SEC

1 -> C

Flags: C = 1

Addressing Mode	Opcode
i	38

Clear Decimal Mode: CLD

0 -> D

Flags: D = 0

Addressing Mode	Opcode

Set Decimal Mode: SED

1 -> D

Flags: D = 1

Addressing Mode	Opcode

i	D8

i	F8

Clear Interrupt Disable Status: CLI

0 -> I

Flags: I = 0

Addressing Mode	Opcode
i	58

Set Interrupt Disable Status: SEI

1 -> I

Flags: I = 1

Addressing Mode	Opcode
i	78

Clear Overflow Flag: CLV

0 -> V

Flags: V = 0

Addressing Mode	Opcode
i	B8

Miscellaneous

Break: BRK

Force an Interrupt

Flags: B = 1, I = 1

Addressing Mode	Opcode
i	00

No Operation: NOP

No Operation

Flags: none

Addressing Mode	Opcode
i	EA

Instruction table

Instruction table

High nibble	Low nibble															
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	BRK i	ORA (zp,x)				ORA zp	ASL zp		PHP i	ORA #	ASL A			ORA a	ASL a	
10	BPL r	ORA (zp),y				ORA zp,x	ASL zp,x		CLC i	ORA a,y				ORA a,x	ASL a,x	
20	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp		PLP i	AND #	ROL A		BIT a	AND a	ROL a	
30	BMI r	AND (zp),y				AND zp,x	ROL zp,x		SEC i	AND a,y				AND a,x	ROL a,x	
40	RTI i	EOR (zp,x)				EOR zp	LSR zp		PHA i	EOR #	LSR A		JMP a	EOR a	LSR a	
50	BVC r	EOR (zp),y				EOR zp,x	LSR zp,x		CLI i	EOR a,y				EOR a,x	LSR a,x	
60	RTS i	ADC (zp,x)				ADC zp	ROR zp		PLA i	ADC #	ROR A		JMP (a)	ADC a	ROR a	
70	BVS r	ADC (zp),y				ADC zp,x	ROR zp,x		SEI i	ADC a,y				ADC a,x	ROR a,x	
80		STA (zp,x)			STY zp	STA zp	STX zp		DEY i		TXA i		STY a	STA a	STX a	
90	BCC r	STA (zp),y			STY zp,x	STA zp,x	STX zp,y		TYA i	STA a,y	TXS i			STA a,x		
A0	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp		TAY i	LDA #	TAX i		LDY a	LDA a	LDX a	
B0	BCS r	LDA (zp),y			LDY zp,x	LDA zp,x	LDX zp,y		CLV i	LDA a,y	TSX i		LDY a,x	LDA a,x	LDX a,y	
C0	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp		INY i	CMP #	DEX i		CPY a	CMP a	DEC a	
D0	BNE r	CMP (zp),y				CMP zp,x	DEC zp,x		CLD i	CMP a,y				CMP a,x	DEC a,x	
E0	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp		INX i	SBC #	NOP i		CPX a	SBC a	INC a	
F0	BEQ r	SBC (zp),y				SBC zp,x	INC zp,x		SED i	SBC a,y				SBC a,x	INC a,x	

References

1. [1] (http://nesdev.com/the%20'B'%20flag%20&%20BRK%20instruction.txt), The B flag does not represent an actual CPU register

Further reading

- Owad, Tom, "Apple I Replica Creation", Syngress, 2005. ISBN 193183640X
- 6502.org (http://www.6502.org/) the 6502 microprocessor resource, particularly the Tutorials and Primers page (http://6502.org/tutorials/).
- NES Programming: The Nintendo Entertainment System uses a version of the 6502
- Super NES Programming: the Super NES uses the 65c816, a descendant of the 6502

- History of Apple Inc.: early Apple computers all used some version of the 6502
- History of Computers/The Rise of the Microcomputer
- X86 Disassembly/Disassemblers and Decompilers#Disassembly of 8 bit CPU code mentions some 6502 disassemblers
- Computer Programming/Hello world#Accumulator .2B index register machine: MOS Technology 6502.2C CBM KERNEL.2C MOS assembler syntax

---