PlasMa

aka Phil's lights and switches Machine

Mini-Mainframe

Simulator Project

Plasm

Assembler Manual

Phil Tipping

www.philizound.co.uk

Table of Contents

1. Introduction

This document describes a program for assembling PlasMa programs offline. It generates hex and paper-tape files suitable for loading into the PlasMa machine and the PlasMaSim simulator. Full details and user manuals for these are on the philizound.co.uk website.

The full version of the assembler program is a single executable file named Plasm3.exe. This can assemble source code for all 3 instruction sets used on the PlasMa machine, namely Toy-A, Toy-B and Advanced. A Directive within the source file identifies the source type.

Limited versions are also available which can only assemble subsets of these. Plasm1.exe can only assemble source code for microcode 1 (Toy-A), and Plasm2.exe can only assemble source for microcodes 1 and 2 (Toy-A and Toy-B).

2. Acknowledgements

The Toy instruction set is used by kind permission of Robert Sedgewick and Kevin Wayne at Princeton university, and is described in their book 'Computer Science'. More details at:

https://introcs.cs.princeton.edu/java/home and https://introcs.cs.princeton.edu/java/60machine

Coursera course site: https://www.coursera.org/learn/cs-algorithms-theory-machines

Thanks also to Adrian Rawson for suggestions and testing.

3. Documentation

These following documents are downloadable from the philizound.co.uk website or by contacting me directly (email address is at the bottom of the website page).

- PlasMa Machine Manual
- PlasMa Instruction Set - Toy-A
- PlasMa Instruction Set - Toy-B
- PlasMa Instruction Set - Advanced
- PlasMaSim Simulator Manual
- Plasm Assembler Manual

4. Installation and Running

The assembler is a console application, and runs inside a command window under Microsoft Windows on a PC. There is no 'installation' as such; just move the relevant Plasm executable file into a folder of your choice. The program creates various files in this folder, so the 'Program Files' folder is not recommended as this is usually write protected. One option is to create a PlasMa folder under Documents, and move the file there.

To run the assembler, open a command window[1], navigate to the folder[2] and type/enter Plasm*N* to display the syntax details (where *N* depends on your assembler variant; all variants can assemble Toy-A programs).

The following output files are created: a Listing file, a Hex file and two Paper tape file(s) in 7-hole and 8-hole formats. Tape files can be read by the Toy-B and Advanced emulations.

1 This depends on your version of Windows, e.g. press Windows+R and type ‘cmd’ (without the quotes).
2 Use the ‘cd <folder name>’ command to change directory/folder. The ‘dir’ command displays the folder contents.

5. Source file

This is a plain-text file with a '.pls' extension, e.g. test.pls

You can create/edit this using a simple text editor such as Notepad or Wordpad, but make sure the file has the correct extension after saving (rename if necessary)[3]. If you use a more advanced word processor, the file must be saved in plain-text format otherwise it may contain codes which will cause assembly errors.

Blank lines and leading/trailing spaces/tabs are ignored so the source can be laid out or indented as required.

Label and Equate names are case-sensitive; everything else is case-insensitive.

Examples in this document (apart from the last two) are for demonstrating source code syntax only; they should all assemble and generate a listing file, but the programs won't do anything sensible if loaded into the machine or simulator.

5.1 Syntax overview

Each non-blank line can be any of the following, optionally followed by a Comment:

- a Directive which tells the assembler how to interpret the following lines.
  The first non-comment line in the source must be a Source code directive.
- a Label definition, optionally followed by an Instruction or a Number.
- an Equate definition.
- an Instruction.
- a Number.

Directives, Label and Equate definitions are *not* allocated any space in memory.

Everything else is interpreted according to the current Directive as either an Instruction or a Number. These are assembled into 16-bit values and allocated to memory at the 'current memory address', which starts at zero and is automatically incremented. The address can be overridden at any point with a Memory address directive.

5.2 Comment

There are two types of comment:

- Line comments are prefixed with a '**;**' character. The assembler ignores all text from this point until the end of the line.
- Block comments are prefixed with a '**{**' character. The assembler ignores all text from this point until an end delimiter '**}**' is found. This could either be within the same line or any other line. Block delimiters must occur as start/end pairs and cannot be nested.

Block delimiters occurring within line comments are ignored and have no effect on an encompassing block comment.

5.3 Number

Numbers can be defined in several ways depending on its prefix.

If the prefix is '$', the number is assumed hexadecimal, so the next 1-4 characters must be hex digits '0' to '9' and 'a' to 'f'.

If the prefix is '+' or '-' or any digit '0' to '9', the number is assumed a decimal integer, so the next 1-5 characters must be decimal digits '0' to '9'.

If the prefix is a single quote, the number is the ASCII code for the next 1-2 characters.

3 Some text editors add a ‘.txt’ extension if you don’t specify one when saving.

Unless otherwise stated, if the prefix is none of the these, the statement is interpreted as the name of a Label or Equate definition, and the number is then defined by the label address or equate value in that definition.

5.4 Directive

These are prefixed with a '**%**' character, and control how subsequent statements are interpreted.

5.4.1 Source code

%s <mcode> defines the type of source code (instruction set) to be assembled.

<mcode> is a Number corresponding to the microcode number used on the PlasMa machine.

The source directive is mandatory and must be the first non-comment line in the source file. It can only be used once.

The remaining directives are optional and can be used multiple times and in any order; each directive overriding the previous one.

5.4.2 Memory address

%m <address> sets the 'current memory address' to <address>. The current memory address defaults to zero when the assembly starts.

<address> is interpreted as a Number but cannot be a label or equate name. The value cannot be less than the current memory address.

5.4.3 Code section

%c defines the start of a code section. Subsequent statements will be interpreted as Instructions. This is the default mode when the assembly starts.

Example

```
%s 1            ;source code 1; Toy-A assembly
%c              ;start of code section
  ld r5 $20     ;instruction at addr $00 (default addr if no %m directive)
  add r6 r7 r8  ;instruction at addr $01
  sub r5 r5 r9  ;instruction at addr $02
%m $50          ;set new memory address
  jp r5 $01     ;instruction at addr $50
  hlt           ;instruction at addr $51
```

5.4.4 Data section

%d defines the start of a data section. Subsequent statements will be interpreted as Numbers.

Example

```
%s 1        ;source code 1; Toy-A assembly
%d          ;start of data section
  56        ;value $0038 at addr $00
  $1234     ;value $1234 at addr $01
  'AB       ;ASCII value $4142 at addr $02
%m $50      ;set new memory address
  'C        ;ASCII value $0043 at addr $50
```

5.4.5 Hex section

%h defines the start of a hex section. Subsequent statements will be interpreted as hexadecimal Numbers. Hex '$' prefixes are optional. If absent, they are added to the listing file for clarity.

Hex sections can also contain labels, equates and memory directives, but any values will be interpreted as hex as above.

If you want to copy a .plh hex file into a hex section, just prefix any 'm' address entries with a '%' character to convert them into memory directives (the address value in a hex file is always hex). See the Simulator manual for hex file format details.

Example

```
%s 1      ;source code 1; Toy-A assembly
%m $20    ;set memory address
%h        ;start of hex section
   1234   ;$1234 at addr $20
   AB5    ;$0AB5 at addr $21
   10     ;$0010 at addr $22
```

5.4.6 Text section

%t defines the start of a text section. Subsequent statements will be interpreted as strings of ASCII character codes. There is no limit on the number of characters per line (unlike ASCII codes in Numbers, which are limited to 2). Quote prefixes are optional. If absent, they are added to the listing file for clarity.

Each character is converted into its ASCII code and allocated to the next free byte within the 16-bit memory location (starting with the ms-byte) at the current memory address. The address is incremented after 2 bytes have been allocated, and the process repeats until the end of the string.

Strings can contain any printable characters with the following exceptions.

- They cannot contain spaces or tabs. Spaces can be created using underscores '_' which are converted into spaces when allocated to memory.
- They cannot contain comment delimiters '**{**', '**}**' or '**;**'.
- The first character cannot be '**%**', '**.**' or '**#**' unless a quote prefix is used.

A zero terminator is automatically allocated to the end of each string. If the string has an even number of characters, an extra memory location containing zero is allocated. If odd, the ls-byte of the final memory location is the terminator; see the listing file for clarification.

Text sections can also contain labels, equates and memory directives, but any values will be interpreted as ASCII codes as above (limited to 2 chars max).

Example

```
%s 1          ;source code 1; Toy-A assembly
%m $20        ;set memory address
%t            ;start of text section
  ABCDE       ;ASCII value for 'AB', $4142 at addr $20
              ;ASCII value for 'CD', $4344 at addr $21
              ;ASCII value for 'E', $4500 at addr $22
  ab          ;ASCII value for 'ab', $6162 at addr $23
              ;Terminator automatically added, $0000 at addr $24
  Hi_there    ;ASCII values for 'Hi there' from addr $25 onwards
```

5.5 Label

Label definitions are prefixed with a '**.**' character. They can be defined on any line, either by themselves or shared with an Instruction, Number or Comment. They occupy no memory and can be used to improve program clarity and maintainability, especially if the same address needs to be used multiple times.

.<name> defines a label called <name> with a value equal to the current memory address. There are no limits on the number of label definitions, but each name must be unique and cannot be the same as an Equate name.

<name> is case sensitive and can be up to 16 characters long.

Names can contain any printable characters with the following exceptions.

- They cannot contain spaces, tabs or commas.
- They cannot contain comment delimiters '**{**', '**}**' or '**;**'.
- The first character cannot be '**%**', '**.**', '**#**', '**$**', '**+**', '**-**', single quote, or any digits ‘0’ to ‘9’.

You can use underscore '_' characters or upper/lower-case for readability, e.g. get_status, GetStatus, etc.

Labels can be used/referenced instead of numbers (in most places) by specifying the <name> without the prefix. They can be referenced multiple times within the source file, and do not have to be defined prior to referencing.

Example

```
%s 1            ;source code 1; Toy-A assembly
%m $10          ;set memory address
  ld r5 count   ;addr $10, label reference (defined later), load 8 to r5
.loop           ;label definition, addr $11
  add r6 r7 r8  ;instruction at addr $11
  sub r5 r5 r9  ;instruction at addr $12
  jp r5 loop    ;instruction at addr $13, label reference, jump to addr $11
  hlt           ;instruction at addr $14
%d  ;start of data section
.count 8        ;label definition, data $0008 at addr $15
%t  ;start of text section
.message        ;label definition, addr $16
  Hello_World   ;ASCII codes from addr $16 onwards
```

5.6 Equate

Equate definitions are prefixed with a '**#**' character. They are similar to Labels in that they can be defined on any line, but they must be on lines to themselves (apart from comments). They occupy no memory and can be used to improve program clarity and maintainability, especially if the same value needs to be used multiple times.

#<name> <value> defines an equate called <name> with a value <val>. There are no limits on the number of equate definitions, but each name must be unique and cannot be the same as a Label name.

<name> is case sensitive and can be up to 16 characters long.

Names can contain any printable characters with the following exceptions.

- They cannot contain spaces, tabs or commas.
- They cannot contain comment delimiters '**{**', '**}**' or '**;**'.
- The first character cannot be '**%**', '**.**', '**#**', '**$**', '**+**', '**-**', single quote, or any digits ‘0’ to ‘9’.

You can use underscore '_' characters or upper/lower-case for readability, e.g. delay_time, DelayTime, etc.

<value> is interpreted as a Number but cannot be a label or equate name.

Equates can be used/referenced instead of numbers (in most places) by specifying the <name> without the prefix. They can be referenced multiple times within the source file, and do not have to be defined prior to referencing.

Example

```
%s 1            ;source code 1; Toy-A assembly
#count $34      ;equate definition, equate 'count' with the value $34
  ld r1 $30     ;
  lda r5 count  ;equate reference, load reg 5 with $34
  add r3 r4 r5  ;
```

5.7 Instruction

Instructions are allocated to memory at the current memory address. There can only be one instruction per line, and the current address is incremented after each one.

They can only be used within code sections; see Directive.

They are split into an opcode field followed by none, one or more operand fields using any of the separator characters: space, tab or comma.

Instructions using both source and destination operands follow the field-ordering convention of 'destination first'.

The opcode mnemonics and operand order must correspond to the source code specified in the source Directive[4]. Full syntax details of each instruction are in the corresponding Instruction Set Manuals.

Register fields comprise the prefix 'r' followed by a register number. The register number must always be a single hex digit (0-9, a-f) with no prefix.

Address and other number fields are interpreted as Numbers.

Example

```
%s 1  ;source code 1; Toy-A assembly
      lda r1 9   ;load register 1 with the value 9
      lda rf 'A  ;load register F with the value $41 (char code for 'A')
      st $23 r4  ;store the contents of register 4 into memory address $23
                 ;note operand order is destination first
.loop
      lda r2 $28 ;load register 2 with the value $28
      st $12 r2  ;store the contents of r2 into address $12
      jz r2 loop ;jump if r2 is zero to address defined by label 'loop'
```

6. Listing file

This is a plain-text file created by the assembler with the same root name as the source but with a '.pll' extension, e.g. test.pll

It contains a summary of how the assembler has interpreted the source code. Each source statement is appended with its allocated memory address and contents where applicable. These values are shown in hex, and can be entered manually into the machine or simulator using the Load switches; see Machine and Simulator manuals for details. The file is also useful when single-stepping through the code or for setting break-points.

Equate values are also shown in hex to clarify their interpretation.

Comments from the source are included, although they may disrupt the tabulated layout depending on where they occur.

Error messages are written to the file at or near the failing statement.

4 Although the Plasm assembler can process different instruction sets, you cannot, for example, assemble Toy-A source and run the code on the Advanced machine.

7. Hex file

This is a plain-text file created by the assembler with the same root name as the source but with a '.plh' extension, e.g. test.plh

It contains a list of memory contents in hex, split into contiguous address sections, each one headed by the starting memory address for that section. The file is intended for loading into the simulator via the 'load hex file' command; see the Simulation manual for details.

The hex values can also be entered manually into the machine or simulator using the Load switches; see Machine and Simulator manuals for details.

8. Paper tape file

These are plain-text files created by the assembler with the same root name as the source but with a '.pl7' or '.pl8' extension corresponding to 7 or 8-hole tape formats, e.g. test.pl7

Both files contain a list of memory contents which can be loaded into the real machine or simulator (running Toy-B or Advanced) using a 'loader' program[5], or loaded directly into the simulator using the 'load paper tape' command; see Machine and Simulator manuals for details.

Data bytes written to tape correspond to contiguous memory locations, starting from the first code or data byte allocated to memory, and continuing to the end of the program. Undefined regions between these two limits (which can occur when memory directives are used) cause data bytes with zero values to be written to tape.

There is no address information in these files, unlike the Hex file, so loading a tape requires a start address to be specified or assumed. On the simulator, this is defined by the switches, but on the real machine, the loader program has to do this, e.g. by using I/O instructions to read the switches or keypad etc.

5 Which you'll need to write... but that's all part of the fun.

9. Assembly example - Toy-A

Copy and paste the lines below into a plain text file called 1000.pls in the same folder as the Plasm executable file.

Open a command window and run the assembler: Plasm*N* 1000 (where *N* depends on your assembler variant; all variants can assemble Toy-A programs).

Unless overridden with the command-line switch, the following output files will be created.

The listing file 1000.pll contains hex values for manually loading into the machine or the PlasMaSim simulator via the switches.

The hex file 1000.plh can be loaded into the PlasMaSim simulator using the 'load hex file' command; see Simulator manual for details.

The paper tape files 1000.pl7 and 1000.pl8 can be discarded as Toy-A does not have a Tape Reader peripheral.

```
;add 2 numbers from memory and display result
%s 1            ;Toy-A source code
  ld r1 n1
  st $ff r1     ;display n1
  ld r3 n2
  st $ff r3     ;display n2
  add r5 r1 r3
  st $ff r5     ;display sum
  hlt
%d              ;start of data section
.n1 1234        ;decimal
.n2 5678        ;decimal
```

10. Assembly example - Toy-B

Copy and paste the lines below into a plain text file called 2000.pls in the same folder as the Plasm executable file.

Open a command window and run the assembler: Plasm*N* 2000 (where *N* depends on your assembler variant; it must be 2 or more for Toy-B programs).

Unless overridden with the command-line switch, the following output files will be created.

The listing file 2000.pll contains hex values for manually loading into the machine or the PlasMaSim simulator via the switches.

The hex file 2000.plh can be loaded into the PlasMaSim simulator using the 'load hex file' command; see Simulator manual for details.

The 7 and 8-hole paper-tape files 2000.pl7 and 2000.pl8 can be loaded into the PlasMaSim simulator using the 'load paper tape' command, or loaded into the real machine or simulator using your loader program.

```
;add 2 numbers from memory and display result
%s 2            ;Toy-B source code
  ld r1 n1
  sys $102      ;display n1
  ld r3 n2
  sys $302      ;display n2
  add r5 r1 r3
  sys $502      ;display sum
  hlt
%d              ;start of data section
.n1 1234        ;decimal
.n2 5678        ;decimal
```