Everyone Likes Some Assembling (ELSA): the native 65C816 assembler

An overview

Version 0.9

Preface

ELSA is a clone of the MAE assembler by John Harris. This was the assembler I had used since around 1996, when I discovered it and switched to it from MAC/65.

I sometimes still use MAE, but as years were passing, MAE was becoming a bit too limited to my needs. Since that assembler is apparently no longer developed, and since its author, John Harris, has refused to publish its source code, I was forced to start writing my own assembler. It of course has the flavour of MAE, the assembler I was accustomed to. But I have not used any part of MAE's code: the code is entirely my own, the ELSA assembler only mimics most of the MAE's syntax, diverging whenever I thought I had a better idea.

Also, ELSA is only an assembler compiler. Unlike MAE, it does not contain an editor or a disassembler/debugger. For that last, I am currently working on my own disassembler for 65C816, and for the editor I still use MAE.

ELSA is entirely written in 65C816 native code and makes use of the RAM past the first 64k: it stores the symbol table there, thus making it virtually unlimited. In the base 64k the program occupies around 25k.

One important similarity between ELSA and MAE is that ELSA, like MAE, was written entirely on Atari. First versions were written and compiled under MAE, later versions compiled with themselves, still however being written in MAE's excellent editor.

For years it was my private program, not really intended for public release. But it apparently has grown so that it can be shown to other people. So it will be with the hope that it turns out to be useful to someone.

KMK/DLT
Warszawa, February-March 2020

Table of contents

I. Operation

As said in the Preface, ELSA is MAE's clone, so first of all it is good to get some acquaintance with that assembler. The MAE User's Manual you can find on the Net will supply you with the basic information on this topic:

http://www.mixinc.net/atari/mae.htm

Unlike MAE (which stands for Macro-Assembler-Editor), ELSA is an assembler compiler only, contains no editor nor debugger.

Also, unlike MAE, ELSA aborts the assembling on first error, emits a bell signal (ASCII 253), and quits to DOS. This allows you to leave the computer alone doing a larger assembly builds instead of being forced to watch the screen constantly for error messages or miss them having been scrolled up out of the display.

Unlike MAE, which compiles from the memory to the memory or from memory to an object file, ELSA compiles from the source file to the object file only. Therefore it is good to have a fast hard drive as storage.

Other requirements are:

1) 65C816 CPU operating at least at 1.77 MHz;
2) 65C816 compatible OS ROM, like DracOS (also known as Rapidus OS);
3) SpartaDOS, preferably SpartaDOS X;

Recommended:

4) at least 64k of the 65C816 High RAM, also known as the linear memory (the flat RAM past the address $00FFFF).

II. Command line arguments

The syntax is:

ELSA [options] source_file_name.ext [options]

The options are:

Option	Function
/C	Case-insensitive labels: with this option the labels "ADR" and "adr" are identical.
/Dlabel=value	Assign "value" to the label named "label" and insert this label into the symbol table before the first assembly pass. Example: /DSTART=$2000
/L	Generate assembly listing during the second pass. The listing will appear on the screen, being formatted for 80-column displays. This switch has a priority over .LS and .LC directives possibly inserted into the source code (i.e. .LC will not be able to switch off the listing if it was enabled with -L).
/Mtarget	Define default target CPU. The available targets are: 6502, 65c02, 65sc02, 65c802 and 65c816. When no target is specified, 65C816 is assumed. How the targets are defined and what are the effects of selecting a particular target CPU, it is explained when the corresponding assembly directives are discussed. Example: /M65C802.
/Ofname.ext	Define the object file name. This switch has a priority over .OUT directives placed within the source code: then /O is specified, any .OUT will be ignored and a warning message will be printed on the screen. When the object file name is defined neither in the command line nor in the source code, the object code will not be saved anywhere.
/Q	Quiet assembly, i.e. suppress warnings.
/P	Warn about branches crossing a page boundary.
/U	Report all unreferenced internal addresses after the second pass. This is reported by default in the final message as "n LABELS DEFINED (m NEVER USED)", adding the switch just causes the unreferenced labels to be explicitly listed.
/V	Report all unused labels after the second pass. Unlike /U, this lists all unused labels regardless of their function, i.e. whether they are meaning addresses or values or whatever. Also the "never used" label count in the final message will then be different than the default one.

Instead of the "/" sign, the minus sign may be used, e.g. -M65C802 is perfectly valid.

III. General assembler syntax

As said above, ELSA is a clone of MAE. In the area of the syntax, MAE is in turn generally following the style of MAC/65, so that switching from the latter to the former makes no trouble. The MAE's oddity is that it only respects the first three characters of the name of a directive, so for example writing in the source .WORD or .WO makes no difference. ELSA keeps many of these quirks for (my) conveni-

ence, but the short forms are in fact explicit aliases for their longer equivalents.

IV. Labels

A label may be up to 245 characters long, which means that there is no practical size limit. The label's first character must be a letter, apart from that the decimal digits, the @ character, the dot (".") and the underscore character ("_") are allowed in the body of a label. The question mark, for the reason explained below, is only allowed as either the first or as the last character of a label (therefore such an expression as BOOT? = $09 is perfectly valid).

In the area of labels, the most notable feature of MAE is the system of local labels marked with "?" character at the beginning. Such a label will serve as a local one in the area between two consecutive global labels. To reference such a local label, just prefix its name with the "?" character (e.g. LDA ?SIZE). When a reference to a local label is required from the outside of its global scope, the respective global label should be used followed by "?" and by the local label the reference is being made to (e.g. LDA IOCB?ICAX1,X). ELSA follows this system as a simple and elegant solution of the problem of label locality.

Any other label is a global label (unless stated otherwise).

The directive .LOCAL *namespace* defines higher level of locality, not to be confused with the aforementioned system modelled after MAE (this may be used without using the .LOCAL keyword). All labels, no matter if "global" or "MAE-style local", when defined between two .LOCAL directives, belong to the local namespace defined by the first of them only. This allows strict separation of local namespaces from the main program and from each other, so that even the same include files, defining the same global labels, may be used multiple times in different parts of the program.

An obvious example is an init segment, which gets overwritten after use: within it you may use the same library procedures and system calls as within the rest of the program, but you do not want to reference accidentally from within the main program something which was only temporarily defined for the init segment.

When a reference between different namespaces is required, the label referenced should be preceded with the name of its namespace and a colon (":", e.g. JMP INIT0:START). The global namespace has no name, so when a reference to a global label is required from within a local namespace, the label being referenced should be preceded with a colon only (e.g. LDA :KBCODES).

References to a MAE-style local label defined within a local namespace from the outside of that namespace are not allowed.

V. Expressions

Like MAE, ELSA does not pay attention to arithmetic operator precedence, the expressions are evaluated straight from left to right, and there are no parentheses. Some day I will have to fix this, probably.

The asterisk ("*"), as in most other assemblers, means the current value of the

PC. But, unlike in MAC/65, it is a read-only symbol and you cannot assign it a new value; so the expression "*=*+value", commonly used in MAC/65 to reserve memory space of the "value" length, will not work - you have to use the .DS directive instead.

VI. Unary operators

Expressions:

Operator	Function
!	Negate the result of the evaluation by applying XOR -1 (one's complement). This operator is applied before the ones mentioned below: <, > and ^.
-	Negate the result of the evaluation by applying (XOR -1) + 1 (two's complement). This operator is applied before the ones mentioned below: <, > and ^.
+	Do nothing.
<	Extract the bits 0-7 of the evaluation's result.
>	Extract the bits 8-15 of the evaluation's result.
^	Extract the bits 16-23 of the evaluation's result.

Addressing modes:

Operator	Function
#	Force the immediate addressing mode (e.g. LDA #VALUE)
<	Force the zero-page addressing mode (e.g. LDA <VALUE)
|	Force the absolute (16-bit) addressing mode (e.g. LDA |VALUE)
!	Same as the | (e.g. LDA !VALUE).
>	Force the long absolute (24-bit) addressing mode (e.g. LDA >VALUE).

These latter ones will be applied first to arguments to mnemonics, then the assembler will proceed normally with the expression evaluation. So STA !0 (address 0 with forced 16-bit addressing) will produce $8D $00 $00, and STA !!0 will produce $8D $FF $FF (the first ! forces 16-bit addressing mode, the subsequent one negates the result of the argument evaluation).

VII. Binary operators

The arithmetic operators +, -, *, /, % (modulo) work as expected. There are slight differences between MAE and ELSA in the area of comparison operators:

MAE	ELSA	Function
=	=	Equal
#	<>	Different

>	>	Greater
<	<	Lesser
(none)	>=	Greater or equal
(none)	<=	Lesser or equal

Also, comparing to MAE, there are novelties in the logical operators:

MAE	ELSA	Function
&	&	binary AND
|	|	binary OR
^	^	binary XOR (EOR)
(none)	&&	logical AND
(none)	||	logical OR

VIII. Directives

The directives are keywords which are steering the process of assembling. In ELSA, as in MAC/65 and MAE, most of these keywords are preceded with a dot. It makes them more visible in the source code and also facilitates its parsing.

The directives must be located past the column 0 of your source file, i.e. there must be at least one space between them and the left margin. Only one directive is allowed per program line, unless stated otherwise.

Symbols used in the table below:

x - expression; w - expression, word value; b - expression, byte value; lb - label name. All these "expressions" must evaluate in the first assembly pass.

Directive	Alias	Synopsis	Examples

[
...
]
		Define a block of code to be used with the conditional pseudo-instructions Rcc and Scc. If the block contains no code or data directives, the assembler will generate a warning.	LDY #$00
			[
			  LDA $2000,Y
			  STA $3000,Y
			  INY
			]
			RNE

.6502	.02	Set 6502 as the current target. Implies .RB. The target CPU is defined as a subset of 65C02, any instruction that does not belong to that subset will generate a warning.	.6502
.65C02	.c02	Set 65C02 as the current target. Implies .RB. The target CPU is defined as a subset of 65SC02, any instruction that does not belong to that subset will generate a warning.	.65C02
.65C802	.802	Set 65C802 as the current target. The target CPU	.65C802

		is practically the 65C816, just the instructions related to 24-bit addressing (operational, but pretty much useless on 65C802) will generate warnings.	
.65C816	.816 .65816	Set the 65C816 as the current target. This is the default, unless overridden in the command line or in the source code.	.65C816
.65SC02		Set 65SC02 (slightly modified 65C02 produced by Rockwell and WDC) as the current target. Implies .RB. The target CPU is defined as a subset of 65C802, any instruction that does not belong to that subset will generate a warning. Rockwell's BBR/BBS instructions and such (which are not continued in 65C802) are not supported.	.65SC02
.AB		Accumulator Byte: tell the assembler, that the current accumulator size is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AW, .IB, .IW, .RB, .RW	.AB
.AW		Accumulator Word: tell the assembler, that the current accumulator size is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AB, .IB, .IW, .RB, .RW	.AW
.BIN	.BI	Binary Include. Unlike in MAE, the contents of the file is not interpreted in any way, it is just verbatim inserted into the object code. Related: .INCLUDE	.BIN PIC.BMP
.BYTE	.BY	Inject the given byte values to the output file. The consecutive "bytes" separated with commas can be: decimal numbers, hexadecimal numbers preceded with the $ character, single ASCII characters preceded with the apostrophe, labels, arithmetic expressions, or text strings included in the double quotation marks. When the first numeric value is preceded with the + or - sign, this value will be added to or subtracted from, respectively, the rest of the values generated by this directive. Related: .CBYTE, .SBYTE	.BYTE 0,$FF,'A,"Hey" .BYTE <VAL,>VAL .BYTE SIZE*2+1 .BYTE +$80,"HELLO" .BYTE -$20,"caps"
.CBYTE	.CB	As .BYTE, except that the last byte generated by the single .CBYTE directive will be "inverted" (i.e. EORed with $80). Related: .BYTE, .SBYTE	.CBYTE "LOAD" .CBYTE +$20,"CAPS"
.CODE		Switch the PC to the code section. In practice, it marks an end of the block which was started	.ZP XY .DS 2

		with .ZP. Related: .ZP	.CODE LDA XY
.DBYTE		As .WORD, but with the inverted order of the bytes (i.e. MSB first). Related: .WORD, .TBYTE	.DBYTE $07FF,13
.DC w b		Define Constant-filled block. The consecutive 16-bit "w" number of bytes will be filled with the 8-bit value of "b". Related: .DS	.DC 345 $FF
.DS w		Declare Storage. It reserves the "w" number of bytes as uninitialized data array (as small as 1 byte). In other words, it does not generate code or data, it just adds the given number to the current PC during assembling, thus making an empty "gap" in the memory. Related: .DC	.DS 32
.ELSE	.EL	This inverts the result of the expression evaluation made by the .IF directive. Related: .IF, .ENDIF, .IFDEF, .IFNDEF	(see .IF)
.END		Closes the object code file and ends the assembling. Related: .ORG	.END
.ENDIF	***	This ends the conditional block started with .IF. Related: .IF, .ELSE, .IFDEF, .IFNDEF	(see .IF)
.ENDR		Marks the end of the block started with .REPT. Related: .REPT	(see .REPT)
.ERROR		Just like .PRINT, but after printing out the required text it also aborts the assembling with an error message. Obviously it makes sense within a conditional block only (.IF / .ENDIF). Related: .PRINT	.ERROR "LM=",LM
.FLOAT	.FL	Store its arguments, separated by commas, in the Floating Point 6-byte DCB format for use with the OS ROM's Floating Point package. Related: .WORD, .LONG	.FLOAT 3.14,262144
.HEX	.HE	Generate a series of hexadecimal numbers. It is similar to .BYTE, but in the particular case of hex number may be more handy, because does not stipulate them to be preceded with the $ sign and the separator is space. Related: .BYTE	.HEX AE 19 00 44 FF
.IB		Index registers Byte: tell the assembler, that the current X and Y register size is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AB, .AW, .IW, .RB, .RW	.IB
.IF x		The beginning of the conditional block. If "x" is evaluated as true, the lines immediately following the .IF directive will be interpreted, or ignored otherwise. The conditional blocks can be	.IF __M6502__=1 .PRINT "NMOS" .ELSE .PRINT "CMOS"

		nested up to 256 levels deep. When this limit is exceeded, the assembler will generate an error message. Related: .ELSE, .ENDIF, .IFDEF, .IFNDEF	.ENDIF .IF AB&&BC ...
.IFDEF lb		Returns TRUE if the label "lb" is defined, i.e. already present in the symbol table. Related: .IF, .ELSE, .ENDIF, .IFNDEF	.IFDEF USE_CIO .INCLUDE CIO.S .ENDIF
.IFNDEF lb		As above, just returns FALSE when the label 'x' is defined. Related: .IF, .ELSE, .ENDIF, .IFDEF	.IFNDEF RTCLOCK RTCLOCK=18 .ENDIF
.INCLUDE	.IN	Includes another source file. These directives can be nested (i.e. the included file may contain another .INCLUDE directives), just remember that the stack space is not unlimited. Related: .BIN	.INCLUDE MATH.S
.IW		Index registers Word: tell the assembler, that the current X and Y register size is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AB, .AW, .IB, .RB, .RW	.IW
.LC		Switch off ("clear") the assembly listing. Related: .LS, .LL	(see .LS)
.LL		List just the following line. Related: .LC, .LS	.LL STA 24*OFFSET+L,X
.LOCAL lb		Define new local namespace named "lb". This automatically ends the current local namespace and switches to the new one. When only terminating the local namespace and switching to the global one is needed, the directive should get no parameters. The "lb" label size is limited to 64 characters.	.LOCAL INIT0 INIT ... .ORG $02E2 .WORD INIT .LOCAL
.LONG	.LO	Stores long, 24-bit words in the object code, in the usual order of bytes, i.e. LSB first. Related: .WORD, .TBYTE	.LONG $F1234,99999
.LS		Switch on ("set") the assembly listing. Related: .LC, .LL	.LS .ORG $0600 START JMP $E477 .END .LC
.LSB		Stores in the memory the Least Significant Bytes of the given series of long, 24-bit word values. An equivalent of .BYTE <VALUE with a bit less typing. Related: .BYTE, .MSB, .USB	.LSB ADR1,ADR2
.MSB		Stores in the memory the Middle Significant Bytes of the given series of long, 24-bit word	.MSB ADR1,ADR2

		values. An equivalent of .BYTE >VALUE with a bit less typing. Related: .BYTE, .LSB, .USB	
.OPT		Specify additional assembly options. For now only one such option is available: H, with which you can suppress (or enable back) writing headers to the object code.	.OPT H- .OPT H+
.ORG w	.OR	Specifies the address where (a portion of) the object code has to be stored in the memory. The assembler is able to generate up to 1024 separate segments within one binary file. Related: .END	.ORG $2000
.OUT	.OU	Specifies the name of the object code. This directive will get ignored, if the output file was specified in the command line. When this file name is not specified either way, the object code will not be stored anywhere.	.OUT TEST.COM
.PRINT	.PR	Prints the given text during the assembling. ASCII strings must be included within double quotation marks, multiple arguments must be separated with commas. When nothing is given, .PRINT will just output an EOL character. Related: .ERROR	.PRINT "PC:",*
.RB		Tell the assembler, that the current size of registers AXY is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AB, .AW, .IB, .IW, .RW	.RB
.REPT w		Marks the beginning of the block of lines in your source file, which have to be repeated "w" times during assembling. The end of that block should be marked with .ENDR. Within the block, the pseudo-label __REPT__ contains the number of the current iteration, and the operator # when added to a label, makes it unique for each iteration. When "w" is zero, the pair REPT/ENDR will do nothing, and the assembler will generate a warning. The .REPT blocks cannot be nested. Related: .ENDR	LDA MATH .REPT 8 ASL ROL MATH+1 BCS SKIP# INC NULS SKIP# .ENDR
.RS w		Reserve space within a structure began by .RSSET. The space will be of "w" bytes. This is similar to .DS with the one exception that .DS adds to the PC (thus reserving memory), while .RS only adds to the internal counter of the structure, just defining offsets within it. The example shows how to define a structure DOT, then reserve space in memory for 100 DOTs, and how to reference it. Note that in this example the label DOT is not really used, it only begins the local area for the actual labels within	DOT .RSSET 0 ?CX .RS 1 ?CY .RS 1 ?CZ .RS 1 ?OP .RS 2 DSZ = __RSSIZE__ DOTS .DS DSZ*100 LDX #offset_of_a_dot LDA DOTS+DOT?CX,X

		the structure. This is also incredibly useful when defining offsets on the stack to be referenced with the b,S and (b,S),Y addressing modes. Related: .RSSET	
.RSSET w		Beginning of a structure. The "w" is the offset of the structure's first element. Related: .RS	
.RW		Tell the assembler, that the current size of registers AXY is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning. Related: .AB, .AW, .IB, .IW, .RB	.RW
.SBYTE	.SB	Like .BYTE, but understands the given bytes as ASCII values, and converts them to Atari screen codes before storing in the object code. Related: .BYTE, .CBYTE	.SBYTE "TIME:" .SBYTE +$60,"NAME"
SET lb = x		Change the value of the label "lb"which was already defined and has a value assigned. Note no dot at the beginning of this keyword.	SET zpc = $0100
.TBYTE		Stores 24-bit long words in the memory in the reverse order of bytes, i.e. MSB first. In other words, it is to .LONG like .DBYTE to .WORD. Related: .LONG	.TBYTE $ABCDEF
.USB		Stores in the memory the Upmost Significant Bytes of the given series of long, 24-bit word values. An equivalent of .BYTE ^VALUE with less typing. Related: .BYTE, .LSB, .MSB	.USB ADR1,ADR2
.WORD	.WO	Stores 16-bit words in the object code, in the usual order of bytes (LSB first). Related: .DBYTE, .LONG	.WORD $E477,20133
.ZP [b]		Switch the PC to the Zero Page section. This keyword allows to declare zero page variables anywhere in your source code. The first .ZP directive accepts an offset on the zero page as an argument - this offset is the first address on the zero page to be used in your program (otherwise the default offset will be $00, rarely a desired thing on Atari). From there you can declare your zero page variables using the .DS directives. The end of the declarations is marked with .CODE. Later, when you want to declare more zero page variables when writing your program, you do not have to add them to the first .ZP block (although it is of course allowed), but you may use the .ZP directive again and declare more variables, again marking the end of this block with .CODE, which will switch you back to the code section. This way the variables may be de-	.ORG $0600 START JSR GETA JMP LIFE .ZP $80 C1 .DS 1 C2 .DS 2 .CODE GETA LDA $D20A STA C1 LDA $D20A STA C2 RTS .ZP PTR .DS 2

		clared just before the subroutines which use them, and you do not have to trouble yourself with assigning the actual addresses, because the use of .ZP, .DS and .CODE directives will automatically perform sequential allocation. When the size of the data being declared on the zero page exceeds the limit (i.e. the "zero page PC" spans the $FF address within the "zero page section"), the assembler will generate an error message. Related: .CODE	.CODE LIFE LDA C1 STA PTR LDA C2 STA PTR+1 JMP (PTR)

IX. MAE's directives not supported in ELSA

Directive	Synopsis
.24	In MAE this enables 24-bit address calculations. In ELSA all expressions are evaluated as 24-bit values, so this directive has no purpose. It causes no error, however.
.EN	This is only supported as an alias for .END, and not as an alias for .ENDIF.
.MC	Move Code. This can be worked around using .OPT to temporarily suppress generating headers.
.MD .ME .MG	These are: Macro Definition, Macro End and Macro Global. They are not supported at the moment because ELSA does not support macros (yet).

X. Pseudo-labels

Pseudo-labels are keywords with a value, which can be used in arithmetic expressions just like normal labels. To differentiate a named pseudo-label from other labels, ELSA marks them by putting two underscore characters (__) before and after the label's name (examples below).

The pseudo-labels usually carry numeric values signalizing currently selected assembling settings, just as register sizes and such things. Therefore they are particularly useful in conditional blocks started with .IF.

Label's name	Synopsis
*	Current value of the Program Counter within the program being compiled. Note that the assembler maintains more than one Program Counter, e.g. there are separate Program Counters for the .ZP segment and for the .CODE segment.
__ASIZE__	Current Accumulator size in bytes, as selected by the directives: .AB, .AW, .RB and .RW.
__DATE__	Current date, day, month, year, stored as a 24-bit word. E.g. 13 February 2020 is represented as $14020d in the usual little-endian byte order: $0d, $02, $14. *If the computer is not running SpartaDOS X, for this function to work you have to install a SpartaDOS-compatible "Z:" device in the system*.
__ISIZE__	Current size of X and Y register in bytes, as selected by the respective directives: .IB, .IW, .RB and .RW.

__M6502__	This has value of 1, if the currently selected target CPU is 6502, and 0 otherwise.
__M65C02__	1, if the currently selected target CPU is 65C02, and 0 otherwise.
__M65SC02__	1, if the currently selected target CPU is 65SC02, and 0 otherwise.
__M65C802__	1, if the currently selected target CPU is 65C802, and 0 otherwise.
__M65C816__	1, if the currently selected target CPU is 65C816, and 0 otherwise.
__REPT__	Current iteration within the .REPT/.ENDR block.
__RSSIZE__	Current offset of the structure defined by the directives .RSSET and .RS.
__TIME__	Current time of day, stored as a 24-bit word. 24-hour clock is used, so e.g. 19:11:05 will be represented as $050b13, i.e. $13, $0b, $05 in the little endian byte order. *If the computer is not running SpartaDOS X, for this function to work you have to install a SpartaDOS-compatible "Z:" device in the system*.

XI. Pseudo-instructions

A pseudo-instruction (otherwise known as a macro-instruction) is a kind of a hard-coded macro: the assembler presents it under a single mnemonic, but during the assembling this mnemonic is expanded into a series of actual CPU instructions.

The general rules for a pseudo-instruction in ELSA are these:

1) a pseudo-instruction has to follow the syntax of a real instruction, i.e. only the otherwise existing addressing modes are allowed;

2) a pseudo-instruction must not generate confusing side effects, e.g. so that one which claims to modify the memory also modifies the accumulator and flags, without a warning.

ELSA implements these:

Syntax	Synopsis	Expands to
ADD ...	Add without carry. The same addressing modes are available as for the ADC, this pseudo-instruction is just 1 byte longer and takes 2 cycles more. Counter-part: SUB.	CLC ADC ...
ASR	Arithmetical Shift Right. As LSR, but bit 7 is pre-served. Implied addressing only. 3 and 4 cycles for 8-bit Accumulator, or 4 bytes and 5 cycles for a 16-bit one.	CMP #$80 ROR
B2H	Binary To Hex: convert the lowest nibble of the accu-mulator into the corresponding hex digit, and store the digit in the lowest byte of the Accumulator. If other nibbles of the Accumulator (8-bit or 16-bit in respective modes) contain anything but zeros, this in-struction may yield undefined results. 6 bytes, 8 cycles for 8-bit Accumulator, 8 and 10 respectively for 16-bit Acc.	CMP #$0A SED ADC #$30 CLD
BSL *address*	Branch to Subroutine Long: a position-independent equivalent of JSR, with the range of a 32k in either	PER *ret*-1 BRL *address*

	direction. 6 bytes, 10 clock cycles.	*ret*
BSR *address*	As above, just the branch is short: the range is 128 up or down. 5 bytes, 9 clock cycles (or 10, when the branch has to cross a page boundary).	PER *ret*-1 BRA *address* *ret*
DEW *address* DEW *address*,X	Decrement Word. The Accumulator gets clobbered in the process and NZ flags are left inconsistent, so the assembler will throw warnings because of that. 8 to 11 bytes, zero page min. 11, max. 15 cycles (17 when index is crossing page boundary), absolute min. 13, max. 18 (20 when index is crossing page boundary). When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), DEW is compiled to a single DEC instruction (7 cycles for the zp, 8 for the absolute addressing mode), which leaves the NZ flags both valid afterwards.	LDA *address*... BNE *skip* DEC *address*+1... *skip* DEC *address*... or: DEC *address*...
DWA *address* DWA *address*,X	Decrement Word using Accumulator. Like DEW, but makes explicit the intermediate use of the Accumulator (hence no warning about it being clobbered). Still, the Accumulator is in undefined state afterwards and so are the NZ flags, which only reflect the state of the LSB of the word being decremented. 8 to 11 bytes, zero page min. 11, max. 15 cycles (17 when index is crossing page boundary), absolute min. 13, max. 18 (20 when index is crossing page boundary). When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), DWA is compiled to the LDA/DEC/STA series of instructions (10 cycles for the zp, 12 for the absolute addressing mode), which leaves the NZ flags both valid afterwards, and the result of the decrementation in the Accumulator.	LDA *address*... BNE *skip* DEC *address*+1... *skip* DEC *address*... or: LDA *address*... DEC STA *address*...
Ecc	Exit (= return from) subroutine if the condition "cc" is met. 3 bytes, 8 cycles taken, 3 cycles not taken (or 4, if the pseudo-instruction components cross a page boundary). This pseudo-instruction is convenient when things are about terminating a subroutine prematurely, but one should remember that using a branch to nearest RTS instead of Ecc may produce better code (i.e. saves one byte, although usually takes one or two cycles more).	Bcc *skip* RTS *skip*
ECC	Exit (= return from) subroutine if Carry Clear.	BCS *skip* RTS *skip*
ECS	Exit subroutine if Carry Set.	BCC *skip* RTS *skip*
EEQ	Exit subroutine if EQual.	BNE *skip* RTS

		skip
EGE	Exit subroutine if Greater or Equal. Same as RCS.	BCC *skip* RTS *skip*
ELT	Exit subroutine if Lesser Than. Same as RCC.	BCS *skip* RTS *skip*
EMI	Exit subroutine if MInus.	BPL *skip* RTS *skip*
ENE	Exit subroutine if Not Equal.	BEQ *skip* RTS *skip*
EPL	Exit subroutine if PLus.	BMI *skip* RTS *skip*
EVC	Exit subroutine if V flag clear.	BVS *skip* RTS *skip*
EVS	Exit subroutine if V flag set.	BVC *skip* RTS *skip*
INW *address* INW *address*,X	INcrement Word. Like an INC *address*, but increments a word located at *address* and *address*+1. When the address is on the zero page, occupies 6 bytes and takes 8 to 12 cycles; when outside the zero page, 8 bytes and 9 to 14 cycles. The N flag is not in a consistent state afterwards, Z is. When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), INW is compiled to a single INC instruction, and then the NZ flags are both valid afterwards.	INC *address...* BNE *skip* INC *address...*+1 *skip* or: INC *address...*
Jcc *address*	Jump if the condition "cc" is met. It is an absolute version of conditional branches Bcc with identical meaning, but the jump range of 64k instead of 256 bytes. 5 bytes, 5 cycles taken, 3 cycles not taken (or 4, when the instruction components cross a page boundary).	Bcc *skip* JMP *address* *skip*
JCC *address*	Jump if Carry Clear.	BCS *skip* JMP *address* *skip*
JCS *address*	Jump if Carry Set. As above, with the opposite condition.	BCC *skip* JMP *address* *skip*
JEQ	Jump if EQual.	BNE *skip* JMP *address*

		skip
JGE	Jump if Greater or Equal. Same as JCS.	BCC *skip* JMP *address* *skip*
JLT	Jump if Lesser Than. Same as JCC.	BCS *skip* JMP *address* *skip*
JMI	Jump if MInus.	BPL *skip* JMP *address* *skip*
JNE	Jump if Not Equal.	BEQ *skip* JMP *address* *skip*
JPL	Jump if PLus.	BMI *skip* JMP *address* *skip*
JVC	Jump if V flag Clear.	BVS *skip* JMP *address* *skip*
JVS	Jump if V flag Set.	BVC *skip* JMP *address* *skip*
PHR	Push Registers. Counterpart: PLR. The size and execution time depends on the target CPU and current circumstances: 6502: 5 bytes and 13 cycles; 65C02: 3 bytes and 9 cycles 65C802/816: 3 bytes and * 9 cycles for all registers byte-sized; * 10 cycles for word-sized accumulator; * 11 cycles for word-sized index registers; * 12 cycles for all registers word-sized. *Note that on 6502 there is an unpleasant side effect: the Accumulator contents gets lost - after the PHR's execution A contains a copy of the Y register. The assembler will therefore generate a warning in this case.*	PHA PHX PHY or (for 6502 target): PHA TXA PHA TYA PHA
PLR	Pull Registers. The reverse of the PHR. The size and execution time depends on the target CPU and current circumstances: 6502: 5 bytes and 16 cycles; 65C02: 3 bytes and 12 cycles 65C802/816: 3 bytes and * 12 cycles for all registers byte-sized; * 13 cycles for word-sized accumulator; * 14 cycles for word-sized index registers; * 15 cycles for all registers word-sized.	PLY PLX PLA or (for 6502 target): PLA TAY PLA TAX PLA

Rcc	Repeat previous instructions if condition "cc" is met. This pseudo-instruction comes in two flavours. In its simple form it just follows one instruction which is to be repeated, in this manner: LDA VCOUNT RNE In its more complex form, an entire block of instructions can be repeated. The block should be defined using the directives [and], in this manner: LDY #$00 [LDA $2000,Y STA $3000,Y INY] RNE When the branch is in 8-bit signed range, this pseudo-instruction is compiled as a Bcc, or as a Jcc otherwise. Therefore the resulting object code may accordingly vary in code size and execution time.	*loop ...* Bcc *loop* or: *loop ...* Jcc *loop*
RCC	Repeat instructions if Carry Clear.	*loop ...* BCC *loop* or: *loop ...* BCS *skip* JMP *loop* *skip*
RCS	Repeat instructions if Carry Set.	*loop ...* BCS *loop* or: *loop ...* BCC *skip* JMP *loop* *skip*
REQ	Repeat instructions if EQual.	*loop ...* BEQ *loop* or: *loop ...* BNE *skip* JMP *loop* *skip*
RGE	Repeat instructions if Greater or Equal. Same as RCS.	*loop ...* BCS *loop* or: *loop ...* BCC *skip* JMP *loop* *skip*
RLA	Rotate bits Left in Accumulator. Counterpart: RRA. Unlike in ROL, bit 7 is copied not only to the C flag,	CMP #$80 ROL

	but also to the bit 0. Timings are identical as in ASR.	

RLT	Repeat instructions if Lesser Than. Same as RCC.	*loop* ...
		  BCC *loop*
		or:
		loop ...
		  BCS *skip*
		  JMP *loop*
		skip

RMI	Repeat instructions if MInus.	*loop* ...
		  BMI *loop*
		or:
		loop ...
		  BPL *skip*
		  JMP *loop*
		skip

RNE	Repeat instructions if Not Equal.	*loop* ...
		  BNE *loop*
		or:
		loop ...
		  BEQ *skip*
		  JMP *loop*
		skip

RPL	Repeat instructions if PLus.	*loop* ...
		  BPL *loop*
		or:
		loop ...
		  BMI *skip*
		  JMP *loop*
		skip

RRA	Rotate bits Right in Accumulator. Counterpart: RLA. Unlike in ROR, bit 0 is copied straight into bit 7. 5 to 6 bytes, 5 to 7 cycles.	LSR
		  BCC *skip*
		  ORA #$80
		skip

RVC	Repeat instructions if V flag clear.	*loop* ...
		  BVC *loop*
		or:
		loop ...
		  BVS *skip*
		  JMP *loop*
		skip

RVS	Repeat instructions if V flag set.	*loop* ...
		  BVS *loop*
		or:
		loop ...
		  BVC *skip*
		  JMP *loop*
		skip

Scc	Skip following instruction if condition "cc" is met.	Bcc *skip*

	This pseudo-instruction precedes the instruction which is to be skipped, in this manner: INC ADR SNE INC ADR+1 In the more complex form the [and] may be used to define the block to skip: LDA $2000 SEQ [LDY #$00 [LDA $2000,Y STA $3000,Y INY] RNE] *Remark: in the current implementation no global labels may be defined within the scope of this pseudo-instruction. For example, the following:* *SNE* *RESET JMP $E477* *...* *will cause the assembler to throw an error. Also, the Scc pseudo-instruction branch range is 127 bytes only. When the defined block exceeds this range, the assembler will throw an error.*	... *skip*
SCC	Skip instruction if Carry Clear.	BCC *skip* ... *skip*
SCS	Skip instruction if Carry Set.	BCS *skip* ... *skip*
SEQ	Skip instruction if EQual.	BEQ *skip* ... *skip*
SGE	Skip instruction if Greater or Equal. Same as SCS.	BCS *skip* ... *skip*
SLT	Skip instruction if Lesser Than. Same as SCC.	BCC *skip* ... *skip*
SMI	Skip instruction if MInus.	BMI *skip* ... *skip*
SNE	Skip instruction if Not Equal.	BNE *skip* ... *skip*

SPL	Skip instruction if PLus.	BPL *skip* ... *skip*
SVC	Skip instruction if V flag Clear.	BVC *skip* ... *skip*
SVS	Skip instruction if V flag Set.	BVS *skip* ... *skip*
SUB	Subtract without carry. The same addressing modes are available as for the SBC, this pseudo-instruction is just 1 byte longer and takes 2 cycles more. Counterpart: ADD.	SEC SBC ...

XII. Instruction aliases

An alias is just an alternative mnemonic for an instruction. ELSA implements a bunch of these following the CPU producer's advice.

Syntax	Synopsis	Equivalent to
BGE *address*	Branch if Greater or Equal.	BCS *address*
BLT *address*	Branch if Lesser Than.	BCC *address*
CLR *address* CLR *address,*X	Clear the specified memory location.	STZ *address* STZ *address,*X
CPA ...	Compare with the Accumulator.	CMP ...
DEA	Decrement Accumulator.	DEC
HLT	Halt the processor.	STP
INA	Increment Accumulator.	INC
LSL ...	Logical Shift Left	ASL ...
PEI (*address*)	Push Effective address, Indirect (move word from ZP to stack)	PEA (*address*)
PER *address*	Push Effective address, Relative	PEA *address*
SWA	SWap Accumulator halves.	XBA
TAD	Transfer Accumulator to Direct page register.	TCD
TAS	Transfer Accumulator to Stack pointer.	TCS
TDA	Transfer Direct page register to Accumulator.	TDC
TSA	Transfer Stack pointer to Accumulator.	TSC

XIII. Divergences from the WDC-recommended syntax

The main divergence from the syntax and mnemonic names, which are recommended by the WDC, concerns the PEA instruction. The WDC syntax is this:

PEA $xxxx – PEA absolute
PEI ($xx) – PEA direct page indirect
PER $xxxx – PEA relative

But this "PEA absolute" simply pushes its 16-bit argument value onto the stack, so you could think that naming it (the argument) "absolute effective address", especially in a machine where effective absolute addresses are 24-bit, is quite an overstatement. Sure, we write JMP $xxxx, and speak of the instruction as being in absolute addressing mode, but JMP actually *uses* its argument as *an address* to change the current location of the PC within the code. If we were thinking of JMP as of a 16-bit move (which it technically is), we could symbolically write it down as MOVE #$xxxx,PC – and yes, in *this* context, *with* the hash.

So, ELSA (and some other assemblers) are treating the first instance of PEA as being in immediate mode. Therefore the syntax is as follows:

ELSA syntax	WDC syntax
PEA #$xxxx	PEA $xxxx
PEA ($xx)	PEI ($xx)
PEA $xxxx	PER $xxxx

As hinted in the previous section, you can still use PEI ($xx) and PER $xxxx besides PEA ($xx) and PEA $xxxx, respectively.

XIV. MAE's bugs

There are several known bugs in MAE's compiler, here is how ELSA will behave in the same circumstances:

Code	Problem	MAE's behaviour	ELSA's behaviour
SOMELONGLABEL012 = 1	Label longer than 15 characters.	Likely crash.	Labels up to 245 characters are allowed.
LDA ($1234),Y	No such addressing mode.	Silently accepted as LDA ($34),Y	Accepted with a warning as LDA ($34),Y
LDX $800000	No such addressing mode.	Silently accepted as LDX $00	Error, improper addressing mode.
LDA #$<1234	Nonsense syntax.	Silently accepted as LDA #$01	Error, bad constant.
AA = BB BB = CC CC = 1 .ORG $0600 LDA AA	AA undefined during second pass.	Silently accepted, LDA AA compiled as LDA 32768	Error, undefined label.

.LONG 0*2	None apparent.	Compiled as .LONG $2A0000	Compiled as .LONG $000000
.WORD -256	None apparent.	Compiled as .WORD $FE00 (= -512)	Compiled as .WORD $FF00