6502 Assembler

by Bernd Boeckmann

1 Introduction

This document describes syntax and semantics of my 6502 assembler. The definitions given in the next section are assumed to be known for the rest of this document.

2 Definitions

2.1 Assembler file

A file holding *assembler text*. The character encoding of the assembler text has to be ASCII (or to be restricted to the ASCII compatible subset of another encoding). Line endings may be of DOS or UNIX style.

2.2 Assembler unit

Assembler text processed by one invocation of the assembler. An assembler unit may consist of several assembler files put together by include directives.

2.3 Data types

Three data types are known to the assembler: *byte*, *word* and *undefined*. A byte can store values from decimal 0 to 255 equaling $FF hex. A word can store values from 0 to 65535 equaling $FFFF. The data type undefined indicates an unknown numeric value of unknown size.

2.4 Value

A numeric value may either be interpreted as a *plain number* or as *an address*, which of both is context specific and depends mainly on the addressing mode of the instruction being processed. Each numeric value belongs to of one of the defined numeric data types.

2.5 Expression

An expression is a region of assembler text that, when evaluated by the assembler, resolves to a numeric value of one of the specified types or to an unknown value of type undefined.

2.6 Address

A memory location number that when encountered in an expression evaluates to its numeric value of type word. The address space is that of the translated assembler unit running in its execution environment.

2.7 Labels and Variables

Labels and variables, each associated with a unique name represented by an identifier, store numeric values and their corresponding type. The purpose of labels is to store memory addresses. Therefore the numeric values stored by

labels are defined to be of type word. Numeric values stored by variables may be of type byte or word. Labels and variables do not occupy storage of the translated assembler unit or memory of its running image but of the assembler process itself.

2.8 Identifier

An identifier is a region of assembler text consisting of alpha numerical characters that represents a label, variable or machine opcode. Identifiers must start with a letter. If contained in an expression, an identifier naming a label or variable evaluates to the numeric value the label or variable is defined to.

3 Syntax

In this syntax definition all characters that appear literally in the assembler text are enclosed by ''.
Assembler text

source := { *line* }

The assembler text consists of any number of lines. The syntax elements defined here are separated by an arbitrary number of white space. Each line ends with either carriage return (UNIX) or carriage return followed by line feed (DOS/Windows).
Comments may be embedded in the assembler text as the last item of a line. A comment starts with semicolon **';'** and reaches till the end of line.

3.1 Number literals

number := *decnum* | *hexnum*
decnum := '0'..'9' { '0'..'9' }
hexnum := '$' '0'..'9' | *'a'..'f'* | *'A'..'F'* { '0'..'9' | *'a'..'f'* | *'A'..'F'* }

Number literals may be given in decimal or in hexadecimal. Hexadecimal numbers start with '$'. Any number literal those value fits in a byte has data type byte under the following conditions:

- if it is a hexadecimal number the digit count must be less than three
- if it is a decimal number the digit count must be less than four.

This gives a convenient way of defining word sized numeric literals for small values by prefixing the number with zeros.

3.2 Identifiers

identifier := 'a'..'z' | 'A'..'Z' | '_' { 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' }

3.3 Line format

line := vardef | *labeldef* | *instruction* | *labeldef instruction* | *empty line*

Each line of assembler text may contain either

- nothing (empty line),
- a label definition,

- an instruction,
- a label definition followed by an instruction,
- a variable definition or
- an assembler directive.

3.4 Label definition

labeldef := *identifier* ':'

A line may start with a label definition optionally followed by an assembler instruction. The program counter before processing the line is assigned to the label defined by the name to the left of the colon.

3.5 Assembler instruction

instruction := *mnemonic* [*argument*]

3.6 Variable definition

vardef := *identifier* '=' *expression*

A variable definition evaluates the expression to the right of the equal sign and assigns its value to the variable identified by the name to the left of the equal sign.

4 Addressing modes

Throughout the following description of the supported addressing modes, parts in brackets are optional. Bold text is to be entered literally. Italic text has to be substituted. *Opc* stands for a machine instruction mnemonic, *imm*, for an 8-bit unsigned immediate value, *rel8* for an 8-bit relative address. *Abs16* represents a 16-bit unsigned absolute address. Examples are given in typewriter font.

4.1 Implicit addressing

opc

The instruction takes no argument or the argument is given explicitly in the instruction opcode. Instruction size is one byte.

Examples

```
CLC       ; clear carry flag
```

4.2 Accumulator addressing

opc
opc **A**

Instructions operating on the accumulator A. If A is not given as an argument it is implicitly assumed. Instruction size is one byte.

Examples

```
ROR       ; rotate register A right through carry
SHL A     ; shift A to the left
```

4.3 Immediate addressing

opc #imm

An expression yielding an immediate (byte) value must be given as an argument for the instruction. Instruction size is two bytes.

Examples

```
ADC #1
ADC #(2 + $1 * 3)
```

4.4 Relative addressing

opc rel8

Mainly used by conditional jumps. While argument is a signed 8-bit offset relative to the current program counter, the assembler expects an expression yielding a 16-bit unsigned number holding an absolute address. That address gets converted by the assembler to an 8-bit relative number. Instruction size is two bytes.

Examples

```
BNE next  ; next is a label defined anywhere else
BE  *-2   ; * is current program counter
```

4.5 Absolute addressing

opc abs16

The operand is the content of the memory address given by the 16-bit unsigned argument abs16.

Examples

```
LDA $FF02 ; load accumulator with contents of addr $FF02
```

4.6 Absolute addressing, X

opc abs16 **, X**

The operand is the content of the memory address given by the 16-bit unsigned argument abs16 displaced by the 8-bit unsigned value in register X.

4.7 Absolute addressing, Y

opc abs16 **, Y**

The operand is the content of the memory address given by the 16-bit unsigned argument abs16 displaced by the 8-bit unsigned value in register X.

5 Supported instructions

6 Assembly Phases

Assembly is performed by to phases.

1. Determination of all instruction sizes and therefore the addresses of all labels by processing the assembler text once (pass one).
2. Final determination of all variable values and generation of machine code for the given instructions by processing the assembler text a second time (pass two).