ASM6502 Assembler Manual

Contents

1: Introduction

ASM6502 is a small two-pass assembler for the MOS Technology 6502 microprocessor used in many home computers of the 8-bit era. It consists of under 2K lines of C code and can be built with compilers conformant to the C89 standard.

ASM6502 implements some advanced features, like local labels, the ability to produce listing files, and the optimization of opcodes. In its current state, it is a usable assembler confirmed to generate the correct code for all supported instructions and addressing mode combinations. Due to the small size, two features are currently missing, namely conditional assembly and macros. The former may be implemented in the future, but the latter probably not.

The assembler outputs plain binary files.

The following listing contains a small sample program. It is the classic hello world for the Commodore C64. To assemble it, invoke the assembler with the source, output, and listing files as arguments:

```
asm6502 helloc64.a65 helloc64.prg helloc64.lst
```

```
 1: ; C64 Hello World
 2:
 3: LOAD_ADDR = $0801
 4:
 5:       .word LOAD_ADDR
 6:       .org  LOAD_ADDR
 7:
 8: CHROUT = $FFD2
 9: SYS    = $9E
10:
11: basic_upstart:
12:       .word @end, 10
13:       .byte SYS, " 2062",0
14: @end  .word 0
15:
16: start:
17:       ldx #0
18: @l    lda hello@msg,x
19:       jsr CHROUT
20:       inx
21:       cpx #hello@len
22:       bne @l
23:       rts
24:
25: hello:
```

```
26:   @msg .byte "HELLO, WORLD!", CR, LF
27:   @len = @ - @msg
28:
29: CR = 13
30: LF = %1010
```

2: Concepts and Terminology

The task of an assembler is to translate an *assembler source* containing human readable *instructions* to a *binary representation* a processor understands. This binary representation is called *machine code*. There is a one-to-one mapping between the human readable instructions contained in the assembler source and the generated machine code.

Each instruction a processor understands is given a name, called *mnemonic*. This name is choosen so that it describes what the instruction does. Each instruction is also assigned a numeric value, called the operation code or *opcode*. This is what the processor uses to decide what to do. Beside the instruction itself, additional information may be required to process it. The additional information is provided in the source by one or more *arguments* following the mnemonic. In machine code this additional information is encoded in binary form following the opcode.

`ADC #42` is an example of an instruction, where `ADC` is the mnemonic identifying the instruction, and `#42` is the argument. This particular instruction, understood by the MOS6502 processor adds the value 42 to the value stored in processor register A. It then writes the result back to A.

The set of instructions a CPU understands is called *instruction set*. There are many different kinds of CPUs and instruction sets. As a result, there is not something like 'the assembler', but there are many of them, all adapted to one or more specific instruction sets. ASM6502 is an assembler that is adapted to the instruction set of the MOS6502 processor family.

Beside generating an output file containing machine code, ASM6502 may also generate a *listing file* containing a *program listing*. This listing shows the lines of the assembler source side-by-side to the generated machine code in hexadecimal notation.

```
FPos  PC    Code          Line# Assembler text
0000  0000  69 2A            1: ADC #42
0002  0002  E8               2: INX
```

FPos indicates the position of the instructions regarding the output file. PC represents the *memory location* or *address* the code gets loaded to when executed. The executable code in memory is also called *image*.

Let's further elaborate what the arguments to instructions may be. In the example above, `#42` is a numeric value that is directly used to do the addition. It is therefore called an *immediate* value, and the mode the processor operates in is called *immediate addressing* mode. The `INX` instruction above implicitly operates on a register called X. For this reason it is called *implicit addressing*. Often, the argument specifies a memory location. This memory location may be specified with the beginning of the address space as a reference. In this case it is called *absolute addressing* mode. If the memory location is specified relative to some other location we call it *relative addressing* mode. Sometimes one does not want to encode a fixed memory location into the machine instruction, but instead use the content of some memory location as the address to operate on. This is called *indirect addressing*.

The sequence of instructions executed by the processor may be altered by the programmer utilizing special machine instructions. Some of these instructions modify this sequence unconditionally, and some alter it if a special condition is met. The instructions are called *jump* or *branching* instructions. The information of the *jump target* is encoded as address within the instruction. The assembler supports the programmer by letting him specify a jump target by giving it a name, called *label*. In the introductory example, `basic_upstart`, `start`, and `hello` are labels.

One task of an assembler is to assign an address to every label the programmer defines. Sometimes this is not that easy. For example, a label may be used as a jump target before it is defined in the assembler source. This is called *forward-reference*. That is the reason ASM6502 is a so called two-pass assembler. In the first pass, the assembler processes the whole file to determine the addresses of all labels. In the second pass all required information is provided, and the machine code is generated.

Using forward-references can sometimes lead to non-optimal machine code, because the assembler has to guess the size of the forward-referenced ‘thing’. In that case the programmer can support the assembler by giving a hint what type the referenced ‘thing’ is. Some *multi-pass assemblers* process the source more than two times and can optimize the machine code even in cases they encounter forward-referenced labels. But for the sake of simplicity ASM6502 does not do that.

Beside labels the programmer may also define *variables*. Variables may be assigned any kind of mathematical expression. Variables and labels are also called *symbols*, and the name identifying them is referred to as *identifier*.

Character sequences which by itself provide a value to the assembler, like the character sequence `42`, which represents the numeric value 42, are considered to be *literals*.

3: Syntax and Semantics

The following chapter describes the data model and the syntax accepted by the assembler.

3.1 Input Files

Input files should have `.A65` as extension to distinguish it from files written for other assemblers. Include-files should have `.I65` as extension. The files should be encoded in the ASCII or UTF-8 character sets.

3.2 Data Types

Two data types are known to the assembler:

- 8-bit *byte* storing numbers of value of 0..255.
- 16-bit *word* storing positive numbers of value 0..65535. Negative numbers are stored in two-complement representation with range -32768..32767.

3.3 Symbols

The assembler distinguishes two types of case-sensitive symbols: *labels* and *variables*. A label stores the address of the current instruction or directive. It is defined at the beginning of a line by appending its name with a colon. The colon may be left out if the label name is not an instruction mnemonic.

A variable is defined by assigning an expression to it. In the following example, hello is a label, and CHROUT is a variable.

```
CHROUT = $ffd2
hello:  jmp CHROUT
```

Labels and variables may be of type byte or word. A label is of type byte if it is assigned an address within the first 256 bytes (zero page). Otherwise, it is of type word. The data type of a variable is that of the expression assigned to it, unless it is forward-referenced.

Symbols may be forward-referenced. That means that they can be used in expressions before they are defined. Forward-referenced symbols are *always* of type word, regardless of what is assigned to them.

Labels may not be redefined. If a variable is assigned a value multiple times, it must be the same value. Otherwise, it is an illegal redefinition.

Symbols may be defined locally by prepending them with `@`. They are associated with the previous non-local label defined. They may be referenced within expressions locally by `@name` or with their qualified name `label@name` outside their local scope. Example:

```
        jmp hello@l   ; fully qualified label reference
hello:
  @l    jmp @l        ; local label reference
```

3.4 Expressions

There are many places where expressions may occur, for example on the right side of a variable definition or as an argument to a machine instruction. The most primitive form of an expression is a numeric constant, which can be given in decimal, hexadecimal, or binary. The value of the constant determines its type. A small value can be forced to be of type word by prepending zeros.

```
5     ; decimal byte constant
$a    ; hexadecimal byte constant
$4711 ; hexadecimal word constant
%1011 ; binary byte constant
$00a  ; hex word constant because more than 2 digits
0123  ; decimal word constant because more than 3 digits
'x'   ; byte typed ASCII character code of x
-1    ; word constant $FFFF (2-complement)
```

Arithmetic operations may be used in expressions. Operator precedence is respected, as in the following example:

```
2+3*5 ; yields value 17
@ - 2 ; current program counter - 2
```

The supported operations are the following:

- lowest precedence: unary byte select: low byte (<) and high byte (>)
- unary and binary addition (+) and subtraction(-), bit-wise or (|), exclusive or (^)
- multiplication (*), division (/), bit-wise and(&)
- highest precedence: expressions enclosed by parentheses

Examples:

```
<$4711    ; selects low byte $11
>$4711    ; selects high byte $47
+(x+2)*5
```

In the last example the unary + is only needed if used as an instruction argument to distinguish from 6502 indirect addressing mode.

The special symbol @ evaluates to the current value of the program counter. It may not be confused with a local label, like `@abc`.

3.5 Line Format

Each line may end with a comment started by a semicolon.

At the beginning of a line, a label may be specified if the line does not contain a variable definition.

```
start:              ; line consisting of a label
loop: BNE loop.     ; label and instruction
msg:  .byte "Hello" ; label followed by a directive
```

Variables are defined by giving the variable name followed by an equal sign followed by an expression yielding a numeric value of type byte or word:

```
CHROUT = $FFD2
```

3.6 Directives

Directives instruct the assembler to do certain things. They may or may not produce output data. Names of directives start with a dot. The directives currently known to the assembler are:

3.6.1 .BINARY directive

Copies binary data from a file to the output file. Numeric expressions specifying a start offset and a length may be given as arguments. If a length is given, the start offset must also be specified.

Example:

```
.BINARY "SPRITE.DAT".          ; copies the whole file
.BINARY "SPRITE.DAT", $10      ; skip the first 16 bytes
.BINARY "SPRITE.DAT", $10, 64 ; copy 64 bytes from offset 16
```

3.6.2 .BYTE directive

Produces one or more output bytes. The arguments are separated by a comma. Strings enclosed by " may also be used.

Example:

```
.BYTE 1
.BYTE 47, 11
.BYTE "Hello, World", 13, 10
```

3.6.3 .FILL directive

Starting from the current position of the output file, emits as many bytes as given by the first argument. If the second argument is given, the region is filled with its byte-sized value. Otherwise, it is filled with zero. The program counter is increased accordingly.

Example:

```
.FILL 100       ; fill 100 bytes with zero
.FILL 16, $EA   ; insert 16 NOPs ($EA) into the code
```

3.6.4 .INCLUDE directive

Substitutes the directive with the contents of a file given by the argument. As a convention the extension of include-files should be `.i65`.

Example:

```
.INCLUDE "c64krnl.inc"
```

3.6.5 .LIST and .NOLIST

If a listing file is given via command line, listing generation is initially enabled. If the user wants some parts of the code to be excluded from the listing, the region can be surrounded by `.NOLIST` and `.LIST` statements.

If listing generation is disabled when an `.INCLUDE` statement is processed, `.LIST` inside the included file has no effect.

The listing generation flag is restored when the processing of an included file finished. If a `.NOLIST` statement is contained in an include file and the listing is activated for the parent file, listing generation is resumed after processing the include file from the line after the `.INCLUDE` line.

3.6.6 .ORG directive

Sets the current program counter to the numeric value of the argument. Does not modify the offset into the output file. This means that .ORG can not be used to 'jump around' in the output file.

Example:

```
.ORG $0801
```

3.6.7 .WORD directive

Produces one or more output words.

Example:

```
.WORD $0801, 4711
```

3.7 Addressing Modes

Every assembler instruction consists of a mnemonic identifying the machine instruction followed by at most one numeric argument including addressing mode specifiers. Instruction mnemonics are case-insensitive. The assembler supports all MOS6502 addressing modes:

3.7.1 Implicit and accumulator addressing

Either no argument or accumulator is implicitly assumed by the instruction

```
CLC ; clear carry
ROR ; rotate accumulator right
```

3.7.2 Immediate Addressing

The byte-sized argument is encoded in the byte following the opcode. The argument for the assembler instruction is prefixed by # to indicate an immediate value. The argument may be any expression yielding a byte-sized numeric value.

```
LDA #42 ; load value 42 into the accumulator
```

3.7.3 Relative addressing

Relative addressing is only used by branch instructions. The branch offset in the range of -128 to

127 is encoded by the byte following the opcode. The assembler interprets the argument, which may be any numeric expression, relative to the current program counter.

```
loop: BNE loop
```

3.7.4 Absolute Addressing

A word-sized address is encoded following the opcode byte. The assembler interprets any word-sized expression following an instruction mnemonic as an absolute address.

```
LDA $4711 ; load contents of address $4711 into the accumulator
```

3.7.5 Zero-page addressing

A byte-sized address is encoded following the opcode byte. The assembler interprets any byte-sized expression following an instruction mnemonic as a zero page address.

```
LDA $47   ; load contents of address $47 into the accumulator
LDA >$4711  ; load contents of address $47 into the accumulator
```

3.7.6 Absolute X and absolute X addressing

The address is encoded in the word following the opcode and displaced by the contents for the X or Y register.

```
LDA $4711,X ; load contents of address $4711 displaced by X
LDA $4711,Y ; load contents of address $4711 displaced by Y
```

3.7.7 Zero-page X and Zero-page Y addressing

The address is encoded in the byte following the opcode and displaced by the contents for the X or Y register.

```
LDA $47,X ; load contents of address $47 displaced by X
LDX >$4711,Y ; get contents of address $47 displaced by Y into X
```

3.7.8 Indirect addressing

The word-sized address is stored in the memory location given by the word-sized argument. In assembler syntax, an indirect address is indicated by enclosing the argument in parentheses, like in the following.

```
JMP ($4711)
```

The following one is a syntax error because the assembler assumes indirect addressing mode instead of a sub-expression grouped by parentheses:

```
JMP (2+3)*1000
```

If one wants to start an expression with an opening parentheses, while not indicating indirect addressing to the assembler, one can write:

```
JMP +(2+3)*1000
```

This one is correct (indirect addressing):

```
JMP ((2+3)*1000)
```

3.7.9 Indexed indirect by X and indirect indexed by Y addressing

Indirect indirect by X addresses the byte referenced by the contents of the word stored at zero page address b + X. Indirect indexed by Y adds Y to the address word stored in zero page address b to calculate the address to operate on.

```
b = 15
ORA (b,X)
ORA (b),Y
```

Appendix A: Instruction Reference

In the following instruction list, `#$42` is a representative for a byte-sized immediate value. This value may be substituted by any other byte-sized value. `$15` is a representative for a zero-page memory address, and `$4711` is a representative for a word-sized memory address.

The first hexadecimal value on a line is the instruction opcode followed by at most two bytes of additional data defined by the instruction argument. The syntax of the different addressing modes is described in one of the previous chapters.

A.1 ADC - add with carry

Flags: N Z C V

```
69 42      adc #$42
65 15      adc $15
75 15      adc $15,x
6D 11 47   adc $4711
7D 11 47   adc $4711,x
79 11 47   adc $4711,y
61 15      adc ($15,x)
71 15      adc ($15),y
```

A.2 AND - bit-wise and with accumulator

Flags: N Z

```
29 42      and #$42
25 15      and $15
35 15      and $15,x
2D 11 47   and $4711
3D 11 47   and $4711,x
39 11 47   and $4711,y
21 15      and ($15,x)
31 15      and ($15),y
```

A.3 ASL - arithmetic shift left

Flags: N Z C

```
0A         asl
06 15      asl $15
16 15      asl $15,x
0E 11 47   asl $4711
1E 11 47   asl $4711,x
```

A.4 BCC - branch if carry cleared

```
90 FE       bcc @
```

A.5 BCS - branch if carry set

```
B0 FE       bcs @
```

A.6 BEQ - branch if equal

```
F0 FE       beq @
```

A.7 BIT - test bits

Negative flag becomes the bit 7 of the operand, overflow flag becomes bit 6 of the operand. Zero flag is set if the bit-wise and operation between the accumulator and the operand is zero, otherwise it is cleared.

Flags: N Z V

```
24 15       bit $15
2C 11 47    bit $4711
```

A.8 BMI - branch if negative

```
30 FE       bmi @
```

A.9 BNE - branch if not equal

```
D0 FE       bne @
```

A.10 BPL - branch if positive

```
10 FE       bpl @
```

A.11 BRK - force break

BRK cannot be masked by setting interrupt disable flag. Forces the processor to continue at the address stored in the IRQ vector $FFFE. Pushes the flags with set break (B) flag to differentiate from a hardware interrupt. RTI and PLP ignore the break flag.

Flags: I=1

```
00          brk
```

A.12 BVC - branch if overflow flag cleared

```
50 FE       bvc @
```

A.13 BVS - branch if overflow flag set

```
70 FE       bvs @
```

A.14 CLC - clear carry flag

Flags: C=0

```
18          clc
```

A.15 CLD - clear decimal flag

Flags: D=0

```
D8          cld
```

A.16 CLI - clear interrupt disable flag

Flags: I=0

```
58          cli
```

A.17 CLV - clear overflow flag

Flags: V=0

```
B8          clv
```

A.18 CMP - compare with accumulator

Flags: N Z C

```
C9 42       cmp #$42
C5 15       cmp $15
D5 15       cmp $15,x
CD 11 47    cmp $4711
DD 11 47    cmp $4711,x
D9 11 47    cmp $4711,y
C1 15       cmp ($15,x)
D1 15       cmp ($15),y
```

A.19 CPX - compare with X register

Flags: N Z C

```
E0 42       cpx #$42
E4 15       cpx $15
EC 11 47    cpx $4711
```

A.20 CPY - compare with Y register

Flags: N Z C

```
C0 42       cpy #$42
C4 15       cpy $15
CC 11 47    cpy $4711
```

A.21 DEC - decrement

Flags: N Z

```
C6 15       dec $15
D6 15       dec $15,x
CE 11 47    dec $4711
DE 11 47    dec $4711,x
```

A.22 DEX - decrement X register

Flags: N Z

```
CA          dex
```

A.23 DEY - decrement Y register

Flags: N Z

```
88          dey
```

A.24 EOR - exclusive or

Flags: N Z

```
49 42       eor #$42
45 15       eor $15
55 15       eor $15,x
4D 11 47    eor $4711
5D 11 47    eor $4711,x
59 11 47    eor $4711,y
41 15       eor ($15,x)
51 15       eor ($15),y
```

A.25 INC - increment

Flags: N Z

```
E6 15       inc $15
F6 15       inc $15,x
EE 11 47    inc $4711
FE 11 47    inc $4711,x
```

A.26 INX - increment X register

Flags: N Z

```
E8          inx
```

A.27 INY - increment Y register

Flags: N Z

```
C8          iny
```

A.28 JMP - jump

```
4C 11 47   jmp $4711
6C 11 47   jmp ($4711)
```

A.29 JSR - call subroutine

```
20 11 47   jsr $4711
```

A.30 LDA - load accumulator

Flags: N Z

```
A9 42      lda #$42
A5 15      lda $15
B5 15      lda $15,x
AD 11 47   lda $4711
BD 11 47   lda $4711,x
59 11 47   eor $4711,y
A1 15      lda ($15,x)
B1 15      lda ($15),y
```

A.31 LDX - load X register

Flags: N Z

```
A2 42      ldx #$42
A6 15      ldx $15
B6 15      ldx $15,y
AE 11 47   ldx $4711
BE 11 47   ldx $4711,y
```

A.32 LDY - load Y register

Flags: N Z

```
A0 42      ldy #$42
A4 15      ldy $15
B4 15      ldy $15,x
AC 11 47   ldy $4711
BC 11 47   ldy $4711,x
```

A.33 LSR - logical shift right

Flags: N=0 Z C

```
4A         lsr
46 15      lsr $15
56 15      lsr $15,x
4E 11 47   lsr $4711
5E 11 47   lsr $4711,x
```

A.34 NOP - no-operation

```
EA          nop
```

A.35 ORA - bit-wise or with accumulator

Flags: N Z

```
09 42       ora #$42
05 15       ora $15
15 15       ora $15,x
0D 11 47    ora $4711
1D 11 47    ora $4711,x
19 11 47    ora $4711,y
01 15       ora ($15,x)
11 15       ora ($15),y
```

A.36 PHA - push accumulator on stack

```
48          pha
```

A.37 PHP - push flags on stack

```
08          php
```

A.38 PLA - pop accumulator from stack

Flags: N Z

```
68          pla
```

A.39 PLP - pop flags from stack

Flags: N Z C I D V

```
28          plp
```

A.40 ROL - rotate left through carry

Flags: N Z C

```
2A          rol
26 15       rol $15
36 15       rol $15,x
2E 11 47    rol $4711
3E 11 47    rol $4711,x
```

A.41 ROR - rotate right through carry

Flags: N Z C

```
6A          ror
66 15       ror $15
```

```
76 15      ror $15,x
6E 11 47   ror $4711
7E 11 47   ror $4711,x
```

A.42 RTI - return from interrupt

Flags: N Z C I D V

```
40         rti
```

A.43 RTS - return from subroutine

```
60         rts
```

A.44 SBC - subtract from accumulator with carry

Flags: N Z C V

```
E9 42      sbc #$42
E5 15      sbc $15
F5 15      sbc $15,x
ED 11 47   sbc $4711
FD 11 47   sbc $4711,x
F9 11 47   sbc $4711,y
E1 15      sbc ($15,x)
F1 15      sbc ($15),y
```

A.45 SEC - set carry flag

Flags: C=1

```
38         sec
```

A.46 SED - set decimal flag

Flags: D=1

```
F8         sed
```

A.47 SEI - set interrupt disable flag

Flags: I=1

```
78         sei
```

A.48 STA - store accumulator

```
85 15      sta $15
95 15      sta $15,x
8D 11 47   sta $4711
9D 11 47   sta $4711,x
99 11 47   sta $4711,y
81 15      sta ($15,x)
91 15      sta ($15),y
```

A.49 STX - store X register

```
86 15      stx $15
96 15      stx $15,y
8E 11 47   stx $4711
```

A.50 STY - store Y register

```
84 15      sty $15
94 15      sty $15,x
8C 11 47   sty $4711
```

A.51 TAX - transfer X to accumulator

```
AA         tax
```

Flags: N Z

A.52 TAY - transfer Y to accumulator

```
A8         tay
```

Flags: N Z

A.53 TSX - transfer X to stack pointer

Flags: N Z

```
BA         tsx
```

A.54 TXA - transfer accumulator to X

Flags: N Z

```
8A         txa
```

A.55 TXS - transfer stack pointer to X

Flags: N Z

```
9A         txs
```

A.56 TYA - transfer accumulator to Y

Flags: N Z

```
98         tya
```