XA80 Macro Assembler Technical Documentation

Duncan Munro

June 2022

Contents

[1 High Level Design 3](#_Toc110348381)

[1.1 Software purpose 3](#_Toc110348382)

[1.2 Key specifications 3](#_Toc110348383)

[1.3 General approach 3](#_Toc110348384)

[1.4 Files used 4](#_Toc110348385)

[2 Detail Design 5](#_Toc110348386)

[2.1 Source Format 5](#_Toc110348387)

[2.2 Source Processing 5](#_Toc110348388)

[2.2.1 Initial Processing 5](#_Toc110348389)

[2.2.2 Label Processing 5](#_Toc110348390)

[2.2.3 Keyword / Opcode Processing 6](#_Toc110348391)

[2.2.4 Operand Processing – Instructions 6](#_Toc110348392)

[2.2.5 Operand Processing - Directives 7](#_Toc110348393)

[2.3 Macro processing 7](#_Toc110348394)

[2.3.1 Macro recursion 7](#_Toc110348395)

[2.3.2 Replacement of local labels and parameters 7](#_Toc110348396)

[3 Data Types and Structures 9](#_Toc110348397)

[3.1 Preprocessor 9](#_Toc110348398)

[3.1.1 Optimise Comments 9](#_Toc110348399)

[3.1.2 Optimise Keywords 10](#_Toc110348400)

[3.1.3 Optimise Labels 10](#_Toc110348401)

[3.1.4 Optimise Operands 10](#_Toc110348402)

[3.1.5 Optimise Indirection 11](#_Toc110348403)

[3.2 LaCoGen grammar – Operands 11](#_Toc110348404)

[3.3 Opcode compiler 12](#_Toc110348405)

[4 File Formats 13](#_Toc110348406)

[4.1 Debug File Format 13](#_Toc110348407)

[4.2 Debug File Line Detail 13](#_Toc110348408)

# High Level Design

This main section details the high level design of the assembler.

## Software purpose

The purpose of the software is to take a source file containing assembly language instructions for the x80[[1]](#footnote-1) series processors and assemble this into an object file which can be executed in an appropriate processor environment.

## Key specifications

Some of the main functional features of the assembler are:

* Source capabilities
  + Compatibility with original Intel 8080 assembler as well as 8085, Z80, Z180
  + Macro capability with nested macro execution
  + Include file capability
  + Conditional assembly using if / then / else
* Advanced features
  + Full expression evaluation
  + Comprehensive error checking
  + Ability to specify different members of the processor family at assembly time
* Output capabilities
  + Object files which can be used with a linker
  + Binary images which can be used directly
  + Map files showing all defined labels
  + Hex files
  + Listing files
  + Debug information (file, line number, address)

## General approach

The assembly is conducted in two passes in order to deal with include files, macro expansions, variable definitions and code generation. The two passes are:

* 1 – Initial pass
* 2 – Final pass

The initial pass will resolve all variable address, preparing for the final pass which carries out the assembly and produces the output files.

## Files used

The following files are used and/or generated by the assembler:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Contents | Ext | Attributes |
| Source | Source code to be assembled | .asm  .inc | Input  Human readable |
| Debug | Records that can be used by an external debugging tool | .dbg80 | Permanent  Optional  Not readable |
| Object | Code records produced by the assembler in a binary form | .obj80 | Permanent  Not readable |
| Hex | Code records produced by the assembler in a text form compatible with the Intel Hex format | .hex | Permanent  Human readable |
| Listing | Listing in human readable form which can be reviewed or printed | .lst | Permanent  Optional  Human readable |
| Log | Details of progress, warnings and errors stemming from the assembly | .log | Permanent  Optional  Human readable |

# Detail Design

## Source Format

The source files are in the format:

[label[:]] [opcode [operand1[,operand2]]] [comment]

[label[:]] [directive [operands]] [comment]

A trailing colon character on each label is optional. Opcodes and directives must start after an amount of whitespace.

There is one known ambiguous situation which is the 8080 assembler SET directive and the Z80 SET opcode (8080 does not have a bit SET instruction), specifically:

START SET $200 ; Set or change label START to $200

START SET 2,B ; Set bit 2 of B

In the above examples, the first line is a directive and the second line is an instruction. XA80 resolves this by testing for two operands to the set command and if found, processes as an instruction.

## Source Processing

Initial source processing consists of splitting the source line into four parts:

* Label
* Directive or opcode
* Operands
* Comments

In general, a preparser is used to split the input line up which is then followed by the main parser which can carry out more complex activities such as expression evaluation.

### Initial Processing

Lines which are empty are not processed any further, other than to include in the listing file if permitted. Active lines have tabs expanded to a default of 4 spaces, however this can be amended on the command line.

### Label Processing

Labels will generally start at character position 1 on the line although this is not mandated. XA80 can figure out what is a label by checking for keywords such as LD, MVI, DEFB etc. and using the context to decide what is a label.

Labels can start with an alphabetic character or underscore. The remainder of the label may also use these characters along with digits. Some examples of valid labels are:

START

\_\_FDECL\_\_

Rec105

Labels are not normally case sensitive unless a command line switch is used to effect this.

### Keyword / Opcode Processing

XA80 must decide if an entry in the next position is one of the following four options:

* A macro reference, e.g. MOVE\_FPACC
* An ambiguous opcode/directive, specifically SET
* A processor opcode, e.g. LD, XCHG
* A directive, e.g. INCLUDE, DB
* Not one of the above, so an error

Processor opcodes will vary by processor type and are defined in the opcode map file. More details of this are given in section @@@@@.

Directives are static and will always be the same list.

Macro references will always be defined before use so can be looked up from a list of defined macros.

If none of the above lists match, a terminal error condition is displayed.

### Operand Processing – Instructions

Operand and comment processing for instructions is handled by a mini parser first. Operands can be simple operands such as 12, or [HL], or may be more complex operands such as (IX+rec\_offset-3).

XA80 tries to pull out the simple operands first, failing that it invokes a more advanced parser to resolve the more complex operands.

Note that the label has been removed, also the opcode itself – this is known to be LD by the calling software and does not need to be sent to the parser. All the parser needs to know is there in an opcode and two operands.

Once it returns, XA80 can then determine if the supplied operands are suitable for the opcode by checking against the opcode map.

The operands can be one of the following items:

* A simple operand, e.g A, (HL)
* A complex operand with an expression, e.g. (IX+ZOFFSET)
* An integer expression, e.g. BUFPTR, $-2
* A string expression, e.g. ‘HELLO’

### Operand Processing - Directives

For directives, the remainder of the line, all operands and any comments, is processed en masse by the LaCoGen lexer and parser.

Some examples of source and what they translate to are:

START IFDEF DEBUG ; Compile if debug mode

Welcome DB ‘Welcome’,0 ; Welcome message

Translate to the following strings for the parser:

IFDEF DEBUG ; Compile if debug mode

DB ‘Welcome’,0 ; Welcome message

## Macro processing

Macro processing involves the following actions:

* Recursive processing of macro within a macro
* Replacement of labels with local labels
* Replacement of called parameters

### Macro recursion

A macro stack is used to support recursion.

Checks are in place to ensure a macro does not contain a macro that is already being processed as this would result in an infinite loop.

Each macro expansion instances a new local prefix which is stored on the macro stack.

### Replacement of local labels and parameters

All labels defined in a macro must be stored and used as local labels. For instance the following macro definitions:

COPY\_B MACRO dst,src ; Copy B bytes from [HL] to [DE]

LD HL,src

LD DE,dst

CPYC\_LP LD A,[HL] ; Could use LDIR but this is just

LD [DE],A ; an example

INC DE

INC HL

DJNZ CPYC\_LP

ENDM

COPY\_4 MACRO dst,src ; Copy 4 bytes from [HL] to [DE]

LD B,4

COPY\_B dst,src

ENDM

FPCPY12 COPY\_4 FPACC2,FPACC1

ENDM

When expanded with the call:

FPCPY12

Will yield:

>>> FPCPY12

>>> COPY\_4 FPACC2,FPACC1

LD C,4

>>> COPY\_C FPACC2,FPACC1

LD HL,FPACC2

LD DE,FPACC1

@0001@CPYC\_LP LD A,[HL] ; Could use LDIR but this is just

LD [DE],A ; an example

INC DE

INC HL

DJNZ @0001@CPYC\_LP

ENDM

# Data Types and Structures

## Preprocessor

The LaCoGen grammar for the preprocessor (xa80pre.lac) is a lightweight grammar which allows the input to be tokenised into:

* String constants
* Commas
* “Something” else

For example the following input:

HERE: DB 27,’Test’ ; Our definitions

Would be processed into:

|  |  |
| --- | --- |
| HERE: | Something |
| DB | Something |
| 27 | Something |
| , | Comma |
| ‘Test’ | String |
| ; | Something |
| Our | Something |
| definitions | Something |

This then goes through five stages of optimisation to further refine the analysis. These are discussed in sections 3.1.1 to 3.1.5. These will further expand the Something/Comma/String options to:

* Comma
* **Comment**
* **Keyword**
* **Label**
* **Operand**
* Something
* String

At the end of the process there should not be any Something or String entries, they should have been converted into one of the bold categories shown above.

### Optimise Comments

The first step in optimisation is to optimise the comments. In our example, the following section:

|  |  |
| --- | --- |
| ; | Something |
| Our | Something |
| definitions | Something |

Is merged into one section marked Comment. Anything with the semicolon or // will become a comment as will anything following it. A \* character in the first position of a line will also force a comment situation.

### Optimise Keywords

The next step is to analyse the inputs looking for ‘Something’ entries and matching them against a keyword table. The keyword table consists of directives such as DEFB, INCLUDE, ORG, EQU and opcodes such as LD, MVI, PUSH. The opcodes will vary depending on the processor selected as, for example, Z80 has very different opcodes from 8080.

In our original example:

|  |  |
| --- | --- |
| DB | Something |

The entry DB will be picked up as a keyword and the entry will be marked as such.

### Optimise Labels

If a parsed line starts with a ‘Something’ after the keyword optimisation has been carried out, we can safely assume this is a label. It may or may not have a : character on the end. If one exists, it is stripped out.

Much like the keywords routine, the entry will be marked as a label. Checks are applied to ensure the line starts with one of:

* A label on its own or followed by a comment
* A label followed by a keyword
* A keyword

Anything else would represent an error.

### Optimise Operands

Optimisation of the operands consists of grouping the items between commas. In our earlier example the ‘Something’ entries already have a one to one relation to operands, however if an input contains:

|  |  |
| --- | --- |
| (IX | Something |
| + | Something |
| ‘E’-‘A’+1) | Something |

All three items will be grouped into one Operand.

### Optimise Indirection

One of the issues with the parser is that it’s possible to use brackets to define grouping within a calculation. For example the expression 1+4\*3 would result in 4\*3 being reduced to 12 and the final calculation would be 1+13 = 12.

The operand parser will allow (1+4)\*3 which gives a different result of 15.

It’s the use of brackets that becomes a problem. For example, Z80 code uses brackets to indicate indirection, for example LD A,(HL) or LD A,(0x200).

The parser will try to reduce the (HL) to a define called HL which may not even exist. Likewise, the (0x200) will be reduced to 0x200 which will cause an error as you cannot load a 16 bit value into A. In either instance, the result is not as intended.

The final step for the preprocessor is to intelligently check the operand and replace any outer enclosing brackets with square brackets, for example (HL) 🡪 [HL] and (0x200) 🡪 [0x200].

It’s not just a given that having outer brackets represents indirection, consider the operand (2+2)\*(1+1). While this has outer brackets, it is not an indirection and should stay as it is.

Having gone through five steps of optimisation, our original example will now look like this:

|  |  |
| --- | --- |
| HERE | Label |
| DB | Keyword |
| 27 | Operand |
| , | Comma |
| ‘Test’ | Operand |
| ; Our definitions | Comment |

## LaCoGen grammar – Operands

A somewhat heavier LaCoGen grammar is used for the operand processing. This positions the operand as one of the 10 following types:

1. IX on its own
2. IY on its own
3. A simple operand other than IX, IY, for example: HL, A, C, PSW, M
4. [IX + <expression>]
5. [IY + <expression>]
6. [IX]
7. [IY]
8. [ <simple operand> ] e.g. [HL], [C]
9. [ <expression> ] e.g. [0x200]
10. <expression>

A full expression evaluator is built into the grammar. Examples for <expression> could be:

* BASE\_ADDR >= 0x1000
* (5+2) \* 7
* (INVAL | BIT\_MASK) & 0xFF

Once the keyword for the opcode and operand type(s) are known, the correct instruction can be selected from the list produced by the Opcode Compiler.

## Opcode compiler

The Opcode Compiler takes a list of opcodes, operands and binary outputs to create a list that can be used by the assembler to generate code output. The following example from 8080.opcode illustrates what each entry looks like:

CMP M | %10111110

CMP A | %10111111

ANI U8 | %11100110 [1:U8]

XRI U8 | %11101110 [1:U8]

CNZ U16 | %11000100 [1:U16]

The input consists of an opcode, zero or more operands and a binary representation to the right of the ‘|’ character. The instructions at the top of each file document all the options.

This input file is then compiled by the oc\_comp utility to turn the 8080.opcode file into 8080.opcode.bin which is loaded into the assembler.

# File Formats

## Debug File Format

The debug file contains source lines and object code which can be used at a later stage by a debugging tool.

The general format is:

Line detail 1

Line detail 2

Line detail 3

: :

This is a binary format.

## Debug File Line Detail

Each debug file line contains the following information in a binary format

16 bit code address, or $FFFF if no code at this address

16 bit code output length indicator

Sequence of bytes output by this line

16 bit string length indicator

String as a sequence of bytes

This continues until the end of the file is reached

Index

Code generation, 3

Conditional assembly, 3

Else, 3

Hex, 4

If, 3

Include file, 3

Include files, 3

Listing, 4

Log, 4

Macro, 1, 3

Object, 4

oc\_comp, 12

Opcode Compiler, 12

Preprocessing, 3

Source, 4

Specifications, 3

Then, 3

1. x80 collectively refers to the 8080 microprocessor and derivatives such as the 8085, Z80, Z180 [↑](#footnote-ref-1)