MIT Concurrent VLSI Architecture Memo 62

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Guide to the M-Machine Assembler and Massl Assembly Language v1.2

Yevgeny Gurevich

Abstract

This memo outlines the usage and features of version 2.0 of the M-Machine assember, `mas2`, and the format of the massl assembly language. It includes discussion of error reporting and recovery, as well as issues of assembling for and linking multiple object files.

Keywords: M-Machine, Assembler, massl.

1 Overview

This document details the usage and features the M-Machine Assembler (the current executable name is `mas2`) which replaces `Masm`. `mas2` parses, assembles, and links multi-file M-Machine Assembly Language programs into files suitable for execution on the M-Machine simulator, `msim`. Included is an overview of the new input language, which is a hybrid of the multiflow input language and L0, and a discussion of common error messages which users may encounter while using `mas2`.

2 Mas Usage

Mas may be invoked with no arguments to see a usage summary. To assemble and link a list of source files, invoke mas with the names of the input files as the arguments:

```
mas2 file1.m file2.m file3.m . . .
```

Object files with `.o` extensions will be created, and the linker called to generate an executable. An optional `-o<outfile>` may be used to denote an output executable filename. If no output file is specified, file executable is called `a.out`. If a single source file is given, the output is written to `input_file.out` instead of `input_file.o`, and the linker is not called.

To assemble a list of source files without linking, use the `-c` flag. Mas will then create `.o` object files for each `.m` source file, but will not run the linker. If a single file is specified with `-c`, then an optional `-o filename` argument may be used to override the default `input_file.o` object file which is created.

Object files may be linked with the `-l` flag, which treats all files on the command line as object files, instead of massl source files.

For debugging purposes, the `-da` flag may be used to force disassembly of instructions as they are emitted by the assembler. This may be used to check input code against what is being placed into the output object file.

Table 1 summarizes command line options.

2.1 Conventions

All massl programs should end in a `.m` extension. Object files should end in `.o` (the default extension when using mas.)

3 M-Machine Assembly Language

The M-Machine Assembly Language, massl, is highly structured. Each massl program must begin with `text;` and end with `end;`. An optional data segment begins with the keyword `data;` and is active until the `text;` declaration. A BNF description of massl is provided in tables 2 and 3.

3.1 Massl instructions

All massl instructions start with the token word `instr`, contain up to three operations (each from a different functional unit) and end with a semicolon. All operands to an operation (except the optional condition/condition-register pair) are comma-delimited and denote sources first, then destinations (i.e. src1,

Option	Description
-o filename	Sets the name of the output object file to *filename* instead of input_file.out, if a single file is being assembled. Sets the name of the executable, when multiple files are given on the command line. May not be used with -c option.
-c file_list	Assemble multiple .m files into .o files (each .m file is assembled to a separate .o file.) The resulting files may be linked together to form an executable .out file at a later time (see -l option.)
-l file_list	link multiple .o files to form an executable. mas2 will call the M-Machine linker to resolve symbols and link the .o files. The .o file with the first operation to be executed **must** be listed first if no `Main` global label is specified. If no file name is specified with -o, a.out is used as the resulting executable name.
-da	Show disassembly of the operations as they are being emitted to object file(s).
-dt	Show values of symbol-table entries (useful for checking how large certain labels are in order to fit them into limited 7- 11- and 18-bit operation slots.)
-D	Include debugging information in the resulting object file. (default)

Table 1: mas2 command line options

src2, src3, destination). pre and post conditions to operations which require them precede all other operands. Bank and Length operands for operations which require them precede all other operands. Operations which have both integer/floating-point and conditional destinations have the int/fp destinations specified first, followed by the cc-regs. See the ISA document for detailed descriptions of M-Machine operations. Examples of certain valid operations are given in figure 2.

Integer literals are any decimal numbers (or hexadecimals preceeded by an `0x`.) Because of the way mas performs parsing, integer literals which are defined larger than may be represented in 64 bits have their significant bits truncated, resulting in incorrect constants. There are no warning messages accompanying this truncation.

Identifiers may be any alphanumeric string which is not a massl keyword (see table 4.) Although not all keywords are used in the current implementation of the assembler, they are reserved and may not have identifier names shadow them. Binary operations have the same meanings as in C. e.g. `<<`, is a logical shift left. See table 5 for details.

3.2 Code Data

Data words may be imbedded in the massl text segment to allow local references to 64-bit values. Such 64-bit data words are called Code Data. Since they appear directly in the text segment, branch or jump instructions must preceed the codedata to insure that execution does not reach the non-instruction word. In order to align the data to M-Machine word boundaries, an `align 0 mod 8` directive should be used prior to the data and its associated label. Since references to the data are usually made relative to the current ip, the data's adderess should be made relative as shown in figure 1.

<table>
<tr><td>valid_massl_file:</td><td>data_segment <code>text;</code> body <code>end;</code></td></tr>
<tr><td>body:</td><td>body directive</td></tr>
<tr><td>directive:</td><td>label | define | instruction | data_item | alignment</td></tr>
<tr><td>alignment:</td><td><code>align 0 mod</code> INTEGER_LITERAL ;</td></tr>
<tr><td>label:
global_label:
local_label:</td><td>global_label | local_label
IDENTIFIER ::
IDENTIFIER :</td></tr>
<tr><td>define:
global_define:
local_define:</td><td>global_define | local_define
IDENTIFIER ::= expression ;
IDENTIFIER := expression ;</td></tr>
<tr><td>instruction:</td><td><code>instr</code> operation_list ;</td></tr>
<tr><td>operation_list:</td><td>| operation operation_list</td></tr>
<tr><td>operation:</td><td>FUNC_UNIT opname operand_list
| FUNC_UNIT CONDITION CCREG OPNAME operand_list</td></tr>
<tr><td>operand_list:</td><td>| operand , operand_list</td></tr>
<tr><td>operand:</td><td>register_symbol | CCREG | # expression</td></tr>
<tr><td>register_symbol:</td><td>REGISTER | HTHREAD . REGISTER</td></tr>
<tr><td>REGISTER:
FUNC_UNIT:
CONDITION:
CCREG:
HTHREAD:
OPNAME:</td><td><code>i0 - i15, f0 - f15</code>
<code>ialu</code> | <code>falu</code> | <code>memu</code>
<code>ct</code> | <code>cf</code> | <code>ua</code> | <code>un</code>
<code>cc0 - cc3</code> | HTHREAD . <code>cc0 - cc1</code>
<code>h0</code> | <code>h1</code> | <code>h2</code> | <code>h3</code>
A valid M-Machine operation name</td></tr>
<tr><td>expression:</td><td>op
| ~ expression
| ! expression
| - expression
| expression binary_op expression</td></tr>
<tr><td>op:</td><td>IDENTIFIER | INTEGER_LITERAL | (expression)</td></tr>
<tr><td>binary_op:</td><td><code>+</code> | <code>-</code> | <code>*</code> | <code>/</code> | <code>%</code>
| <code>&</code> | <code>|</code> | <code>^</code>
| <code>&&</code> | <code>||</code> | <code>^^</code>
| <code><<</code> | <code>>></code>
| <code><</code> | <code>></code> | <code><=</code> | <code>>=</code> | <code>!=</code> | <code>==</code></td></tr>
<tr><td>IDENTIFIER:
INTEGER_LITERAL:</td><td>An alphanumeric string, which is not a keyword.
A decimal number, or hexadecimal preceeded by an 0x</td></tr>
</table>

Table 2: The massl language spec (not including data segment)

data_segment:	| `data;` data_list
data_list:	| data_item data_list | label data_list
data_item:	data_int | data_float | data_pointer
data_int:	`u64` INTEGER_LITERAL ;
data_float:	`f64` INTEGER_LITERAL . INTEGER_LITERAL `e` INTEGER_LITERAL ;
data_pointer:	`ptr` PROTECTION . LENGTH . ADDRESS ;
PROTECTION:	`r` | `rw` | `xu` | `xp` | `eu` | `ep` | `xm` | `c` | `k` | `p` | `er`
LENGTH:	*A 54-bit hexadecimal constant. No 0x prefix*
ADDRESS:	*A 6-bit hexadecimal constant. No 0x prefix*

Table 3: The massl language spec for the data segment

```
ascii asciiz s8 s16 s32 u8 u16 u32 u64 ptr f32 f64 f64l f64h xf32 xf64
common data text bss begin end
ialu falu memu
align assign false icall imme import instr jmp ljmp mod true xor
if then else
call mem or pad skip regs return typeof
r rw xu xp eu ep xm c k p er
ct cf ua un cc0 cc1 cc2 cc3 h0 h1 h2 h3 i1 i2 ...  i31 f0 f1 ...  f31
```

Table 4: Massl keywords.

3.3 Massl data segment

The massl data segment may be filled with 64-bit integers, floats, or pointers. Each item is placed into the data segment and may be referenced through labels which offset from the beginning of the data segment (i.e. a label at the beginning of the data segment has value 0x0, while a label after two integers have been defined has the value 0x10.) See figure 2 for some examples.

3.4 Labels and Scoping

Labels, which may be used as jump and branch offsets into the text segment (or relative load offsets into the code data), or load offsets into the data segment, are alphanumeric symbols suffixed by a `:` or `::` . Global labels are visible across modules are used to patch undefined symbols during linking. Local labels are visible only in the local module and may shadow local symbols in other modules.

Labels placed into the text segment are have byte-offset values referenced from the beginning of the text segment. Therefore, a label declared immediately after the `text;` directive has the value 0x0. A label declared after a 3-operation instruction has the value 0xC (3 4-byte operations.) Similarly, labels declared in the data segment start from 0 at the beginning of the segment and increase by 8 bytes for each 64-bit

```
text;
instr ialu add i4, i5, i6;
instr br foo; instr ; instr ; instr ;

align 0 mod 8;
data_word:
u64 0xABCDEF;

foo:
instr ialu mov #(data_word - relative_ip_label), i3;
relative_ip_label:
instr ialu lea i1, i3, i3;             /* reference relative to current ip */
instr memu ld i3, i3;                  /* load the actual value */
.
.
.
end;
```

Figure 1: Example use of code data

data word.

As mentioned previously, code data words must be aligned to 64-bit M-Machine word boundaries. If no explicit align directive is used, and a label appears before a code data definition, the label may be off by 4 bytes because the assembler places an explicit alignment before the code data but after the label is processed. This can lead to inconsistencies since labels may appear to have incorrect values. To guard against this behavior, always either use the align directive, or place the code-data label following the code-data definition, and reference the label with a -8 offset (i.e. `(data_label - 8)`).

3.4.1 System Labels

To distinguish user-program labels from system labels, all non-system routines written in massl should have an underscore (_) preceeding text and data labels. The system label **Main** is reserved as the entry point of the executable and will be used by the loader for any appropriate pointer patching.

3.5 Variable Declaration and Expression Evaluation

Variables may be used alone or in expressions as immediate arguments in many M-Machine operations. They may be declared by the `:=` or `::=` directives. Both variables and immediate values may be placed into expressions and evaluated. Expressions are enclosed in parentheses and include a variety of arithmetic operations such as addition, subtraction, multiplication, and logical and bitwise operations.

As is the case with labels, global definitions are visible across modules, while local definitions may not be accessed outside of the module where they are defined and do not conflict with definitions in other modules.

Mas currently performs no checking when arithmetic involving data and text labels is performed in the same expression, however perrforming such arithmetic is forbidden, because addressing for text and data labels is offset from the beginning of their respective segments and any arithmetic performed between them would be meaningless.

```
data;
start_of_segment:  /* label has value 0    */
u64 0xabcdef123;                                 /* 64-bit integer       */
f64 1.456544e34;                                 /* 64-bit float         */
ptr r.453.ff4a45;                                /* 64-bit pointer       */
negative:                                        /* local data label     */
u64 -1003023434;                                 /* can use negatives    */
f64 1.54;                                        /* no exponent required */
ptr rw.1a.f3ddf4f599;                            /* all ptr args are hex */
value_of_foobar::                                /* global data label    */
u64 foobar;                                      /* will be computed
                                                    properly             */

text;
start_of_text:                                   /* local label          */
instr memu        ldsu  0,    0, i1, i2, cc0;    /* pre and post conditions
                                                    precede all other operands */
foobar:                                          /* local label          */
instr memu ct cc2 ldsu  1,    1, i1, i2, cc1
      ialu cf cc0 br    foobar;
instr ;                                          /* nops                 */
instr ialu nop;                                  /* equivalent           */
instr ;
_main::                                          /* global label         */
instr memu cf cc3 ld   i3,   #0, h1.f7;
instr memu un cc0 ld   i2, #-11, h2.f12;
x := 5;                                          /* local variable       */
y := 6;                                          /* definitions          */
z ::= (x + y);                                   /* global definition    */
instr ialu        add  i2, #((x*y)-10), i3       /* expressions are valid
                                                    for immediate-operand
                                                    fields if they can be
                                                    evaluated at assembly-time. */
      memu        lea  i1, #0x1b,       i15      /* hex immediates are
                                                    prefixed by 0x     */
      falu        fadd f1,    f2,       f3;

/* convoluted way to perform same action as br foobar;                 */
instr ialu        mov  #value_of_foobar, i5;
instr ialu        lea  i2, i5, i5;
instr ialu        ld   i5, i5;                   /* optional operand to
                                                    ld is omitted.      */
instr ialu        br   i5;
instr ;
instr ;
instr ;

end;
```

Figure 2: Example massl code

<table>
<tr><th>Token</th><th>Operator</th></tr>
<tr><td>~</td><td>Bitwise Not</td></tr>
<tr><td>!</td><td>Logical Not</td></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>%</td><td>Modulus</td></tr>
<tr><td><< >></td><td>Left and Right shift</td></tr>
<tr><td>< > <= >=</td><td>Relational operators</td></tr>
<tr><td>== !=</td><td>Equals/Not Equals</td></tr>
<tr><td>&</td><td>Bitwise And</td></tr>
<tr><td>^</td><td>Bitwise Xor</td></tr>
<tr><td>|</td><td>Bitwise Or</td></tr>
<tr><td>&&</td><td>Logical And</td></tr>
<tr><td>^^</td><td>Logical Xor</td></tr>
<tr><td>||</td><td>Logical Or</td></tr>
</table>

Table 5: Common operators

3.6 Preprocessing

3.6.1 Macros and Comments

Macro support is provided through a transparent pass of the input file through the C preprocessor (cpp) which generates a temporary massl file used for the rest of the assembly process. No Massl language constructs interfere with the cpp grammar, so a pass through cpp before the actual assembly process will not mangle source files.

All comments in massl programs should be the `/* */` C-style comments. Nested comments are not allowed. To comment out large sections of code, use the cpp `#if 0, #endif` pair.

3.7 Error Handling

Parse errors in Massl input files are handled by printing a diagnostic, synchronizing on the next semicolon, and continuing. For this reason, errors are often reported for valid input lines because they are entirely skipped to find the next semicolon. Be aware of this when debugging your massl code.

In addition, gramatically correct but semantically incorrect programs may be discovered. During the first pass, mas identifies and warns about incorrect argument types or numbers of arguments to operations. Second-pass error detection and recovery includes flagging cases where immediates cannot fit into operation fields, or when symbols and variables are undefined in the current module.

Common errors and their meanings are listed in table 6.

4 Construction of object files

Object files contain a header, test and data relocation sections which provide information for the linker, loader, and simulator, and then text and data sections (data immediately following text.) Symbol table information is concatenated to the object file. Although the data segment immediately follows the text

Error Message	Meaning
syntax error	a fatal error while parsing the massl file. Mas will try to synchronize to the next semicolon and continue.
missing <arg> argument	An operand which was expected to be given for the operation was not provided. Check that no commas are missing and that no operands were inadvertantly left out (such as pre and post conditions)
<arg> may not be in a remote bank	An operand other than the destination was defined as <hthread-id>.register. This is illegal.
global condition register out of range	a remote ccreg other than cc0 or cc1 was used. Only cc0 and cc1 may be addressed when remote.
destination cc register cannot be remote.	The operation only allows use of local ccregs.
type argument expected, but not provided	A missing operand (like src1, src2, src3, destination, bank, length, etc.) most likely.
destination out of range: expected *type* register	The destination register was from the incorrect register file.
destination operand was not an fp or integer register	Probably tried to set the destination as an immediate or a cc register, where only int or fp registers are allowed.
operand provided is of an unexpected type	Usually stems from passing a register operand where only immediates are allowed, or vice versa.
Too many operands found to operation *op*	The operation was defined with too many operands.
expected *funcunit* opcode, found *string* instead	The opcode specified is not a valid M-Machine opcode for the declared function unit. Check your opcode string for typos and your function-unit declaration.
Error evaluating *expression*	The operand expression could not be evaluated during the assembler's second pass. Most likely due to an incorrect expression definition. Expressions which involve labels and constants or labels and other labels must be evaluated at assembly-time, so any undefined labels used in these expressions will cause such an error. Check for correct spelling of labels (since all label definitions and uses are case-sensitive)
operand could not be placed in *x*-bit field	The immediate (or evaluated expression or label) which was specified was too large to be placed into the immediate field. Use a smaller number or changed the code to perform a move if 7 bits is inadequate but 11 is.
operand could not be evaluated to an absolute	Specified operand (like bank and length) was evaluated to a symbol (perhaps a text or data label) instead of an absolute value. Only absolutes are allowed for that operand.
encountered more than one *funcunit* operation in a block	The instruction definition contained more than one operation for the specified function-unit. At most one operation per function unit is allowed.

Table 6: Common mas error diagnostics

segment in the object file, it may be loaded into a different section of memory once the object file is executed or simulated. Therefore, referencing off the end of the text segment by placing a label after the last instruction is forbidden.

When multiple object files are linked into a single executable, their text segments are concatenated and placed into the new executable. Their data segments are also subsequently concatenated and appended to the new executable's text segment. The `a_entry` field of the executable header contains the executable's entry point. The linker looks for the global label `Main` to calculate this entry point. If no such label exists, execution begins at the beginning of the text segment.

Currently, because the entry point is not recognized by the simulator, the first object file specified in a list of files to be linked will be placed first in the executable. Execution will start with that object file's text segment.

Text and data labels are incremented appropriately during linkage: if file1.out and file2.out are linked, then the value of the text labels in file2.out are incremented by the size of the file1.out text segment. This holds for the concatenated data segments as well.

All references to text symbols are stored in relocation information entries in the individual object files, so even referenced to local text symbols are updated to reflect changes in the text-label values. For this reason, absolute references to local symbols may no longer fit in some operation slots.

5 Migration from L0

The migration from current L0 programs to new massl programs is provided by a translating program called l2m. The l2m program is invoked with `l2m <infile>.L0 <outfile>.m`, and it parses a valid L0 program, producing a valid massl file. All L0 files (including header files) should be run through l2m. Note, though that a valid massl program has a single `text;` and a single `end;` declaration. Use `l2m -h <infile>.L0 <outfile>.m` to produce a massl header file without `text;` and `end;`.

l2m is meant for migration purposes only, all additional assembly code written for the M-Machine should be developed in massl.

Please note that `#include` directives are not changed in the translation, so include statements with references to .L0 files should be changed to refer to .m files (the standard massl extension.)

6 Bug Reports and Comments

Please send all bug reports, comments, or questions regarding the assembler to `yev@ai`.