An Assembler and Linker System for the M-Machine Software Project

by

Yevgeny Gurevich

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author . . . .
Department of Electrical Engineering and Computer Science
May 6, 1994

Certified by . . . .
William J. Dally
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

An Assembler and Linker System for the M-Machine Software Project

by

Yevgeny Gurevich

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This document details the design and implementation of an assembler and linker, and the specifications of their input and output languages. The assembler and linker are system tools for the M-Machine - a multi-node parallel computer being developed at the MIT AI Lab's Concurrent VLSI Architecture Group. The assembler is a derivative of the Multiflow Assembler - a system tool written for computers of similar design to the M-Machine. The assembler/linker system and included specification of the M-Machine assembly-language and M-Machine object file format, provide a solid software base for building other system tools and user programs targeted for the M-Machine.

Thesis Supervisor: William J. Dally
Title: Associate Professor

Acknowledgments

I have a great many people to thank.

First, to the many teachers and instructors who have made learning exciting for me: Mrs. Crich, Mrs. Stintson, Mrs. Budnick, Mrs. Romero, Mrs. Most, Mr. Mossa, Mrs. Kronen, Mr. Weinberg, Mr. Pullano, Mr. Orenstein, Mr. Bergman, Mrs. Berman, Mrs. Hockheimer, Mr. Pizetsky, Mr. Abramsky, Mr. Winukor, Dr. Ince, Mrs. Evans, Mrs. Kornberg, Mr. Lugo, Mr. Weinberg, Mr. Jaye, Mr. D'Alleva, Mr. Scotti, Mr. Lonardo, Dr. Nichol, Mr. Waldo, Mr. Rothenberg, Mr. Preiss, Mrs. Murphy, Prof. Fogelson, Prof. Frieden, Prof. Donaldson, Prof. Craig, Prof. Abelson, Prof. Sussman, Prof. Weihl, Prof. Guttag, Prof. Liskov, Prof. Karchmer, Prof. Lynch, Prof. Drake, Prof. Siebert, Prof. Dally, John Buck, Tony Eng, and the dozens I have missed - thank you.

Thanks to Chris B. Moore for his LaTeX wizardry. To David W. LeCompte and Stephen Chamberlin, for their friendship and support.

A special thanks to Hugh B. Morgenbesser for his help with the mechanics of writing documents and more importantly, his support and friendship for many years now. To Scott A. A. McDermott, a note of profound respect and gratitude for his constant concern and support. Thanks for teaching me to "revel in everything!"

To Marco Fillo, Nick Carter, and Rich Lethin, for their invaluable help in explaining the many aspects of this project, and patience with the bugs which they helped ferret out.

To Steve Keckler, for making my work at CVA a wonderful experience which I hope continues, for his patience when helping me in project work, his constant spirit, and that knowing smile as I approached his door with yet another question - I am greatly in your debt.

To Professor William Dally, a special note of appreciation for his supervision and advice! Thank you for allowing me to work among very bright individuals, and giving me a project which was an excellent learning experience and one I am proud of.

To Mark Gurevich, thank you for your help and support, and your love and kindness as we were growing up.

To Mikhail and Sonya, I have wished so much that you would be here to see me off. I hold you in my heart, always.

Finally, to Lina and Zinovy Gurevich: for bringing me into this wonderful world, and giving me the independence to learn and explore. You have taught me that curiosity and the thirst for knowledge is exciting. I have you both to thank for helping me get here, and beyond.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The M-Machine is a multi-node computer based on a very long instruction word (VLIW) architecture currently being designed at the Concurrent VLSI Architecture group of the MIT AI Lab. Each of the machine's nodes is capable of issuing up to 12 operations per clock cycle. This paper details the design and implemention of the machine's assembler and linker system which provides an important base of system tools as part of the M-Machine software project.

1.1 Motivation

The machine language specification is vital to compiler-writers who must have a target language for their code generator, and to current M-Machine programmers who, without a compiler, must generate system tools and sample code for the M-Machine. The assembler/linker system is used to generate executable binaries which conform to the object file format and instruction set architecture for simulation on the M-Machine simulator.

All of these system tools and specifications are available and have been used to write small-scale programs for execution on the existing simulator. In the near future, as programmers become more familiar with the newly-available tools, they will be able to start working on a base of system code for the M-Machine. This software base, including such modules as stack and function-call libraries and fault handlers will be moderate-size programs which would not be feasible to write without an efficient and robust assembler/linker system.

1.2 Overview of Project Dependencies

The M-Machine instruction set architecture (ISA) guide [1] has been used as the specification for what operations the M-Machine assembly language needs to express. The guide has also been used as a specification for the object file format since it defines the binary layout of operations as they are to be executed on the M-Machine. The assembly language specification is used by the assembler implementation, application programmers of the M-Machine, and the compiler-development project. The object-file format is used by the assembler's code-generator, the linker, and the M-Machine simulator. The proposed loader tool will also use this specification. Figure 1-1 shows these dependencies among the specifications and system tools. The input format of a system tool depends on a specification if there is an arrow leading from a specification to that tool. The output format depends on a specification if there is an arrow leading from system tool to spec.

1.3 Document Overview

The first part of this document presents two pairs of a specification and the system tool which uses it:

- Chapters 3 and 4 contain the M-Machine assembly language specification and assembler implementation
- Chapters 5 and 6 contain the M-Machine object-file format and linker implementation

The modular structure of the assembler and linker code base is described in chapter 7. Chapter 2 gives an example of how the assembler and linker tools can be used in the creation of a small M-Machine application. Finally, a discussion of programs which need to be added to the M-Machine system tools is presented in chapter 8.

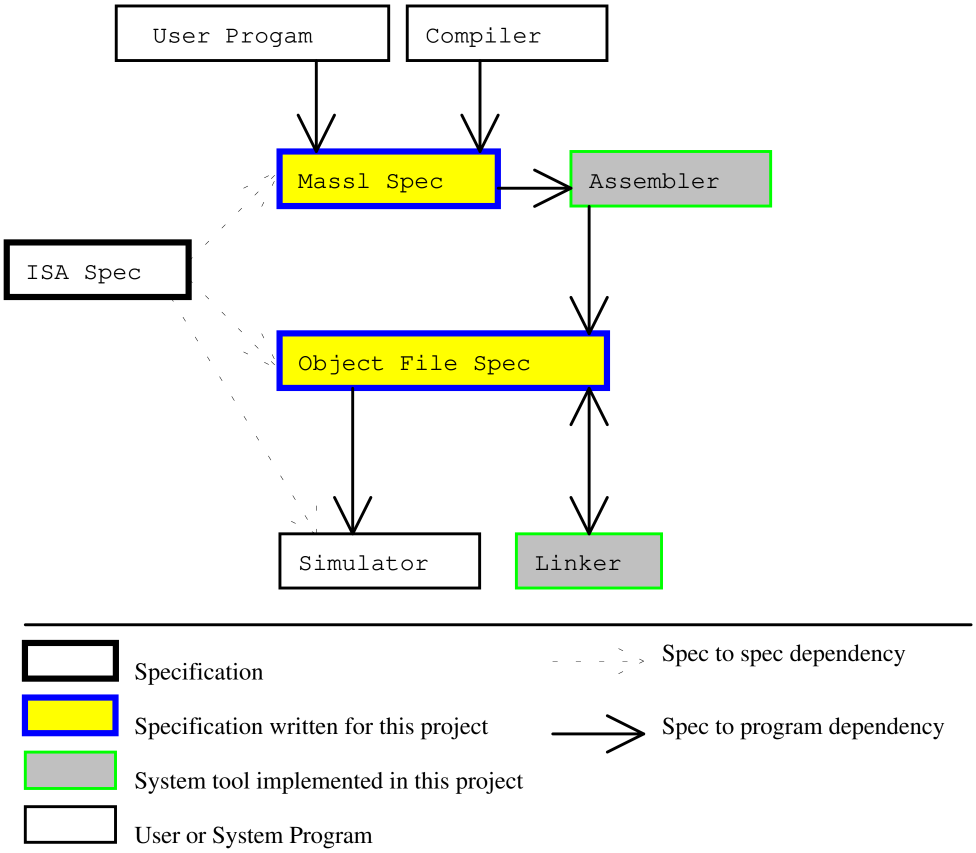

Figure 1-1: Project Dependencies

Chapter 2

Example use of system tools

This section details the use of the system tools presented in this document in the implementation of a small M-Machine based application. The Longest Common Subsequence algorithm as realized in C code is used for this example.

2.1 Original C Code

The original C code is shown in figure 2-1. It makes use of three arrays to perform computation and then store the value in another location in memory. This value is then compared against a known value to check correctness.

2.2 Massl multi-file translation

The C program is hand translated into several massl modules which take care of setting up registers correctly and also include standard definitions used in massl programs. These definitions are not strictly necessary, but provide the common set of definitions and namings which allow for easier maintenance of programs as the code base increases. Figure 2-2 shows the register namings which are used throught the project's massl code. This file, `lcs-std-head.h` is included in the massl source code. The file `lcs-std-head.m`, shown in figure 2-3 includes the data segment definition which holds the sample data set as well as allocated space for a temporary array. The

```
#define CONST_LEFT 1
#define CONST_UP   2
#define CONST_DIAG 3

int PRINT_LCS(int *b, int *X, int i,int j, int x, int curcount) {
    if (i == 0) return curcount;
    if (j == 0) return curcount;
    if (b[i+j*(x+1)] == CONST_DIAG) {
        curcount = PRINT_LCS(b,X,i-1,j-1, x, curcount);
        fprintf(stdout,"%d ",X[i-1]);
        curcount++;
    } else if (b[i+j*(x+1)] == CONST_UP) {
        curcount = PRINT_LCS(b,X,i-1, j, x, curcount);
    } else curcount = PRINT_LCS(b,X,i,j-1, x, curcount);

    return curcount;
}

int lcs(int *X, int *Y, int *c, int *b, int x,int y) {
    int i, j;
    int matchcount = 0;
    for (i = 1; i < (x+1); i++)
      for (j = 1; j < (y+1); j++) {
          if (X[i-1] == Y[j-1]) {
              c[i + j*(x+1)] = c[i-1 + (j-1)*(x+1)] + 1;
              b[i + j*(x+1)] = CONST_DIAG; /* diag*/
          } else if (c[i-1 + j*(x+1)] >= c[i + (j-1)*(x+1)]) {
              c[i + j*(x+1)] = c[i-1 + j*(x+1)];
              b[i + j*(x+1)] = CONST_UP; /* up*/
          } else {
              c[i + j*(x+1)] = c[i + (j-1)*(x+1)];
              b[i + j*(x+1)] = CONST_LEFT; /* <--*/
          }
      }
    matchcount = PRINT_LCS(b,X,x,y,x,0);
    fprintf(stdout,"\n");
    return matchcount;
}
```

Figure 2-1: Original LCS C Code

massl code in lcs-std-head.m also sets up registers to point to correct array locations in the data segment address space. Finally, lcs-v0.m, shown in figure 2-4, contains the massl implementation of the lcs algorithm, with verification code at the end if the routine to make sure that the actual result matches the expected result.

2.3 The Makefile

Just as for other system tools, the UNIX `make` utility may be used with the assembler linker system. All object files depend on massl source files, while the executable file depends on the individual object files. The makefile used for this project is shown in

```
/* Special registers    */
#define CP i1  /* c0.i1 */
#define DP i2  /* c0.i2 */
#define IntZero i0

/* Register assignments */
#define X i3 /* start of the X[] array */
#define Y i4 /* start of the Y[] array */
#define C i5 /* start of the C[] array */

#define i i6 /* outer loop index */
#define j i7 /* inner loop index */

#define m i8 /* Length( X[] ) */
#define n i9 /* Length( Y[] ) */

#define rcPrev  i10
#define lcentry i11
#define ychar   i12
#define ctentry i13

/* temp registers       */
#define Temp0 i14
#define Temp1 i15
#define Temp2 i10 /* shadow rcPrev  */
#define Temp3 i11 /* shadow lcentry */
```

Figure 2-2: LCS Header File

figure 2-5. `lcs.out` is the name of the executable file.

2.4 Mas assembly

The M-Machine assembler is called to assemble each massl file into an object file ready for linking. Warnings given during the assembly process tell the programmer that external undefined symbols were encountered, suggesting that a link pass is necessary to generate an executable which can run on the M-Machine.

2.5 Mas linking

The M-Machine linker can be used to examine the resuling object files. Figures 2-6, 2-7, and 2-8 show output from using the `lnk` debugging features (for displaying text segment disassembly, the symbol table, and text and data relocation tables).

To generate an executable, the linker is called on the resulting object files.

```
data;
XSize::
u64 4;
YSize::
u64 8;
XStart::
u64 71;
u64 114;
u64 97;
u64 94;
YStart::
u64 117;
u64 117;
u64 71;
u64 94;
u64 117;
u64 117;
u64 117;
u64 117;
CStart::
u64 0;
u64 0;
u64 0;
u64 0;
u64 0;
u64 0;
u64 0;
u64 0;
Actual::
u64 2;
Answer::
u64 0;
```

```
#include "lcs-std-head.h"
text;
/* Constants */
LogWordSize ::= 3;

_main::
/*      define the X and Y array bounds         */
_lcsinit:
instr ialu mov XStart, Temp0;
instr ialu lea DP, Temp0, X;
instr ialu mov YStart, Temp0;
instr ialu lea DP, Temp0, Y;
instr ialu mov CStart, Temp0;
instr ialu lea DP, Temp0, C;

/*      load in the X and Y length values       */
instr ialu lea DP, XSize, Temp0;
instr memu ld  Temp0, #8, m;
instr memu ld  Temp0, #8, n;

instr ialu br _postinit;
instr;
instr;
instr;
end;
```

Figure 2-3: lcs-std-head.m massl code

2.6 Simulation

Finally, the executable is passed to the M-Machine simulator. The simulator uses the object-file read and data-access functions which are provided by some of the modular linker source files to open the executable and read the text and data segments. Once its internal data structures are set up, it may simulate the executable as it will eventually run on the hardware.

```
#include "lcs-std-head.h"

text;
_postinit::
/* L0:  i := 0                          */
instr ialu mov #0, i;

_OuterLoop::
/* C:   ychar = YStr[i]                 */
instr ialu lsh i, LogWordSize, Temp1;
instr ialu lea Y, Temp1, Temp1;
instr memu ld Temp1, ychar;

instr ialu mov #0, lcentry;
instr ialu mov #0, rcPrev;

/* L0:  j := 0                          */
instr ialu mov #0, j;

_InnerLoop::

/* C:   ctable_entry = CTable[j]        */
instr ialu lsh j, LogWordSize, Temp1;
instr ialu lea C, Temp1, Temp1;
instr memu ld Temp1, ctentry;

/* C:   if (XStr[j] == ychar)                   */
instr ialu lsh j, LogWordSize, Temp1;
instr ialu lea X, Temp1, Temp1;
instr memu ld Temp1, Temp0;
instr ialu nop;
instr ialu nop;
instr ialu ieq Temp0, ychar, cc2;
instr ialu nop;
instr ialu ct cc2 br _foundMatch;
instr ialu nop;
instr ialu nop;
instr ialu nop;

/* not found match.                             */
/* C:   if (lst_ctable_entry > ctable_entry)    */
instr ialu igt lcentry, ctentry, cc2;
instr ialu nop;
instr ialu ct cc2 br _Case2;
instr ialu nop;
instr ialu nop;
instr ialu nop;

/* C:   lst_ctable_entry = ctable_entry         */
instr ialu mov ctentry, lcentry;
instr ialu br _continue;
instr ialu nop;
instr ialu nop;
instr ialu nop;

_foundMatch:
/* C:   lst_ctable_entry = rc_prev + 1;         */
/* CTable[j] = lst_ctable_entry;                */
instr ialu add rcPrev, #1, lcentry;
instr ialu lsh j, LogWordSize, Temp1;
instr ialu lea C, Temp1, Temp1;
instr memu st lcentry, Temp1;
instr ialu br _continue;
instr ialu nop;
instr ialu nop;
instr ialu nop;
end;
```

```
_Case2:
/* C: CTable[j] = lst_ctable_entry; */
instr ialu lsh j, LogWordSize, Temp1;
instr ialu lea C, Temp1, Temp1;
instr memu st lcentry, Temp1;

_continue:
/* C: rc_prev = ctable_entry; */
instr ialu mov ctentry, rcPrev;

/* C: j := j + 1        */
instr ialu add j, #1, j;

/* C: j <= n + 1        */
instr ialu ilt j, m, cc2;
instr ;
instr ialu ct cc2 br _InnerLoop;
instr ;
instr ;
instr ;

/* END of inner loop                            */
/* C:       i := i + 1                          */
instr ialu add i, #1, i;

/* C:       i <= m + 1                          */
instr ialu ilt i, n, cc2;
instr ;
instr ialu ct cc2 br _OuterLoop;
instr ;
instr ;
instr ;

/* END of outer loop */
/* no C counterpart.  Places answer in a magic location. */
instr ialu mov Answer, Temp0;
instr ialu lea DP, Temp0, Temp1;
instr memu st lcentry, Temp1;
instr ialu mov Actual, Temp0;
instr ialu lea DP, Temp0, Temp0;
instr memu ld Temp0, Temp0;
instr ialu ieq Temp0, lcentry, cc1;
instr ialu ct cc1 br Pass;
instr ;
instr ;
instr ;
instr ialu br Fail; /* if something goes wrong */
instr ;
instr ;
instr ;
```


Figure 2-4: lcs-v0.m massl code

```
ASSEMBLER = mas2
LINKER    = lnk

lcs.out: boot.o lcs-std-head.o lcs-v0.o
        $(LINKER) boot.o lcs-std-head.o lcs-v0.o -o lcs.out

boot.o: boot.m
        $(ASSEMBLER) boot.m -o boot.o

lcs-std-head.o: lcs-std-head.m
        $(ASSEMBLER) lcs-std-head.m -o lcs-std-head.o

lcs-v0.o: lcs-v0.m
        $(ASSEMBLER) lcs-v0.m -o lcs-v0.o
```

Figure 2-5: Example makefile for the LCS project

```
Text Segment for lcs-std-head.o:
$00000000: [002c7010] ialu mov #16, i14
$00000004: [00b0190e] ialu lea i2, i14, i3
$00000008: [002c7030] ialu mov #48, i14
$0000000c: [00b0210e] ialu lea i2, i14, i4
$00000010: [002c7070] ialu mov #112, i14
$00000014: [00b0290e] ialu lea i2, i14, i5
$00000018: [00b47100] ialu lea i2, #0, i14
$0000001c: [40844708] memu ld i14, #8, i8
$00000020: [40844f08] memu ld i14, #8, i9
$00000024: [00640013] ialu br #19
$00000028: [00010000] ialu nop
$0000002c: [00010000] ialu nop
$00000030: [00010000] ialu nop
$00000034: [00010000] ialu nop
$00000038: [00000000] ialu ill
$0000003c: [00000000] ialu ill
$00000040: [00000000] ialu ill
$00000044: [00000000] ialu ill
```

Figure 2-6: Linker Debugged Text Segment Output

```
lcs-std-head.o Symbols:
              Actual    D       0x000000b0
              Answer    D       0x000000b8
              CStart    D       0x00000070
         LogWordSize    A       0x00000003
               XSize    D       0x00000000
              XStart    D       0x00000010
               YSize    D       0x00000008
              YStart    D       0x00000030
            _lcsinit    t       0x00000000
               _main    T       0x00000000
           _postinit    U       0x00000000
```

Figure 2-7: Linker Debugged Symbol Table Output

```
lcs-std-head.o Relocation:
        # Reloc entries = 5
                $00000000:    B    Relocation 11-bit    5       XStart
                $00000008:    B    Relocation 11-bit    7       YStart
                $00000010:    B    Relocation 11-bit    2       CStart
                $00000018:    AECI Relocation  7-bit    4       XSize
                $00000024:    Relative G Rel. 18-bit    10      _postinit

        # Reloc entries = 24
                $00000000:              64-bit Int    -1      <ABS>
                $00000008:              64-bit Int    -1      <ABS>
                $00000010:              64-bit Int    -1      <ABS>
                $00000018:              64-bit Int    -1      <ABS>
                $00000020:              64-bit Int    -1      <ABS>
                $00000028:              64-bit Int    -1      <ABS>
                $00000030:              64-bit Int    -1      <ABS>
                $00000038:              64-bit Int    -1      <ABS>
                $00000040:              64-bit Int    -1      <ABS>
                $00000048:              64-bit Int    -1      <ABS>
                $00000050:              64-bit Int    -1      <ABS>
                $00000058:              64-bit Int    -1      <ABS>
                $00000060:              64-bit Int    -1      <ABS>
                $00000068:              64-bit Int    -1      <ABS>
                $00000070:              64-bit Int    -1      <ABS>
                $00000078:              64-bit Int    -1      <ABS>
                $00000080:              64-bit Int    -1      <ABS>
                $00000088:              64-bit Int    -1      <ABS>
                $00000090:              64-bit Int    -1      <ABS>
                $00000098:              64-bit Int    -1      <ABS>
                $000000a0:              64-bit Int    -1      <ABS>
                $000000a8:              64-bit Int    -1      <ABS>
                $000000b0:              64-bit Int    -1      <ABS>
                $000000b8:              64-bit Int    -1      <ABS>
```

Figure 2-8: Linker Debugged Relocation Table Output

Chapter 3

The Massl Assembly Language

The massl assembly language is the primary low-level language used for writing valid blocks of code which can be executed on the M-Machine.

3.1 Precursors to Massl

The massl assembly language was not designed as a new language from the ground up. Instead, it is a hybrid of several existing languages which have preceeded it, combined with original additions to support new features.

3.1.1 Multiflow Assembly Language

Massl is based in part on the native language of the Multiflow computer system, whose assembler was used as a basis for the M-Machine assembler. The Multiflow machine is a VLIW architecture, and its language therefore supports the concept of instruction blocks which contain multiple operations (each meant for a different functional unit.) In addition, the language supports a variety of literal types, such as signed and unsigned 16- and 32- and 64-bit integers, and 32- and 64-bit floats.[1] The standards for naming local and global labels and variables, and constructing expressions have been taken directly from this specification which encapsulates all

[1] Although 16- and 32-bit integer data types are supported in the language specification and assembler and linker implementations, the 64 bits is the native M-Machine data word size.

of the basic requirements of a useful assembly language. Finally, the more global language features such as the text and data segment delimiters have all been taken from this language specification as well.

3.1.2 M-Machine L0 Assembly Language

The precursor to the M-Machine assembler was the Masm assembler, which used L0 as its native input language. L0 is a lisp-like assembly language which captures the more machine-specific features of the M-Machine architecture such as predicated operations. The standard method of expressing conditions for operations has been taken directly from L0 with few modifications.

Both the multiflow language and L0 recognize multiple register files as names for register operands. Massl borrows this <cluster>.<register> notation for its <hthread>.<register> identification of all register operands.

3.1.3 Massl-Specific Language Features

Support for 64-bit pointer and integer types as data words was added to the language specification, since this support was missing from both L0 and the multiflow assembly language. In addition, massl supports global condition-register addressing.

3.2 Massl Specification

Massl is a highly-structured, instruction-oriented language. Each massl program must begin with a segment directive (either `text;` or `data;`) which is active until the next segment supersedes it and defines the active program segment. All massl programs must end with the `end;` directive. A BNF description of massl is shown in tables 3.1 and 3.2.

Integer literals can be represented either by decimal numbers, or hexadecimals if preceded by `0x`. An identifier is an alphanumeric string which is not a massl keyword (see table 3.3.) Although not all keywords are used in the current implementation

```
valid_massl_file:     data_segment text; body end;

body:                 body directive

directive:            label | define | instruction | data_item | alignment

alignment:            align 0 mod  INTEGER_LITERAL ;

label:                global_label | local_label
global_label:         IDENTIFIER ::
local_label:          IDENTIFIER :

define:               global_define | local_define
global_define:        IDENTIFIER ::= expression ;
local_define:         IDENTIFIER := expression ;

instruction:          instr operation_list ;

operation_list:       | operation operation_list
operation:            FUNC_UNIT opname operand_list
                      | FUNC_UNIT CONDITION CCREG OPNAME operand_list

operand_list:         | operand , operand_list
operand:              register_symbol | CCREG | # expression

register_symbol:      REGISTER | HTHREAD . REGISTER

REGISTER:             i0 - i15, f0 - f15
FUNC_UNIT:            ialu | falu | memu
CONDITION:            ct | cf | ua | un
CCREG:                cc0 - cc3 | HTHREAD . cc0 - cc1
HTHREAD:              h0 | h1 | h2 | h3
OPNAME:               A valid M-Machine operation name

expression:           op
                      | unary_op expression
                      | expression binary_op expression

op:                   IDENTIFIER | INTEGER_LITERAL | ( expression )

unary_op:             ~ | ! | -

binary_op:            + | - | * | / | %
                      | & | | | ^
                      | && | || | ^^
                      | << | >>
                      | < | > | <= | >= | != | ==
IDENTIFIER:           An alphanumeric string, which is not a keyword.
INTEGER_LITERAL:      A decimal number, or hexadecimal preceeded by an 0x
```

Table 3.1: The massl language spec (not including data segment)

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

Table 3.2: The massl language spec for the data segment

of the assembler, they are reserved and may not have identifier names shadow them. Binary operations have the same meanings as in the C programming language. e.g. `<<`, is a logical shift left. Table 3.4 lists all recognized expression operators and their meanings.

```
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 3.3: Massl keywords.

3.2.1 Instructions - The Text Segment

All massl instructions start with the token word `instr`, contain up to three operations (each from a different M-Machine functional unit) and end with a semicolon. Operands to an operation (except the optional condition/condition-register pair) are comma-delimited and denote sources first, then destinations (i.e. src1, src2, src3,

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

Table 3.4: Common operators

destination). Figure 3-1 contains some examples of basic instruction definitions. The code demonstrates the declaration of the text segment with the `text;` directive. Several instructions follow, and an `end;` directive denotes the end of the massl program.

```
text;                                /* beginning of text segment     */
instr ialu add i1, i2,    i3;        /* simple integer ialu operation */
instr ialu add i1, #15,   i3;        /* use of an immediate           */
instr ialu add i1, #0x1B, i3;        /* use of a hex immediate        */
instr ialu ct cc1 add i1, i2, i3     /* operation predicated on cc1   */
      memu        ld  i4, #0, i5;    /* 2nd operation in instruction  */
end;                                 /* end of code.                  */
```

Figure 3-1: Sample massl instructions

Pre and Post conditions to operations which require them are comma-delimited and precede all other operands as shown in Figure 3-2. Bank and Length are treated in the same manner as pre and post condition operands, and precede all other register and immediate operands as shown in Figure 3-2. Since no operation requires both pre/post conditions and bank/length operands, there is no ambiguity when requiring that pre/post and bank/length precede all other operands. Operations which have both integer/floating-point and conditional destinations have the integer/floating-point register destinations specified first, followed by the condition reigsters. The

M-Machine ISA document [1] provides detailed descriptions of M-Machine operations and their operand fields.

```
text;                               /* beginning of text segment     */
instr memu lds 2, 1, i3, f2;        /* pre and post conditions
                                       are listed before other
                                       register operands.            */
instr ialu send0 1, 3, i3, i4, cc2; /* bank and length are listed
                                       first as well.                */
end;                                /* end of code.                  */
```

Figure 3-2: Sample instructions with Pre and Post conditions, and Bank and Length operands

A wider range of allowable text-segment directives is shown in figure 3-3. This example code includes declarations of both the text and data segments. Local and global text and data labels are shown, as well as local and global variable declarations (`x`, `y`, and `z` in the example.)

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 code data words appear directly in the text segment, branch or jump instructions must precede them to insure that execution does not reach non-instruction words. 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 3-4.

3.2.2 Massl Data Directives - The 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

```
text;
start_of_text:                                  /* local label       */
instr memu       ldsu  0,    0, i1, i2, cc0;    /* pre & post condi-
                                                   tions precede all
                                                   other operands    */
foobar:                                         /* local label       */
instr memu ct cc2 ldsu  1,    1, i1, i2, cc1
      ialu cf cc0 br    foobar;
instr ;                                         /* nop               */
instr ialu nop;                                 /* same as above     */
instr ;
_main::                                         /* global text 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 ;
data;                                           /* start of the data
                                                   segment           */
value_of_foobar:                                /* local data label  */
u64 foobar;                                     /* 64-bit data word  */
end;
```

Figure 3-3: Example massl text and data segment code

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

align 0 mod 8;              /* alignment required when using codedata */
data_word:                  /* text label with address of data word   */
u64 0xABCDEF;               /* actual value placed in text segment    */

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 3-4: Example use of code data in massl

data segment has value 0x0, while a label after two integers have been defined has the value 0x10.) Figure 3-5 contains examples of a data segment definition.

3.2.3 Labels and Scoping

Labels, which may be used as jump and branch offsets into the text segment, relative load offsets into the code data, or load offsets into the data segment, are alphanumeric symbols suffixed by `:` for local labels or `::` if global. Global labels result in the generation of externally-defined symbols which are visible across object-file modules when used in linking (see chapter 6.) Local labels are visible only in the local module and may shadow local symbols in other modules.

Labels placed into the text segment 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

```
data;                      /* denotes that all following data word
                              definitions go into the data segment */
start_of_segment:          /* data label has value 0 relative to
                              data segment pointer                 */
u64 0xabcdef123;           /* 64-bit integer                       */
f64 1.456544e34;           /* 64-bit float                         */
ptr r.453.ff4a45;          /* 64-bit pointer                       */
foobar:                    /* local data label                     */
u64 -1003023434;           /* can use negatives                    */
f64 1.54;                  /* no exponent required for floats      */
ptr rw.1a.f3ddf4f599;      /* all pointer fields are hexadecimal   */
value_of_foobar::          /* global data label                    */
u64 foobar;                /* The value of label foobar will be
                              located in this 64-bit slot in the
                              data segment.                        */
end;
```

Figure 3-5: A data segment defined in massl

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, either the `align` directive should be used, or the code-data label placed *following* the data word definition, and then referenced with a -8 offset (i.e. `(data_label - 8)`.)

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 the function it represents will be used by the loader for any appropriate pointer patching.

3.2.4 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 `::=` tokens. Both variables and immediate values may be placed into expressions and evaluated at assembly time. 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.

Performing arithmetic involving variables or labels from different segments in the same expression is forbidden, because addressing for text and data labels is offset from the beginning of their respective segments and any arithmetic performed between them could be meaningless.

3.2.5 Macros and Comments

There is no direct support for macros in the massl specification. See section 4.3.1 for details of assembler-level macro support through the C preprocessor.

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, supported by the assembler implementation.

3.3 Future extensions

A useful feature to add to the massl specification is the use of an Hthread$n$ directive to signify the global designation of the hthread number for which code is written. In this way, it is possible to add support for both relative and absolute thread-addressing of condition and integer/fp registers by comparing the current-code hthread designation with the hthread designation of the register in question.

Chapter 4

The M-Machine Assembler

The M-Machine assembler, `mas`, was written to supercede the original assembler for the L0 language, `Masm`. The assembler also serves as an interface to the M-Machine linker for creation of full executables. It is a two-pass assembler based heavily on the Multiflow Assembler. The input to the assembler consists of text files which conform to the massl specification. Mas outputs binary files conforming to the M-Machine object file specification.

The mas assembler should be able to correctly assemble all legal massl programs and reject input files not conforming to the massl specification. Diagnostic error and warning messages are essential for this assembler since massl programers and compiler-writers will be targeting their code to this tool, and need to identify where errors in their code occur and why.

4.1 Origin of Mas - the Multiflow Assembler

The Multiflow Assembler (MFA) was used as the basis of mas. It was chosen for its efficiency, support of a VLIW architecture similar to that of the M-Machine, and the potential for leveraging much of the code to support the needs of the M-Machine assembler. In those respects, it has served the development effort well.

MFA is a two-pass assembler which parses input files using C code generated by the `yacc` tool to generate intermediate data structures to be used in its second pass. The second phase performed many machine-specific transformations which were replaced with translations to M-Machine binary operation-encodings based on the ISA document .

A useful feature of the Multiflow system is its recognition of tokens left by the C preprocessor for identifying source file names and lines when file inclusion is used. This becomes very useful when an include-file, no longer visible as an input file at the time of assembly, contains errors which need to be reported.

Beyond the target-architecture specific changes to the MFA, support for longer datatypes (64-bit integers) and multi-file assembly has been added to the base code. In addition, error-checking and handling has been greatly improved - whereas MFA stopped parsing at the first severe parse error, mas synchronizes instruction parsing on the semicolon token and continues parsing.

4.2 Mas Usage

Mas, when used in its default state, assembles files identified by their filenames on the command line and calls the external M-Machine linker to build an executable:

```
mas file1.m file2.m file3.m file4.m
```

Intermediate object files for each massl input file are created during this process and may be reused in the link phase, without the need for reassembly. Command line options allow the user to override this default behavior and force only assembly or only linkage of input files. Output object or executable file names may be specified as well to override defaults. Table 4.1 shows all command-line arguments supported by mas.

4.3 Phases of assembly

Mas maintains the multi-phase nature of MFA by dividing the source file assembly into several stages. The preprocessing stage, missing from the MFA implementation, is a standard component of the mas assembly process. When performing multi-file

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 to *filename* instead of a.out, when multiple files are given on the command line. May not be used with -c option if more than one file is being assembled.
-c *file list*	Assemble multiple .m files into .o files (each *filename*.m is assembled to *filename*.o) 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. Mas 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.)
-I*directory*	Adds *directory* to the path used by cpp to search for include files. Multiple -I directives may be used in a call to mas. Useful for defining system-code header files with global project definitions.
-D	Include debugging information in the resulting object file. (default)

Table 4.1: Mas Command Line Options

assembly, mas runs through all of the phases described below on each file in turn. If fatal errors occur when assembling an input file, files still remaining to be assembled are not processed.

4.3.1 Preprocessing

A transparent pass of input files through the C preprocessor, `cpp`, generates a temporary massl file which is used for the rest of the assembly process. This phase adds the support for macro definition and substitution which is not part of the massl spec., as well as the special `#ifdef` directives for conditional inclusion of source text. Since no massl language constructs interfere with the cpp input grammar, source files will not be mangled by this pass. The previously-mentioned support for recognizing special tokens left by cpp when processing `#include` directives in the MFA parser was left intact, allowing error reporting which uses original input file names and line numbers, rather than those of the temporary input file which is never seen by the user.

The `-I` command line option to mas passes include directories to cpp during this phase. This way, directives of the form `#include <filename.h>` will allow cpp to find header files if they are located in canonical locations outside of the current directory.

4.3.2 Parsing

The massl language spec is implemented through a grammer definition targeted for the `yacc` parser. The grammar, a modified version of the MFA grammar, closely follows the BNF language description given in the massl specification. Certain constructs recognized by MFA, such as uninitialized data segment definitions and symbol-table directives, have remained in the yacc file to allow future extensions to the massl language and their consequent support in the assembler. During the parsing stage, internal linked lists of instructions and data words are generated. These intermediate data structures were modified from their original MFA structure to conform more closely to the M-Machine architecture. Each text or data segment entry is represented by a `block` which contains a linked list of instructions. Blocks contain a program counter which stores the number of bytes between the start of the segment and the block, and a size parameter which represents the length of the block in bytes (a three-operation block has a size of 12, while block representing a 16-bit unsigned integer data word has a size value of 2.) Linked lists of blocks are maintained for each segment - text, data, and uninitialized (BSS).

4.3.3 Verification and Operation Filling

After the parsing stage, expressions and symbol-tables are verified for consistency. Every operation stored in the block intermediate data structures is then packed into the 32-bit binary format specified in the M-Machine ISA document. Expressions which cannot be evaluated at this stage due to multiple undefined symbols cause fatal errors which halt the assembly process. Values which cannot be placed in limited-width instruction bitfields also cause fatal errors. Relocation entries for all non-absolute symbols are generated for every operation and data word which refers to a symbol.

4.3.4 Object Code Generation

Since the M-Machine and Multiflow object file formats closely mirror the UNIX object-file format, the Multiflow Assembler's code-generation section was used as a good base for the assembler's back end. Issues specific to M-Machine code generation which were addressed by mas include

- M-Machine specific operation and data relocation entries.
- 64-bit word packing of operations
- generation of a packed-instruction format text segment
- ability to output both regular and debug-format object files

Relocation Entry Generation

Text and data relocation entries are generated to conform to the M-Machine object file format specification (chapter 5.) The MFA relocation-entry generation code has been replaced by functions specific to the M-Machine. The relocation-entry data structures which are used conform to their UNIX formats, as opposed to the MFA's multiflow-specific structures. In addition, references to all text symbols, whether local or global, require relocation entry generation. This behavior is required because certain M-Machine operations such as `jmp` use absolute text-segment offsets which can change once segments are concatenated during the link phase. Therefore, even though the value of a symbol is known at assembly time because it is local, its value may change if the local module's text segment is concatenated to other code and all local and global symbols are adjusted.

64-bit Word Packing

Since the M-Machine is a fully 64-bit architecture, pairs of 32-bit operations need to be packed into 64-bit words. Therefore, if first operation A and the operation B need to be stored in the object file, they must be placed in a single 64-bit word of the form BA. Calls to write operations to the object file are sent to a function which maintains a static variable to output pairs of operations in this byte ordering. A special call to flush out stored operations, in the case that the number of operations defined was odd, is made in the object file generation code at the end of the text segment output function.

Instruction Packing

Mas follows the ISA document specification for a packed instruction stream format. [1] Each instruction which contains fewer than the maximum of three operations does not waste instruction-stream space by the insertion of explicit nops. Instead, each internal instruction data structure is examined to determine exactly which operations are present, and appropriate pack bits are used when these operations are written to the object file. In the case that instructions contain no operations (such as when defined as `instr ;`, an explicit integer nop operation is used.

Regular and Debugg-format Object File Generation

Although the current implementation of the assembler defaults to outputting object files in the debug-format, the regular object file format is also supported in the text segment and relocation entry output code.[1] This feature has not been extensively tested, however.

4.4 Error Handling

Parse errors in massl input files are handled by printing a diagnostic, synchronizing on the next semicolon token, and continuing. For this reason, errors are often reported for valid input lines because they are entirely skipped to find the next semicolon. This facility often allows mas to continue parsing despite catastrophic errors in input

[1] Debug-format assembly is required by the M-Machine simulator, which only accepts debug-format executables in its current implementation.

files. This has the advantage of catching more than one or two errors in the input file, relieving the user of the assembler from needing to reassemble input files after each error is found and fixed. [2]

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

Common errors and their meanings are listed in table 4.2.

In all cases, catastrophic errors prevent an object file from being generated. Warnings, such as those generated when undefined symbols are referenced, do not prevent object file output.

4.5 Future Extensions

The current implementation of the assembler cannot resolve arithmetic expressions which contain references to undefined symbols. It is possible to add simple expressions which are based on undefined symbols into text and data relocation records for evaluation at link time. Binary addition or subtraction expressions, where one of the branches can be resolved at assembly-time while the other depends solely on an undefined symbol, may be expressed as the sum of an undefined symbol and a signed integer which is known at the time of assembly and placed into the relocation record of the operation which referenced the expression. There is currently room in the relocation record for a 32-bit signed integer (see section 5.4.)

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 4.2: Common Mas Error Diagnostics

Chapter 5

The M-Machine Object File Format

There are currently two M-Machine object file formats - the regular and the debug-format. Both formats closely follow the standard for UNIX object files as specified in the `a.out.h` and `exec.h` standard C header files. M-Machine object files usually consist of 6 sections: a file header, text and data segments, text and data relocation entry tables, and a symbol and string table. Valid object files must contain the file header, but may have some of the other sections missing. Valid executable file must contain the text segment. Figure 5-3 provides a good overview of all object file sections and data structures discussed in this chapter.

5.1 File Header

The first 32 bytes of every object file contain the file header. The header contains information on the file's segments, and relocation and symbol tables. Table 5.1 summarizes header-file contents.

The contents of the object file header may be used to access the file's individual components. Due to the object file format's conformance to the UNIX standard, macros in the standard header file `a.out.h` (such as `N_TXTOFF()` and `N_DATOFF()`) may be used to obtain file byte-offsets into the object files of individual segments and

Bytes	Description
1	Object file version information. The 7-bit value starting at bit one of this byte contains the object file type. Currently, 0x1 denotes a regular object file, while 0x41 denotes a debug-format object file.
2	Machine type (unused.)
3-4	Magic Number (currently, the magic number of the platform where lnk was compiled.)
5-8	Text Segment Size. The size of the object file's text segment in bytes.
9-12	Data Segment Size. The size of the object file's data segment in bytes.
13-16	BSS Segment Size. The size of the object file's bss segment in bytes.
17-20	Symbol Table Size. The size of the object file's symbol table.
21-24	Execution Entry. The byte address in the text segment where program execution should start.
25-28	Text Relocation Size. The size of the text relocation table in the object file.
29-32	Data Relocation Size. The size of the data relocation table in the object file.

Table 5.1: M-Machine object file header spec

tables.

The object file version bits are used to determine the type of object file and will provide support for future file formats.

5.2 Text Segment

The object file text segment begins at byte 33 of the object file (if the header is considered to be in bytes 1-32) and continues for the number of bytes specified in the text segment size field of the file header. Addresses in the text relocation table of the object file are referenced off the beginning of this local text segment, so byte address 64 in a relocation entry denotes the operation which starts at byte 97 of the object file.

Debug-format object files contain a tag byte for every 64-bit word in the text segment which denotes whether the following 64-bits are operations, or a 64-bit code data word. Therefore, the true size of the text segment as loaded into M-Machine memory (one with the tag bytes stripped) is 8/9 of that denoted in the file header.

5.3 Data Segment

The data segment begins where the text segment ends and continues for the number of bytes specified in the header data segment size field. Data reloction addresses reference off the beginning of this data segment, so byte address 64 denotes the data word staring at byte (97 + size of the text segment in bytes).

5.4 Text Relocation Table

Text relocation entries are placed in the text relocation table of the object file for each symbol which was referenced in the text segment of the massl source for that object file. Each relocation entry is a 12-byte structure of the form shown in table 5.2. The C definition of the structure representing this entry, taken from the current development platform for M-Machine system tools, is given in figure 5-1. All UNIX systems should have relocation entry definitions similar to the one shown.

Address	Symbol Index	Relocation Type	Unused
4	3	1	4

Table 5.2: Relocation Entry Layout

```
struct reloc_info_sparc {            /* used when header.a_machtype == M_SPARC */
   unsigned long int r_address;      /* relocation addr (offset in segment) */
   unsigned int      r_index   :24;  /* segment index or symbol index       */
   unsigned int      r_extern  : 1;  /* if F, r_index==SEG#; if T, SYM idx  */
   int                         : 2;  /* <unused>                            */
   enum reloc_type   r_type    : 5;  /* type of relocation to perform       */
   long int          r_addend;       /* addend for relocation value         */
};
```

Figure 5-1: C relocation-entry definition

The first 32 bits denote the address into the text segment of the operation or, in the case of Code Data, data word which referenced the symbol. The next byte identifies the type of relocation entry. If the most significant bit is set, the relocation references an external symbol. The 5 least significant bits determine the exact type of operation or data word involved in the reference. This relocation type is summarized in table 5.3.

Value	Relocation Type
0	absolute 7-bit class A, C, E, or I operation symbol reference.
1	absolute 11-bit class B operation symbol reference.
2	absolute 18-bit class G operation symbol reference.
3	relative 18-bit class G operation symbol reference.
9	64-bit integer symbol referenced in a data word.
11	64-bit floating-point symbol referenced in a data word.
12	64-bit M-Machine pointer symbol referenced in a data word.

Table 5.3: Relocation Types

The next field contains either the index into the local symbol table of the symbol referenced or a generic symbol type. For debug-format object files, this always a denotes a symbol-table index. For regular object files, the field is an index into the local symbol table if the relocation entry is for an external symbol, and a generic symbol type (as defined in table 5.5) otherwise.

Due to the fact that debug-format object files contain relocation entries for each text segment code data word and data segment word, the special value of `0xffffff` denotes a relocation entry which has no corresponding symbol in the symbol table.[1] The final 4 bytes are currently unused.

5.5 Data Relocation Table

The data relocation table contains entries identical in layout to those of the text relocation relocation table. There exists a data relocation entry for every symbol referenced by a data word in the data segment. In addition, the debug-format object file format includes an entry for every word of the data segment, irrespective if whether that word referenced a symbol or not. These relocation entries provide information

[1]Relocation entries for data words are required to identify their types, so as to distinguish 64-bit floats, pointers, and integers.

about the type of data word stored at each data segment address. Relocation types 9, 11, and 12 are used for this purpose. (see table 5.3.) This support for debug-format entries in the data segment is required for simulation purposes, since the current M-Machine simulator requires tagged data types. The debug-format relocation entries provide this tagged information without altering the data segment itself. Therefore, in the future it will be possible to ignore the relocation entries: if the executable is simulated on a system which does not require knowledge about data types in the data segment, the simulator need only access the data segment which is composed of 64-bit, untagged words.

5.6 Symbol and String Tables

The symbol table is appended to the object file data relocation table and contains entries for each symbol which is referenced by relocation entries. It acts as an index into the string table to access actual symbol names, and as a source of symbol types and values. Each symbol-table entry is 12-byte structure of the form show in table 5.4.

String-Table Index	Symbol Type	Unused	Symbol Value
4	1	3	4

Table 5.4: Symbol Table Entry Layout

The first 4-bytes are an index into the object file string table, which is appended to the symbol table. The string table contains null-terminated ASCII-strings naming all symbols represented by the symbol table entries. The next byte represents the type of symbol (local or global; text, data, or absolute; defined or not.) If the least-significant bit of the bit is set, the symbol is considered globally-defined (an external symbol.) Table 5.5 lists a taxonomy of allowed symbol types and their meanings. After 3 bytes currently left unused, the 32-bit value of the symbol follows. If the symbol type represents an undefined symbol, this value is meaningless. Otherwise, it represents the value of the symbol known at assembly-time. Since the value field is limited to 32 bits (in conformance to the UNIX object file standards), symbols in current M-Machine object files may not reference addresses longer than 32 bits. A C structure definition for string table entries is given in figure 5-2. The first four bytes are represented as a union type because once the symbol table and string table are read into memory, it is move convenient to use the first 4 bytes of the symbol table entry as a pointer to an in-memory string rather than as an index into the string table.

Type Value	**Symbol Description**
0	Undefined symbol (a reference to some symbol externally-defined.)
2	Absolute Symbol
4	Text Segment Symbol
6	Data Segment Symbol
8	Uninitialized Data Segment (BSS) Symbol

Table 5.5: Relocation Types

```
struct  nlist {
union {
char    *n_name;        /* for use when in-core   */
long    n_strx;         /* index into object file
   string table           */
} n_un;
unsigned char   n_type;         /* symbol type flag       */
char    n_other;                /* unused                 */
short   n_desc;                 /* unused                 */
unsigned long   n_value;        /* value of symbol        */
};
```

Figure 5-2: Symbol table entry structure definition

The string table following the symbol table begins with a 32-bit unsigned integer denoting the size of this last part of the object file in bytes - the *string table size*. The minimum size of a string table is therefore 4 bytes - the case when it contains no strings. Those four bytes contain the value 4, representing the very same 32 bits used to store the size of the empty table. If the string table size is larger than 4, then the table is non-empty and the total number of bytes of string data stored in the table is *string table size - 4*. Null-terminated strings in the string-table are stored one after another; their starting characters are referenced by the symbol-table index fields.

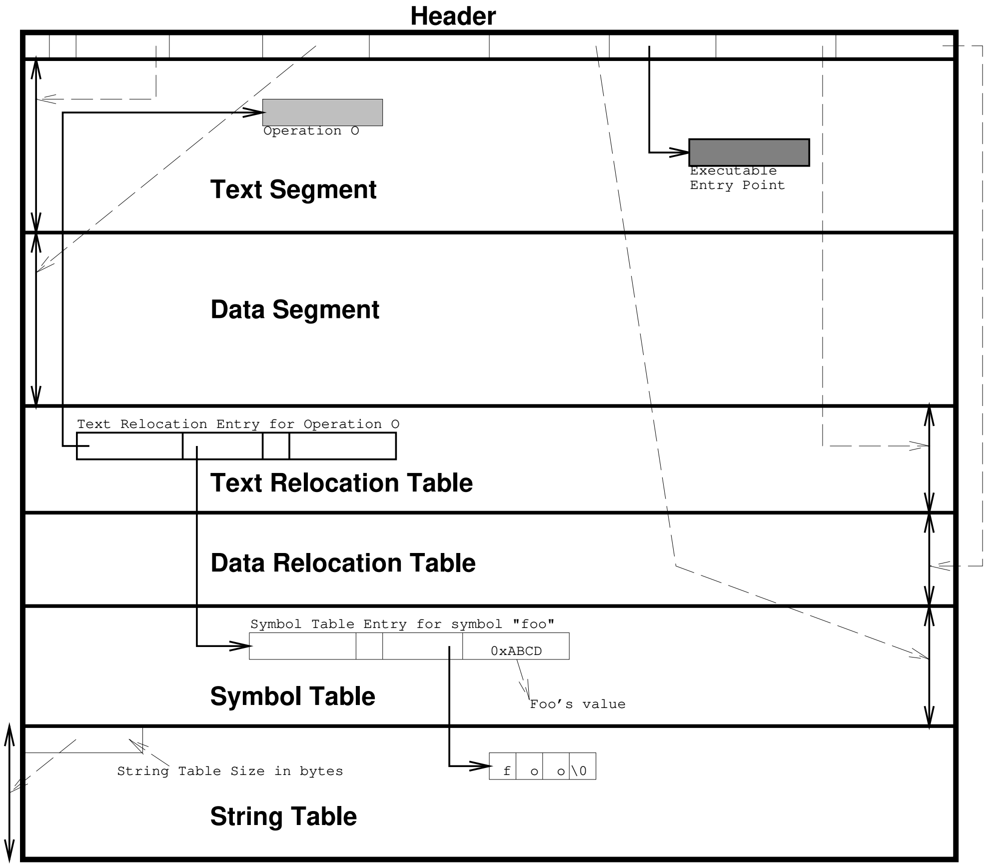

Figure 5-3: Object File Layout

Chapter 6

The M-Machine Linker

The M-Machine linker, `lnk`, is a system tool which links multiple M-Machine object files into a single binary suitable for execution on the M-Machine (or nominally, the machine simulator.) The following section details the design and implementation of the linker, with special attention to operation-patching, and debugging features.

Lnk uses the M-Machine object file format as a specification of input files and outputs single files which conform to that format. Additional constraints placed on executables involve resolution of symbols - all referenced symbols must be have defined values and a valid entry point for execution. The resulting linked executables are generated as debug-format or regular object files, depending on the types of input files given.

6.1 Lnk Usage

Lnk expects a list of object file names as command-line arguments. An optional `-o outputfile` directive specifies the name of the executable. If no executable name is given, lnk aborts the link process. In addition to its capacity as a linker, lnk may be used to extract information about object files. These debugging command-line options are summarized in Table 6.1 and described in more detail in Section 6.5.

To process the set of input object files, the contents of each file is first read into a data structure which contains all relevant sections of the object file and has a

Option	Description
-Di	Disassembles the text segment of each file given on the command line.
-Dis	Disassembles the text segment and skips nops.
-Dd	Displays the data segment of each file on the command line.
-Du	Displays the data segment of each file on the command line, sorted by data type.
-r	Displays the text and data relocation entries for each file on the command line.
-s	Displays the local symbol table of each file on the command line.
-h	Displays the file header.

Table 6.1: lnk debugging command line options

complete, reconstructed symbol table. Details of the data structures and the portable functions employed in their creation are given in section 7.3

6.2 Symbol Resolution

If the input object files are valid, their symbol tables need to be checked for consistency, and have their values patched, if necessary.

6.2.1 Symbol-Table Checking

Since each object-file contains a local symbol table with all symbols which are defined and referenced in that file, it is possible that two object files contain the definition of the same external symbol. Lnk checks each local symbol table for external symbol definitions and determines whether any such symbol is multiply defined. If such a case is encountered, all such multiply-defined symbols are collected and displayed in an error diagnostic before lnk aborts the link process.

Another error condition is when an undefined symbol is referenced (perhaps in more than one file) but is nowhere defined as an external symbol. As in the case of multiply-defined symbols, these symbols are collected and displayed in an error message before lnk quits.

6.2.2 Undefined Symbol Resolution

If the symbol-checking phase is successful, each undefined symbol has one (and only one) external definition, located in the local symbol table of one of the object files being linked. Every undefined symbol is then resolved to the correct externally-defined value. The local symbol table entries of undefined symbols in each object file are updated to reflect these values. By the end of this process, every local symbol table contains only locally-defined or externally-defined symbols.

6.2.3 Symbol Patching

Once symbol-resolution is complete, another pass through all local symbol tables is made to update symbol values to reflect the eventual concatenation of text and data segments - if the text segment of file2 follows that of file1, then all text symbols of file2 are incremented by the size of the file1 text segment. The same is true for data symbol offsets - which must be updated because data segments will eventually be concatenated.

6.3 Text and Data Segment Patching

After all symbols referenced in the text and data relocation have locally or globally defined values, and have been updated to reflect eventual concatenation of text and data segments, operations and data words which have relocation entries attached are updated with values from their local symbol tables. Relocation entries are used to find address offsets into the local object file text or data segment. The relocation symbol index is used to find the appropriate local symbol table entry, and the entry together with the relocation type is used to patch the operation's immediate field or the entire data word with an appropriate symbol value. An instruction pointer is maintained during the patch phase to correctly patch operations which require relative rather than absolute entries since all symbol values stored in the symbol table are absolute.

6.4 Object File Output

In the current implementation of the linker, text and data segment patching occurs during object file output, although this is not strictly necessary. A bogus file header is written to the output executable first, to take up space until a final header is written to the file at the last stage of executable-file creation.

Text segments, with all their operations patched, are written to the new executable output file one after another. After the concatenated text segment is complete, the data segments are concatenated in a similar manner. Finally, new text and data relocation tables are created which reflect updated text and data segment addresses, and symbol values from a combined symbol table with entries comprised of the union of local symbol tables. These relocation tables and the combined string and symbol table are written to the object file next. Finally, a file header with updated values for the concatenated text and data segments, and combined relocation and symbol tables, is written to the beginning of the executable. The new executable follows the M-Machine object file format and may be read by the M-Machine simulator for execution.

6.5 Debugging Features of the Linker

In addition to creating executable M-Machine files, the M-Machine linker also functions as a limited debugging environment. Instead of linking object files to an executable, the linker may be told to display debugging information about these files if they are debug-format object files. The linker can disassemble text segments (displaying pretty-printed operations and imbedded code data words), data segments (outputting data segment address and data words meaningfully printed by type), and complete local symbol tables. In addition, text and data relocation tables with references to local symbols can also be displayed. All of these functions use the portable modules described in section 7.3.3.

6.6 Future Extensions

The M-Machine linker currently does not use the atend field of relocation entries when patching operations or data words. The 4-byte atend field can be used by the M-Machine assembler to patch an integer offset to an unknown symbol value. If this is implemented, the linker will then be required to use this signed long integer value to modify the symbol value before the operation or data word is patched. This modification to the linker would be a simple change, but necessary if the assembler is modified as described.

Since the M-Machine linker uses standard UNIX object file data structures, it can only support text segments that can be addressed by at most 4 bytes. This limits segment and relocation entries to a greater extent than necessary. In the future, it may be necessary to drop these restructions and create a full 64-bit object file format and corresponding linker. The modular nature of the linker code should make this process relatively simple since for the most part, only underlying data structures such as the file header, and relocation and symbol-table entries would need their code modified.

Chapter 7

Project Modularity

Many functions which have been added to the assembler and linker systems have been developed in a modular manner so as to form an abstraction on top of the ISA and the object file format. The M-Machine assembler, linker, and simulator take advantage of this barrier which is maintained by the `mm_opcodes.h` header file, `opbuf_convert` source and header files, and the various modules of the linker code.

The opcodes and opbuf_convert files provide an abstract definition of the instruction set architecture of the M-Machine and define conversion functions from the binary operation format specified in the ISA Guide to a character-based form similar to the massl input format. This source code is already shared by the linker and simulator for pretty-printing decoded operations and filling standard operation data structures for simulation.

The object-file reading, printing, and data-access functions which were written as part of the linker code form a good base upon which other system tools can be built. These functions are defined in the modules `object_read` and `object_unparse`. The underlying data structures which for these modules are defined in `datastruct`. Together, all of these files may be used to access individual components of object files and to display object-file information in a meaningful way. Currently, the M-Machine simulator makes use of the object-read and data-access functions to actually open executable files and extract appropriate values from text and data segments. Additional system tools which examine relocation entries, disassemble the text segment, or display symbol tables may be built with great ease.

7.1 Opcodes Header File

The mm_opcodes.h file is central to the assembler, simulator, and linker. All valid M-Machine operations are described in this file in a static array of structures called *Opcode Dispatch Table Entries*. Each entry contains the massl name of the operation, a description of its operands, and information on the type of binary layout of the operation as required by the hardware. Additional information mainly for simulator and decoding use has also been added. Figure 7-1 contains the C structure definition of the table entry. MM_ACTION is an enumeration with a unique entry for every M-Machine operation. MM_CLASS is an enumeration of classes as given in the ISA document [1]. Each class uniquely identifies the operand layout of an operation's binary representation. The operation arguments field is composed of flags which are ORed in to describe operands to the operation and detailed in table 7.1.

```
typedef struct {
    char *name;                 /* operation name */
    MM_ACTION action;           /* operation action */
    unsigned long opcode;       /* opcode bitstring */

    unsigned long arguments;    /* operation arguments */
    unsigned long opdesc;       /* operation description */
    MM_CLASS opclass;           /* operation class */
} Opcode_Dispatch_Table_Entry;
```

Figure 7-1: Opcode Dispatch Table Entry.

In addition to the opcode definition table, the opcodes header file also contains information on the binary format of all operations. These bits data structures closely follow the naming conventions of the M-Machine ISA document - each operation is assigned an operation class in the opcode table which defines the operation's layout in binary format. For example, class A operations contain a 7-bit field for the second source operand. If the operation allows immediates as the second source operand, those immediates are limited to the two's complement integer range of -64 to 63 (see

Name	Value	Description
A_NONE	0	Operation has no operands (like `nop`.)
I_DEST	1	Destination operand must be from the integer register file
F_DEST	2	Destination operand must be from the floating-point register file
A_DEST	4	Destination may be either floating-point or integer
CC_DEST	8	Destination is a conditional register
I_SRC1	16	Source 1 operand may be an integer register.
F_SRC1	32	Source 1 operand may be a floating-point register
CC_SRC1	64	Source 1 operand may be a cc register
IMME07A	128	Source 1 operand may be a 7-bit immediate.
IMME11A	256	Source 1 operand may be a 11-bit immediate.
IMME18A	512	Source 2 operand may be a 18-bit immediate.
I_SRC2	1024	Source 2 operand may be an integer register.
F_SRC2	2048	Source 2 operand may be a floating-point register
CC_SRC2	4096	Source 2 operand may be a cc register
O_SRC2	8192	Source 2 operand is optional
IMME07B	16384	Source 2 operand may be a 7-bit immediate.
IMME11B	32768	Source 2 operand may be a 11-bit immediate.
F_SRC3	0x010000	Source 3 operand must be a floating-point register.
BANK	0x020000	Bank operand required
LENGTH	0x040000	Length operand required
A_PRE	0x080000	Precondition operand required
A_POST	0x100000	Postcondition operand required
LOCAL_RF	0x200000	Destination int or fp register operand must be in the local register file
LOCAL_CC	0x400000	Destination cc register operand must be in the local cc register file

Table 7.1: Dispatch table entry argument flags

figure 7-2.) Each `bits` field entry corresponds to a field in the operation layout as defined in the ISA document.

The operation class and bits structures are used by the back-end of the assembler when checking immediate ranges of operands and packing operations into their 32-bit binary formats, and by the operation decoder in the opbuf_convert module.

Because the assembler uses the opcodes file when checking for correctness of massl input programs as they are being parsed, it is highly likely that minor changes to the ISA, including changes to operands of operations, additions or deletions to the list of valid operations, or even changes to the encoding of operations in their 32-bit binary format, will only require changes to this opcodes file.

```
typedef struct {
    unsigned pack   : 2;
    unsigned cond   : 2;
    unsigned ccreg  : 4;
    unsigned opcode : 5;
    unsigned imm    : 1;
    unsigned dh     : 2;
    unsigned dr     : 1;
    unsigned dest   : 4;
    unsigned src1   : 4;
    unsigned src2   : 7;
} OpcodeBitsA;
```

Figure 7-2: Bits structure for class A operations

7.2 Disassembly

The `opbuf_convert` source and header files are based on the opcodes file and provide functions for the translation from the M-Machine 32-bit binary operation format to a text-based massl format. These disassembly functions are used in the simulator, assembler, and linker. Specifically, the assembler makes use of this feature as a testing tool - to make sure that input massl programs correctly check against its binary output (if all operations given as input appear on the output in the same format, it is possible to write an automated massl-file generator and assembler-output tester to test the assembler on all valid M-Machine operations.) The linker uses the conversion functions for its debugging features, to display a pretty-printed version of the text segment. The M-Machine simulator uses the conversion functions both for filling internal data structures from the 32-bit binary operation format, and when pretty-printing ranges from the text segment. These translation functions (which decode operations based on their operation class) form an abstraction for the M-Machine instruction set which, like the opcodes file, shields client code from minor changes to the ISA.

7.3 Common Linker Data Structures

The linker module's `datastruct.h` file defines data structures used throughout the linker and the portable functions discussed below. At the heart of the data structures is the *FListNode* structure - a node which can be added to a linked list, or used independently. The FListNode contains fields which can be filled to encapsulate all data stored in an object file, including text and data segments, relocation tables, and symbol tables. Client code deals only with the FListNode structure, providing a data abstraction from the internals of M-Machine object files. The C structure definition of FListNode is given in figure 7-3.

```
typedef struct flist_node {
    char *inFileName;              /* name of the object file which
                                      this node represents.        */
    FILE *inFile;                  /* the object file handle       */
    unsigned char isDebug, isLinked;/* status flags for the file    */
    struct exec header;            /* file header                  */
    codebits      *TextSegment;    /* an array of operations
                                      representing the text segment */
    char          *DataSegment;    /* an array of bytes
                                      representing the data segment */
    unsigned char *TextInfo;       /* tagged bytes for specifying
                                      where codedata is stored in
                                      the text segment.            */
    RelocList     relocation;      /* text relocation entry list   */
    RelocList     datareloc;       /* data relocation entry list   */
    SymList       symbol_table;    /* local symbol table           */
    struct flist_node *next;       /* link to the next node if used
                                      as a node in a list          */
} FListNode;
```

Figure 7-3: FListNode definition

Although the FListNode is the most important data structure, lower-level structure definitions are provided for work with specific sections of object files. datastruct.h defines the *SymNode* to hold symbol-table entry information, and the *RelocNode* for data and text relocation entries. All of these higher-level data structures contain *entry* fields which are pointers to the actual standard C structure definitions such as `reloc_info_sparc` and `nlist` presented in chapter 5.

7.3.1 Object File Reading Functions

Functions in the linker's `object_read` module allow client code to fill an FListNode with all data extracted from an object file, or to read individual object-file components into the FListNode structure. All functions described below expect a valid FListNode pointer, with an *inFile* field which is a handle to an open object file; a new pointer can be created by calling the `FlistNode_create()` function. Functions other than the toplevel `ReadObjectFile` require the FListNode argument to have a valid header field, read from the object file it represents.

ReadObjectFile

ReadObjectFile takes a node as an argument and uses the *inFile* field of the FListNode to read and construct all other object-file related components of the FListNode. It returns 1 if successful and 0 otherwise. After successful call to ReadObjectFile, client code may safely access all components of the file node, including text and data segments, and relocation and symbol tables.

ReadRelocationInformation

This function reads the text and data relocation tables of the open object file specified by the FlistNode's *inFile* field, returning a 1 if relocation tables were successfully read and 0 otherwise.

ReadSymbolTable

ReadSymbolTable reconstructs the object file's symbol table and stores it in the *symbol_table* field of the FListNode. It allocates space for all string names found in the string table and replaces the string-index field each symbol-table entry with a string pointer. ReadSymbolTable returns a 1 if successful and 0 if an error in reading or storing the information occured.

ReadTextSegment

With valid inFile and header fields from its FListNode argument, ReadTextSegment creates and fills the array of operation words in the *TextSegment* field of the FListNode with operations from the object file being read. If code data is encountered, tags for the occurence of these data words are stored in the *TextInfo* field which mirrors the TextSegment and provides tag bytes for every 32-bit operation word. ReadTextSegment differentiates between debug-format and normal object files and can appropriately deal with the debug-information bytes present in debugged text segments. This function returns a 1 if successful and 0 otherwise.

ReadDataSegment

The ReadDataSegment function reads in the entire data segment from the file specified by the FListNode inFile field into the node's *DataSegment* array, which it creates. It returns a 1 on success and 0 on failure.

7.3.2 Data Access Functions

Data access functions defined in the `object_unparse` module are high-level routines used to access data words from an FListNode's data segment or code data from the text segment. Since data word type and code-data location information is stored in relocation entries which are generated only for debugged object files, the FListNode must represent a debug-format object file.

GetData

GetData takes an FListNode and a tag representing the type of data word to filter from the data segment. Tags may request integers, floats, or pointers. This function returns an array of `<address, data word>` tuples, with an array element for every item of the appropriate type which is found in the data segment.

GetCData

GetCData works almost identically to the GetData function, but returns words from the object file's text segment, instead of data segment.

7.3.3 Printing functions

A set of functions to pretty-print internal FListNode data structures to streams are provided in *Print* and *Display* functions, implemented in the `object_unparse` module. All Print functions take an FListNode as an argument and send information to the standard output stream (stdout). All Display functions take a file stream and low-level data structure pointer and send output to that file stream.

PrintDataSegment

PrintDataSegment determines whether the object file represented by the FListNode argument has debug information and either displays the contents of the data segment as 32-bit words or fully-tagged 8- 16- 32- and 64-bit data types. If debug information is missing, a listing of address and 4-byte hexadecimal word is printed since there is no way to tell which data types the bytes in the data segment represent.[1] If debug information is present, however, the data segment relocation entries are used to locate all data words in the data segment so that their native form can be displayed (integers, characters, strings, floats, and M-Machine pointers are supported.)

PrintSortedDataSegment

This function will only work with debugged object files, and prints the contents of the data segment sorted on three data types - 64-bit integers, 64-bit floats, and 64-bit M-Machine pointers.

[1] Since the native word size of the M-Machine is 64 bits, this should be changed to print eight bytes.

PrintTextSegment

PrintTextSegment displays the operations and code data words of the FListNode argument's text segment structure. This function works best when used on debbuged object files, since the presence of code data words cannot be detected for undebugged files and may throw operation-disassembly off.

DisplayRelocation

DisplayRelocation takes a stream, a relocation table, and a symbol table as arguments and sends a pretty-printed representation of the contents of the relocation table (complete with symbol name found in the symbol table if the relocation entry references a symbol) to the stream. For each relocation entry, Display relocation prints the address of the operation for which the entry was generated, the type of relocation entry (such as 11-bit abolute, or 18-bit relative), and the index and name of the symbol referenced, if known.

DisplayHeader

Given a stream and a header data structure (such as as the header component of an FListNode), DisplayHeader prints relevant information on segment and relocation sizes to the stream.

DisplaySymbolTable

DisplaySymbolTable takes a stream and symbol table structure as arguments and prints the contents of the symbol table to the stream in a form similar to that used by the UNIX *nm* tool. The function displays the name of the symbol, its value, and a byte which describes its type - U for undefined, T for external text symbol, t for local text symbol, A for external absolute symbol, a for local absolute symbol, D for external data symbol, and d for local data symbol.

Chapter 8

Future work

Although the system tools presented in this document provide the capability for writing system and application code for the M-Machine, further tool development can greatly aid the software project.

In particular, a loader tool would be most useful for introducing a segmented memory model to the M-Machine runtime system. Instead of maintaining monolithic text and data segments, a loader tool, with the aid of a modified linker and object file format, could partition multiple text and data segments across a segmented virtual address space and generate valid segment pointers to reference these sections.

8.1 Distribution and Replication of Code

Further loader development with this new multi-segment model would allow distribution and replication of code across the nodes of the M-Machine. This enhancement would require a few more modifications to the object file format and added keywords in the massl specification to support node naming so that massl programmers could specify which code segments should be placed on which nodes.

In addition, since user programs will be assembled and then linked with system routines, support for multiple initialized data segments would result in protection of system data space by insuring that user programs cannot accidentally overwrite system data. This is accomplished by separating user and system data segments and only allowing system routines to access system data segment pointers. In this way, even if a user-level program adds offsets to its data segment pointer which extend beyond the end of its data segment definition, there will be no risk of that pointer accidentally reaching system address space.

8.2 Uninitialized Data

Although support in the current implementation of the assembler exists for maintaining an uninitialized data segment, the massl specification does not yet support the `bss` segment. Further, the linker does not recognize uninitilized data segments and therefore does not perform patching of `bss` symbols to account for the concatenation of the uninitialized segments in the flat address space. These modifications would be simple to implement, however, since the code for updating symbol tables already deals correctly with text and initialized data segments. There is nothing special about the `bss` segment in itself, aside from the fact that no operations or actual data are placed there as assembly-time. Most of the work, therefore, would involve the addition of `bss` support in creating relocation entries for text and data words which refer to `bss` symbols, and the addition of `bss` references in the massl specification.

Appendix A

Linker Code

This chapter contains the C code for the M-Machine linker. The code includes support for multiple-data segment object and executable files which were not specified in this document. The linker is fully backward-compatible with the object-file formats as described in this document.

The code following this page is the source code for:

- datastruct.c
- datastruct.h
- lnk.c
- lnkdefs.h
- object_read.c
- object_read.h
- object_unparse.c
- object_unparse.h
- object_write.c
- object_write.h
- patch.c
- patch.h

Appendix B

Opcodes and Object Header Files

This chapter contains source code for the mm_opcodes.h and mm_object.h C header files which define the machine architecture and certain object-file specifications.

Bibliography

[1] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The MAP Instruction Set Reference Manual v1.1. MIT Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, February 1994.

[2] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly and Associates, Sebastopol, California, second edition, October 1992.