Lima

A structured macroassembler by Paul Robson September 2022

Lima is the fourth version of my structured shorthand macroassembler concept, and hopefully the final one.

Lima sees the 6502 as a two register device. The main register is either A or XA. XA is known as R. The other register is the Y register, which is used in indexing.

Lima is at heart a simple pattern matching code generator.

Lines of code are compared against a dictionary of such, and each of those lines has a code generator attached to them. Some ‘code generators’ are code themselves, executed to do specific actions, most just copy the code out.

The idea came from looking at the PC structured assembler TERSE – this is not the same, or a clone, but it has similar ideas.

One can make an argument that this is just syntactic sugar on 6502 assembly ; it is slightly more than that, in that it provides 16 bit instructions. But yes, that’s pretty much it.

Operation

The Lima code is stored text files. These are translated into elements as follows

- An unknown identifier – this can be a keyword or an as yet undefined identifier. Identifiers are built from A-Z 0-9 $ % and . Characters. They are converted to upper case – lower case indicates a 'wild card'.
- A byte or word constant – can be decimal or hexadecimal. Constants can be postfixed with a minus sign.
- A known identifier – can be a procedure, an integer variable, or a short variable.
- A punctuation character.

These are converted into elements, each one of which has the top bit of its last character set. Procedures, Integer Variables, Short Variables and Constants are converted into lower case equivalents. These are as follows :-

Type	**Letter**
Procedure	p
Byte Constant	b
Word Constant	w
Zero Page Byte Variable	s
Zero Page Word Variable	i
Absolute Byte Variable	c
Absolute Word Variable	l

A seperate table is kept of what these refer to , because replacing a known identifier with a single character loses information.

Note that things that are defined on a line do not become visible till the next code line – so you cannot define and use a variable in the same line. At present this does not work.

The way Lima works is that this is then compared against a dictionary of matches, stored in backwards order (so longest first).

Each match can only have one 'wild card' - the lower case letter in the previous table in it.

Matches

A match consists of three parts. The wildcards that are matched, the match itself, and the code it generates. These are generated from text files, where the wildcard(s) and the match are on the first line, and the generated code is on subsequent lines, indented, so it looks a bit like Python, assembler elements can be seperated by semicolons.

For example, this Lima macro loads a variable into word register R (note case does not matter)

```
SI      R = <var>
                lda     <addr> ; lda <next>
```

The first part is the SI. This indicates that the definition is for short variables and integer variables. In this particular instance, this is actually *four* definitions. This is because SI does not produce just a match for S and I , it also produces one for L and C – this will match.

R=s   R=i   R=l   R=c

This is because you want different code generation for absolute addresses (c and l) and zero page addresses (s and I) – in this example it would generate 6 bytes of code for absolute and 4 for zero page.

Note this doesn't cover byte loads into A, which would require a second group of four.

```
SI      A = <var>
                lda     <addr>
```

<var> <proc> and <const> are used in similar ways, they are all shorthand for the '*' - the character that is matched is determined by the match on the left. It just makes it more readable. So you could have

```
P       <proc>()
                jsr     <addr>
BW      R = <const>
                lda     #<low> ; ldx # <high>
```

or

```
P       *()
                jsr     <addr>
```

* is also available as a left hand wildcard, which means 'don't care'

Code Generation

The code generated is produced by 64tass by replacing the <elements> with code equivalents ; these are placeholders for actual values. Thus 64tass figures out the differences, which is why Lima is so small – it dodges most of the work.

The substitution is as given below. The $63 $73 $83 markers are replaced by data when code is generated.

Marker	Purpose	Values used
<addr>	Address of variable	$6373 in absolute mode, $63 in zero page mode.
<next>	Address of variable + 1	$8373 in absolute mode, $83 in zero page mode.
<low>	Low byte of data	$63
<high>	High byte of data	$73
[execute:xxx]	Execute code	$A3 nn where nn is a constant relating to label Action_xxx , generated automatically.
[data]	Set data variable	$93 ll hh sets data variable to llhh
[iszero]	Checking	$53 Errors if the data matched is nonzero

[execute:xxx] is used to run code, so proc (procedure definition) is defined as

```
*       proc
                [execute:define_proc]
```

This calls a code routine “Action_define_proc” which does the actual work – extracting the identifier, checking the parameter brackets etc. Similar code is used for structures and variable definition.

[data] is used primarily in conditionals. Conditional tests are matches on their own e.g. R>=0? checks if R >= 0. The structures – if and repeat – use these tests. The data is used to pass forward which branch to use (in this case, it would probably be the opcode for bpl), so this might be written as :

```
*       r>=<const>
                cpx # 0                 // high byte of XA
                [iszero]                // constant must be zero
                [data] bpl *            // pass information forward.
```

Dictionary Storage

There are two dictionaries, system and user. The system dictionary contains almost exclusively matches, and the user dictionary contains procedures and variables. It is not possible to have matchers in the user dictionary.

The structure of those dictionaries are:

Offset	Contents
+0	Offset to next, or zero. Normally the length of the 'name' + 6
+1	Type Byte, This is P(rocedure) M(atch) I(nteger) or S(hort)
+2,+3	Address of variable or procedure, code generator address of match (this is a count byte followed by code with the $x3 markers)
+4	Bank number for procedure, 0 otherwise
+5	Length of matching name
+6…	Name – this can be a sequence of one or more elements.

By convention, local variables and procedures begin with an dot character. There is an action which removes all that do not fit this from the dictionary "remove_locals"