XA80 Macro Assembler Technical Documentation

Duncan Munro

June 2022

Contents

[1 High Level Design 3](#_Toc107909758)

[1.1 Software purpose 3](#_Toc107909759)

[1.2 Key specifications 3](#_Toc107909760)

[1.3 General approach 3](#_Toc107909761)

[1.4 Files used 4](#_Toc107909762)

[2 Detail Design 5](#_Toc107909763)

[2.1 Source Format 5](#_Toc107909764)

[2.2 Source Processing 5](#_Toc107909765)

[2.2.1 Initial Processing 5](#_Toc107909766)

[2.2.2 Label Processing 5](#_Toc107909767)

[2.2.3 Directive / Opcode Processing 6](#_Toc107909768)

[2.2.4 Operand / Comment Processing – Instructions 6](#_Toc107909769)

[2.2.5 Operand / Comment Processing - Directives 7](#_Toc107909770)

[2.3 Pre-parser state machine 7](#_Toc107909771)

[2.3.1 Pre-parser states 7](#_Toc107909772)

[2.3.2 Pre-parser actions 8](#_Toc107909773)

[3 Technical Information 11](#_Toc107909774)

[4 File Formats 12](#_Toc107909775)

[4.1 Debug File Format 12](#_Toc107909776)

[4.2 Debug File Line Detail 12](#_Toc107909777)

# 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
  + Macro capability with nested macros
  + 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
  + Executable 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 | .dbg | Permanent  Optional  Not readable |
| Object | Code records produced by the assembler in a binary form | .obj | Permanent  Not readable |
| Hex | Code records produced by the assembler in a text form | .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]

If label is specified it must start on the first character of the line. A trailing colon character 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.

There is an option to force colons for labels, in this instance the label does not have to start at the beginning of the line.

## Source Processing

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

* Label
* Directive or opcode
* Operands and 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 always start at character position 1 on the line. Any whitespace in this position indicates that a label is not used.

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

### Directive / 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 / Comment Processing – Instructions

Operand and comment processing for instructions is handled by a mini parser first. This splits the input into the opcode and up to two operands as pure text.

For example, LD A,(HL) is split into an opcode and two operands:

* LD
* A
* [HL]

XA80 counts the number of operands for instructions and makes a note of any that are totally enclosed in parenthesis. This is necessary as the expression evaluator in the parser will turn ($20) into $20 so it will be impossible to know if this is a memory address or an immediate value. The cure is to replace the outer parenthesis with square brackets.

These are then passed to the main parser as an opcode placeholder along with the operands. Comments are removed by the pre-parser.

The source line:

START LD A,(HL) ; Grab byte

Translates to the following strings for the main parser:

@OPCODE@2@ A,[HL] ; Grab byte

The opcode placeholder can be @OPC0@, @OPC1@ or @OPC2@ depending on the number of 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 / Comment 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

## Pre-parser state machine

The pre-parser splits the input up as described in sections 2.2.1 to 2.2.5 above. It assumes the following has already been done:

* Blank lines are eliminated and will not be sent to the pre-parser
* Lines starting with \* are eliminated as comment lines
* Tabs are expanded out to spaces
* An EOL (end of line) character has been appended to the input

### Pre-parser states

The basis for the pre-parser is a state machine which switches state depending on the the input. The state table is as follows:

|  |  |
| --- | --- |
| State | Description |
| stStart | Start position when no input is known |
| stWhitespace1 | In whitespace between label and directive. This will only be one or more ASCII $20 space as the tab characters will have been dealt with already |
| stWhitespace2 | In whitespace between directive and operand |
| stLabelStart | First character in a label which can be A..Z,a..z or \_ underscore |
| stLabelRemain | Remaining characters in a label which can be A..Z,a..z,0..9 or \_ underscore |
| stLabelColon | State if the optional colon character is found in the input |
| stDirect | Directive state which can be any characters in the range A..Z,a..z |
| stOperand | Operand state which can be any of the printable characters including spaces but excluding commas. Text enclosed in ‘single’ or “double” quotes is dealt with, as are the escape characters which are permitted within the text strings |
| stOperandSQuote | Inside single quoted text |
| stOperandDQuote | Inside double quoted text |
| stComma | Comma state which separates the operands |
| stComment | Inside a comment |
| stEOL | End Of Line |

### Pre-parser actions

Actions from the pre-parser can be to:

* Ignore an input, for example in the case of an optional colon
* Continue with input, for example further characters being added to a label
* Switch state and accept the input

Consider the following input line:

START: XOR A,A ; Zero acc

The following sequence will be used for the input:

|  |  |  |  |
| --- | --- | --- | --- |
| State | Peek | Actions | Buffer |
| stStart | S | Leave stStart  Change to stLabelStart  Move S into buffer | S |
| stLabelStart | T | Leave stLabelStart  Change to stLabelRemain  Move T into buffer | ST |
| stLabelRemain | A | No state change  Move A into buffer | STA |
| stLabelRemain | R | No state change  Move R into buffer | STAR |
| stLabelRemain | T | No state change  Move T into buffer | START |
| stLabelRemain | : | Leave stLabelRemain[[2]](#footnote-2)  Change to stLabelColon  Move : into buffer | : |
| stLabelColon | [space] | Leave stLabelColon  Change to stWhitespace1  Move [space] into buffer | [space] |
| stWhitespace1 | [space] | No state change  Move [space] into buffer | [2 x space] |
| stWhitespace1 | [space] | No state change  Move [space] into buffer | [3 x space] |
| stWhitespace1 | X | Leave stWhitespace1  Change to stDirect  Move X into buffer | X |
| stDirect | O | No state change  Move O into buffer | XO |
| stDirect | R | No state change  Move R into buffer | XOR |
| stDirect | [space] | Leave stDirect[[3]](#footnote-3)  Change to stWhitespace2  Move [space] into buffer | [space] |
| stWhitespace2 | A | Leave stWhitespace2  Change to stOperand  Move A into buffer | A |
| stOperand | , | Leave stOperand  Change to stComma  Move , into buffer | , |
| stComma | A | Leave stComma  Change to stOperand  Move A into buffer | A |
| stOperand | [space] | No state change  Move [space] into buffer | A[space] |
| stOperand | ; | Leave stOperand  Change to stComment  Move ; into buffer | ; |
| stComment | [space] | No state change  Move [space] into buffer | ; [space] |
| stComment | Z | No state change  Move Z into buffer | ; Z |
| stComment | e | No state change  Move e into buffer | ; Ze |
| : | : | : | : |
| stComment | c | No state change  Move c into buffer | ; Zero acc |
| stComment | [eol][[4]](#footnote-4) | Leave stComment  Change state to stEOL  Move EOL into buffer[[5]](#footnote-5)  Terminate the activity | [eol] |

Leaving one state will trigger an on-leave routine if appropriate, joining a new state will trigger an on-enter routine if appropriate.

## 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

# Technical Information

# 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, 4

Conditional assembly, 3

Else, 3

Expressions

evaluation, 4

File format, 4

Hex, 4

If, 3

Include file, 3

Include files, 3, 4

Interim file, 3

Intermediate, 4, 8

Listing, 4

Log, 4

Macro, 1, 3

expansion, 4

Object, 4

Parser, 4

Preprocessing, 2, 3, 4

Source, 4, 8

Specifications, 3

Then, 3

1. x80 collectively refers to the 8080 microprocessor and derivatives such as the 8085, Z80, Z180 [↑](#footnote-ref-1)
2. This will trigger saving the label and leaving a state will always empty the buffer after leave processing is completed [↑](#footnote-ref-2)
3. This will trigger saving of the directive [↑](#footnote-ref-3)
4. End Of Line character ASCII 10 / LF [↑](#footnote-ref-4)
5. Not necessary, however it’s part of the way all other inputs are treated [↑](#footnote-ref-5)