Blitz-64:

Assembler, Linker, and Object File Format

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

3 November 2022

This document describes the following tools:
>   **asm** — The Assembler
>   **link** — The Linker
>   **dumpobj** — A tool to display object files in human-readable form
>   **createlib** — A tool to create object library files

In addition, it describes the format and layouts of:
>   **.o** — Object files
>   **.lib** — Library files
>   **a.out** — Executable files

This document gives the information needed to use these tools. It also provides some details about their implementation and algorithms.

Table of Contents

Chapter 1: Introduction

Assembly Language

The assembler is a tool which will translate programs written in assembler (or "assembly language") into binary machine code. Machine code can be loaded into memory and executed. Machine code consists of a sequence of binary bits and cannot be practically created or deciphered by humans.

Assembly code is a human-readable notation in which to specify **machine code.**

Although high-level programming languages strive to be platform independent, the opposite is true of assembly language. Each processor has a unique assembly language tailored to its design. There is much similarity in the assembly languages for different machines, but there is also significant difference.

This document describes the assembly language for the Blitz-64 processor core. It assumes that you have familiarity with the Blitz-64 Instruction Set Architecture (ISA). The Blitz-64 architecture is described in the following document:

"Blitz-64: Instruction Set Architecture Reference Manual"

Programming in assembly language is an acquired taste and should not be attempted by beginning programmers. Assembly programming requires an enormous attention to detail and an extremely high degree of conscientiousness, commitment, and precise logical thinking. The resulting programs are totally non-portable. Merely getting an assembly program to work on a different model in the same processor line is non-trivial.

Assembly language and the skill to code in assembler is important for several reasons.

First, there are often tasks that simply cannot be done in high-level languages and the code must be written in assembler. Assembly code is required for operating

systems kernels and for accessing certain specialized aspects of the hardware that cannot be addressed in a high-level language. Although much work has been done to add capabilities to high-level languages to minimize the amount of assembly code, some assembly code is required.

Second, high-level languages must be compiled to run on physical hardware. (Here we speak of compiled languages (like "C" and "C++") and not of interpreted languages (like just about every other modern language, including Java, JavaScript, Python, Perl, etc.). This means the source code must ultimately be translated into the bit patterns recognized by the intended target hardware.

The typical approach is for a compiler to translate the source code into assembly code. In a second step, the assembler tool is used to translate the assembly program into machine code. While there are many approaches, this approach works well since it breaks the task into two smaller, tasks: translation into assembly code, followed by translation into machine code. It removes many of the hardware details from the compiler and also permits the compiler writers to determine whether a compiler is working properly and producing the correct output.

Third, understanding the assembly language for a processor is a requirement for anyone who wants to understand and improve the runtime execution performance of programs written in high-level code. Those programmers seeking to maximize performance need to understand assembly language so they can spot inefficient code sequences and determine whether the compiler is producing the best code.

Fourth, assembly language programming is required for new, state-of-the-art processors for which no high-level languages are available. Assembly language programming may also be required for obscure or specialized processors for the same reason.

Finally, there are a few programmers who actually enjoy programming in assembler.

The Linker

The assembler tool translates assembly source code into machine code. However, programmers break large programs into pieces which we will call "**modules**". Generally, a small number of pieces are combined to produce an executable program. For example, one piece might contain a number of mathematical support functions

(like "sin" and "sqrt"). This might be combined with the "main" function of a program to produce an executable. Obviously, the mathematical support functions are written separately and reused in many different programs.

In practice, there is a tremendous number of code modules. The sharing and re-use of modules is critical.

The assembler tool takes as input a single source code file (containing the code and data for a single module) and produces a "**object file**". For example, a large program consisting of 5 modules will require the assembler to be run five times, once for each module, producing five object files.

The linker tool is named "**link**". (We chose a name more meaningful than the traditional name used in Unix/Linux, which was "ld".)

The linker tool is used to combine the object files and produce a single executable file. In other systems, the linker also takes as input some sort of textual script or program to give the linker instructions. But in the Blitz-64 approach, such additional information is not needed. The input to the Blitz-64 linker consists of only the object files.

The executable file is stored in a file and, when the program is to be run, the operating system will read this file (understanding the format of executable files) and will load the bits into memory just prior to beginning execution.

The primarily programming language for the Blitz-64 system is **KPL (Kernel Programming Language)**. While almost every other computer uses the "C" language — a language from the early 1970s — as the core language upon which all the remaining software is constructed, Blitz-64 does not support "C". Blitz-64 takes the radical approach of not supporting *any* legacy software.

KPL supports a concept called "**packages**". Each package is separately compiled into an assembly language program. In this way, KPL works like program development in "C". For example, five packages will be separately compiled, yielding five different assembly files.

Each assembly file produced by the compiler will assembled separately. Additional modules that have been hand-code in assembly language will also be assembled by the assembler tool. Finally, the linker tool will be run to combine all the object modules into a single executable file.

Additional Tools

A separate tool, called "**dumpobj**", is also provided. It can be used to look at the contents of an object file or an executable file. These files are not text files and are not meant to be human readable. The "dumpobj" tool merely prints out information about the file contents in a format that humans can read. This tool is not normally needed in program development, so it is used less often.

Several object modules can be combined into a "**library**". The linker can consult a library file to locate modules as needed by the program being linked. The "**createlib**" tool is used to create library files.

Another tool, called "**hexdump**", is also provided to look at the contents of any file. The "hexdump" tool prints out the contents of any file in hex. It also prints out any ASCII characters. The "hexdump" tool is useful in determining what exactly is in a file.

Tool Names and File Extensions

The names of the tools are:

kpl	The KPL compiler tool
asm	The Assembler tool
link	The Linker tool
dumpobj	Tool to display info about object and executable files
createlib	Tool to create a library file
hexdump	Tool to display the contents of any file in hex

Blitz-64 uses file extensions to suggests the nature or type of material in a file, and the file extensions are similar to other systems:

.c	KPL source code ("c" for "code")
.h	KPL header files
.s	Assembly programs
.o	Object files
.lib	Library files

Executable programs usually do not have extensions, but when no filename is supplied, the traditional default name of "**a.out**" is used.

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name are used. The document history is:

Date	Author
27 October 2018	Harry H. Porter III <document created>
28 May 2019	Harry H. Porter III
19 March 2022	Harry H. Porter III
18 October 2022	Harry H. Porter III
3 November 2022	Harry H. Porter III <current version>

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> *You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.*

Any material lifted should be referenced.

Program Versions

In the Blitz-64 project, version numbers are not used for programs and documents. Instead, dates are used. This document describes the following programs.

By comparing dates, you can determine whether this document matches the tools you are using or, if not, which is more recent.

Tool	Version Described Here	Coding Status
asm	*< same date as this document >*	Completed
link	*< same date as this document >*	Completed
dumpobj	*< same date as this document >*	Completed
createlib	*< same date as this document >*	Completed
hexdump	*< same date as this document >*	Completed

Chapter 2: Assembler Syntax

An Example Program

```
################
#
# MyFun
#
# This function does such and such. It uses…
#
################
        .begin
        .align  4
        .export MyFun
MyFun:  stored  0(sp),r2        # Save registers
        stored  8(sp),r3        # .
loop:                           # LOOP
        loadb   r3,0(r2)        #   IF r4>*r2 THEN
        ble     r4,r3,endif     #   .
        sub     r1,r5,r3        #     r1 := r5-(*r2)
endif:                          #   ENDIF
        addi    r2,r2,1         #   r2++
# … etc …
        jump    loop            # ENDLOOP
```

A Second Example

The following example illustrates a number of different instruction and operand combinations.

This code assembles without error as a standalone source file, although taken as a whole, the code is obviously not a program to be executed.

```
# Examples showing different pseudo-ops

      .begin       startaddr=0x0456,executable,writable
a:    .byte        0x12              # allocates 1 byte
b:    .halfword    0x1234            # allocates 2 bytes
c:    .word        -1234 + 0x5d      # allocates 4 bytes
d:    .doubleword  y+246             # allocates 8 bytes
e:    .float       -123.456e78       # allocates 8 byte double float
s_1: .string       "hello\n"         # allocates N bytes
      .export      MyLabel           # make symbol avail to other mods
      .import      OtherLabel        # use a symbol from other module
x:    .skip        100               # skips over bytes, w/ zero-fill
      .align       8                 # inserts 0x00 bytes as necessary
y:    .equ         100               # Defines symbolic constant

# Examples showing different operands

      sysret                         # Format A-0: <no operands>
      checkw       r1                # Format A-1: Reg1
      sextw        r7,r1             # Format A-2: RegD,Reg1
      add          r7,r1,r2          # Format A-3: RegD,Reg1,Reg2
      alignd       r7,r1,r2,r3       # Format A-4: RegD,Reg1,Reg2,Reg3
      csrswap      r7,csr_status,r2  # Format A-7: RegD,CSRReg1,Reg2
      csrread      r7,csr_status     # Format A-8: RegD,CSRReg1
      getstat      r7                # Format A-9: RegD
      addi         r7,r1,-456        # Format B-1: RegD,Reg1,immed-16
      load.d       r7,250(r1)        # Format B-2: RegD,immed-16(Reg1)
      checkaddr    r7,5              # Format B-3: RegD,Reg1,immed-3
      syscall      123               # Format B-4: immed-10
      slli         r7,r1,63          # Format B-5: RegD,Reg1,immed-6
      csrset       csr_status,0x03   # Format B-6: CSRReg1,immed-16
      store.b      123(r1),r2        # Format C-1: immed-16(Reg1),Reg2
      b.eq         r1,r2,+8          # Format C-2: Reg1,Reg2,immed-16
      jal          lr,-12            # Format D-1: RegD,immed-20

# Examples showing different synthetic instructions

      movi         r7,0x123456789abcdef0    # Format S-1
      blt          r1,r2,MyLabel            # Format S-2
      call         MyFun                    # Format S-3
```

```
        loadw        r7,MyVariable              # Format S-4
        loadw        r7,my_offset(r1)           # Format S-4
        storeb       MyVariable,r2              # Format S-6
        stored       my_offset(r1),r2          # Format S-7


MyVariable:    .doubleword    0
MyLabel:       jump           MyLabel
my_offset:     .equ           100
```

Terminology, Notation, and Basic Concepts

- **Byte**   8 bits
- **Halfword**   16 bits   2 bytes
- **Word**   32 bits   4 bytes
- **Doubleword**   64 bits   8 bytes

Binary values are frequently specified in hex.

	number of bytes	number of bits	example value (in hex)
byte	1	8	A4
halfword	2	16	C4F9
word	4	32	AB12CD34
doubleword	8	64	0123456789ABCDEF

To clarify and prevent confusion, hex numbers are often preceded by "0x". For example:

```
0x1234
```

As in most other computers, main memory is **byte addressable**, which means that every byte in memory has a unique address.

Main memory is **Big Endian**, which means that the most significant byte of a value is stored first, at the starting address. For example, if the value 0x1234 is stored in memory at address X, then the first byte 0x12 will be in location X and the second byte 0x34 will be in location X+1. This means that the bytes are not rearranged.

Many other computers (including x86) use the opposite convention, Little Endian, which reverses the byte order.

The notation **[n:m]** is used to identify bits. For example, [63:60] means the most significant (MSB) 4 bits in a doubleword.

We use the term **KiByte** to mean 1,024 (i.e., 2^{10}). We avoid using the term KByte (i.e., 1,000 = 10^3). Likewise, we use **MiByte** and **GiByte** instead of MByte and GByte.

		Hex Value	Decimal Value
KiByte	2^{10}	400	1,024
MiByte	2^{20}	10_0000	1,048,576
GiByte	2^{30}	4000_0000	1,073,741,824

The Blitz-64 processor has certain alignment requirements. A **halfword aligned address** is an even number and, when represented in binary, ends with a 0 bit. A **word aligned address** is a multiple of 4 and ends with 00. A **doubleword aligned address** is a multiple of 8 and ends with 000.

The Blitz-64 is "strongly 64 bits", which means that all arithmetic is done with 64 bits. The processor has minimal support for legacy sizes such as 8, 16, or 32 bits.

Integers are represented with **signed, two's complement** values.

	Size in bits	Range of values
byte	8	-128 ... 127
halfword	16	-32,768 ... 32,767
word	32	-2,147,483,648 ... 2,147,483,647
doubleword	64	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807

Sign-extension enlarges an integer represented in signed two's complement binary. For example, sign-extending the halfword 0x8C32 to a doubleword yields the following result:

```
0xFFFFFFFFFFFF8C32
```

For large numbers, we often add underscores every 16 bits, to prevent confusion[1]:

```
0xFFFF_FFFF_FFFF_8C32
```

The underscore is in assembler code, as well as documentation and comments.

Size reduction (e.g., from 64 to 32 bits) results in an "overflow" error whenever a the value exceeds the range of the smaller size..

Tokens and Lexical Issues

Identifiers may contain letters, digits, and underscores. For example:

```
MyLabel
_entry
lab_23_
```

Identifiers must begin with a letter or underscore. Case is significant.

Identifiers are limited in length. Currently the limit is set to 1,000 characters. [This limitation is hardcoded into the assembler tool and requires recompiling "asm" to change.]

Identifiers may contain only ASCII characters. By "letters and digits", we mean one of the the 26+26+10 characters in { a ... z A ... Z 0 ... 9 }.

Keywords   The assembler recognizes a number of special keywords which otherwise resemble identifiers. These keywords may not be used for identifiers.

Although the period character is not allowed in identifiers, several keywords contain the period character. For example:

```
load.w
.begin
```

---

[1] Sometimes a comma is used as a separator. The Blitz-64 tools recognize and accept underscores, but not commas.

The following classes of keywords are recognized:

	Examples
Opcodes for machine instructions	`add`, `load.w`, `syscall`, …
Opcodes for synthetic instructions	`mov`, `call`, `bgt`, …
Pseudo-ops	`.begin`, `.equ`, `.string`, …
Registers	`r0`, … `r15`, `t`, `sp`, `lr`, …
CSR Registers	`csr_status`, `csr_cycle`, …
Misc. keywords	`page`, `startaddr`, …

An **integer value** may be specified in decimal or in hex. If specified in decimal, the integer value must lie between 0 and 9,223,372,036,854,775,807 (i.e. 2^{63}-1). Commas are not allowed.

Integer values may be given in **hex notation**, and must be preceded by "0x". For example:

```
0x1234abcd
0x1234ABCD      ← case does not matter
```

A hex constant may optionally contain underscore characters, which may be used to improve readability. An underscore should be placed after every fourth hex digit, but this is not enforced.

```
0x4d03_55e2_3a8e_47a9        ← recommended style
0x4d_0355e23a___8e47a9       ← also allowable
```

Every integer constant specifies a 64-bit signed value, regardless of how many digits appear.

Integer values can be specified in either decimal or hex. Hex notation and decimal notation are fully interchangeable. Anywhere a decimal value can be specified, a hex value can be used instead, and vice-versa:

```
123
0x7b   ← equivalent value
```

Decimal can only be used for positive values but a preceding minus sign can be used to form an expression so, effectively, negative numbers can be specified. For example:

```
–123
```

Unary negation can be applied to any integer, whether specified in decimal or hex, so the following all represent the same 64-bit value:

```
–4660
–0x1234            ← identical value
0xffffffffffffedcc  ← identical value
```

A number given in hex is <u>not</u> sign-extended by the assembler.

```
0xc8a4             ← equal to +51,364
0x000000000000c8a4  ← identical value
```

If a negative number is specified in hex, sign-extension to 64-bits is required. For example

```
0xc8a4
```

is equal to -14,172 as a signed, 16-bit value. To specify this value in hex, the leading 1 bits must be given. This value can be specified in any of the following ways:

```
0xfffffffffffffc8a4
–14172
–0x375C
```

Note that the following values are equal:

```
0xffffffffffffffff
–1
```

If a hex number has fewer than 16 hex digits, it will be interpreted as a positive number. Be careful:

```
0xffffffffffffffff    ← missing an "f"
1152921504606846975  ← identical value
```

Note that the most negative 64 bit value may not be specified in decimal since the positive portion exceeds the limit for positive numbers. This value must be specified in hex:

```
-9223372036854775808        ← not allowed
0x8000_0000_0000_0000       ← use this instead
```

Strings are written using double quotes. For example:

```
"Hello, world"
```

The following escape sequences are allowed in strings:

```
\0   \a   \b   \t   \n   \v   \f   \r   \e   \"   \'   \\   \xHH
```

where HH represents any two hex digits. The escape sequences have the traditional meanings:

```
\0      0x00    NULL
\a      0x07    BELL (alert)
\b      0x08    BS (backspace)
\t      0x09    HT (tab)
\n      0x0A    LF (linefeed, newline, NL)
\v      0x0B    VT (vertical tab)
\f      0x0C    FF (form feed, new page)
\r      0x0D    CR (carriage return, enter)
\e      0x1B    ESC (escape)
\"      0x22    double quote character
\'      0x27    single quote character
\\      0x5C    backslash character
\xHH    0xHH    arbitrary byte (where H is any hex character)
```

In a string constant, we make a distinction between the string "**source**" characters and the string "**value**". For example, in the following string

```
.string  "\n"
```

there are two source characters '\' and 'n'. In the string value, there is only one byte, namely 0x0a.

The string value is a sequence of zero or more bytes, and there is no constraint on what byte values or sequences are allowed.

However, there are constraints on the string source characters.

The string source may not include any ASCII control characters directly. Instead, the programmer may use escape sequences, such as \0, \n, \t, etc.

One implication is that strings may not contain newlines directly. In other words, a string may not span multiple lines. Use \n or \r within the string source instead.

The **.s** source file is a "text" file encoded in UTF-8. Non-ASCII characters (as in the next example) are allowable in comments and within string source (between the quotes in a string constant). Non-ASCII characters are not allowed anywhere else.

Any Unicode character can appear in a string source, with the aforementioned prohibition on ASCII control characters (i.e., codepoints within 0x00 ... 0x1F and 0x7F).

The string source will be translated into a value — a sequence of bytes — encoded in UTF-8.

Consider the following string:

```
str:    .string     "∉"
```

This string source contains 1 character, a Unicode character called "NOT AN ELEMENT OF".

The UTF-8 encoding of this character requires three bytes. Thus, this string value consists of three bytes.

The following is exactly equivalent. Both place exactly the same bytes at location "str".

```
str:    .string     "\xE2\x88\x89"
```

[See the document titled "An Overview of Unicode", which describes UTF-8.]

String values are limited in length. This limit is identical to the length limit for identifiers.

The operand of the **.string** pseudo-op must be a string.

In addition, a string may be used in an expression in place of an integer. However, in this case, the string value must have exactly 8 bytes. The characters will be used to construct an 8 byte (i.e., 64 bit) integer value.

(More precisely, the UTF-8 encoding of the characters will be used.)

For example, the following are four ways to represent the same 64-bit value:

```
"Hello!\n\0"
"\x48\x65\x6C\x6C\x6f\x21\x0A\x00"    ← identical value
0x48656C6C6f210A00                    ← identical value
5216694956355291648                   ← identical value
```

Character constants are given using single quotes. For example:

```
'q'
```

There must be exactly one character, or an escape sequence representing a single byte. The same escape sequences as used in strings are allowed.

A character constant can be used any place an integer is allowed and is equivalent to an integer value between 0 and 255 (i.e., between 0x0000000000000000 and 0x00000000000000FF).

For example, the following are equal and can be used interchangeably:

```
'\n'
10
0x0A
```

Since a character constant is always exactly one byte, only ASCII characters are permitted, not arbitrary Unicode characters.

A **floating-point constant** is used to specify a double precision (8-byte) floating point value. To be differentiated from an integer constant, the value must have either a decimal point or an exponent. The exponent is signified by either "E" or "e".

Examples of floating point constants:

```
.float   123.456
.float   –3.4E–21
.float   +4.5e+21
```

Floating point constants are used in the **.float** pseudo-op, and nowhere else.

Comments begin with the hash or pound symbol (#) and extend thru end-of-line.

Punctuation symbols The following have special meaning:

,	separates operands
:	follows labels
=	used for keyword operands in .begin pseudo-op
(	expression grouping
)	expression grouping
+	addition and unary plus
–	subtraction and unary minus
*	multiplication
/	integer divison
%	remainder after division
&	bitwise AND
|	bitwise OR
^	bitwise XOR
!	bitwise NOT
<<	shift left logical
>>	shift right logical
<<<	shift left arithmetic
>>>	shift right arithmetic

White space The assembler parses each line by first identifying lexical tokens and removing comments. Lexical tokens may be separated by "white space", which is defined as spaces and tabs.

End-of-line The EOL character is treated as a token, not as white space; the EOL is significant in syntax parsing. The source file can use \n (i.e., NEWLINE, 0x0A) or \r (i.e., RETURN, 0x0D) to indicate the EOL; either will work.

Instruction Syntax

Each line in the assembly source file must have the following syntax:

  [**label :**] [**opcode operands**] [**# comment**] EOL

(The brackets indicate optional material.)

The label is optional. It need not begin in column one. It must be followed by a colon token. A label may be on a line by itself. If so, it will be attached to the next thing following it. In other words, a label will stand for the address of an instruction and the instruction can be given on the same line, or on the following line.

The opcode must be a legal Blitz-64 instruction or a pseudo-op. The opcode is always lowercase.

Operands are separated by commas. The exact syntax of the operands is determined by the instruction opcode. Some Blitz-64 instructions take no operands while some instructions require several operands.

A comment is optional and extends to the end of the line if present.

Each line is independent. The end-of-line (EOL) is treated as a separate token, not as white space (as occurs in many programming languages). Every instruction must be on only one line, although lines may be arbitrarily long.

Assembler pseudo-ops have the same syntax. Some permit labels and others forbid labels.

The following formatting and spacing conventions are recommended:

- Labels should begin in column 1.
- The op-code should be indented by 1 tab stop.
- The operands, if any, should be indented by 1 additional tab stop.
- Each Blitz-64 instruction should be commented.
- The comment should be indented by 2 additional tab stops.
- A single space should follow the # comment character.
- Block comments should occur before each routine.
- Comments should be indented with 2 spaces to show logical organization.

Here is an example of the recommended style for Blitz-64 assembly code. (The header line shows standard tab stops.)

```
           t          t          t          t          t          t
################
#
# MyFun
#
# This function does such and such. It uses…
#
################
        .begin
        .align  4
        .export MyFun
MyFun:  stored  0(sp),r2        # Save registers
        stored  8(sp),r3        # .
loop:                           # LOOP
        loadb   r3,0(r2)        #   IF r4>*r2 THEN
        ble     r4,r3,endif     #   .
        sub     r1,r5,r3        #     r1 := r5-(*r2)
endif:                          #   ENDIF
        addi    r2,r2,1         #   r2++
# … etc …
        jump    loop            # ENDLOOP
```

Of course assembly code produced by a compiler will probably not be commented or formatted so nicely.

Register Names

Register names must be in lowercase. Several registers have two names. The programmer can use either name. Generally, the alternate name is recommended.

	Alternate Name	Function
r0		Zero
r1		Argument 1 / Return Value
r2		Argument 2
r3		Argument 3
r4		Argument 4
r5		Argument 5
r6		Argument 6
r7		Argument 7
r8	**t**	Temp register, used by assembler/linker
r9	**s0**	Work reg (caller-saved)
r10	**s1**	Work reg (caller-saved)
r11	**s2**	Work reg (caller-saved)
r12	**tp**	Thread data pointer
r13	**gp**	Global data pointer
r14	**lr**	Link register
r15	**sp**	Stack pointer

Register "r0" is the zero register. Its value is always read as zero and writes are ignored. The programmer often uses the zero register as a destination when the goal is to discard a value.

Several instructions require the name of a **Control and Status Register (CSR)**.

There are 16 CSR registers. Their names must be written in lowercase.

		<u>**Description**</u>
0	**csr_version**	Version of the BLITZ-64 architecture ISA
1	**csr_prod**	Product Identifier
2	**csr_core**	Core number
3	**csr_instr**	Instruction counter (Reset upon power-on-reset)
4	**csr_cycle**	Cycle counter (Reset upon power-on-reset)
5	**csr_timer**	Time of next interrupt, in cycles
6	**csr_status**	System status register
7	**csr_stat2**	Previous System Status Register
8	**csr_trapvec**	Pointer to page table root node
9	**csr_pgtable**	Pointer to page table root node
10	**csr_prevpc**	Previous PC (for trap handler)
11	**csr_cause**	Trap code, indicating which trap just happened
12	**csr_bad**	Offending instruction (for Kernel Exception: cause)
13	**csr_addr**	Offending Virtual Address
14	**csr_ptr**	Ptr to Process Control Block (& reg save area)
15	**csr_temp**	Temp work register

Machine and Synthetic Instructions

The Blitz-64 instructions are documented separately in

"Blitz-64: Instruction Set Architecture Reference Manual"

For each instruction, that document describes:

- what operands are used
- what each instruction does when executed
- how each instruction is represented in machine code

Each line in the assembly program contains either:

- A machine instruction,
- A synthetic instruction, or
- A "pseudo-op" instruction

(In addition, some lines will contain only labels or comments. Blank lines can be used to improve readability.)

By "**machine instruction**", we mean the line contains the human-readable assembly code form of an instruction implemented directly by the Blitz-64 hardware.

The opcode (such as "addi") determines exactly which machine instruction is intended and exactly which operands are required. Each opcode corresponds to exactly one machine code instruction, so there is a one-to-one correspondence between machine opcodes (like "addi") and machine instructions.

For each machine instruction, there is exactly one allowable syntax for the operands. In the case of "addi", two registers and an immediate value (in that order and separated by commas) are required:

```
addi  r3,r6,1234
```

The assembler will translate each machine opcode into a single 32-bit machine code. For example, this instruction will be translated to:

```
0x0104d263
```

You can understand this instruction as follows:

```
01      machine opcode for "addi"
04d2    hex representation for 1,234
6       register "r6"
3       register "r3" (the destination)
```

A "**synthetic instruction**" does not correspond to exactly one machine instruction. Instead, the assembler will translate synthetic instructions into machine instructions that perform the desired operation.

For example, the following synthetic instruction:

```
neg   r7,r3          #  r7 ← −(r3)
```

will be translated into this machine instruction:

```
sub   r7,r0,r3       #  r7 ← 0−r3
```

It will assembled as if the programmer had used the subtract instruction instead. (Note that register "r0" always contains the the value zero.)

The translation of a synthetic instruction will usually be to a single machine instruction. However, some synthetic instructions will require several machine instructions and may require as many as four instructions.

For example, the following synthetic instruction:

```
    movi      r1,0x1122334455667   #  r1 ← very large value
```

will be translated into the following sequence of three machine instructions:

```
    upper20   r1,0x11223
    shift16   r1,r1,0x3445
    xori      r1,r1,0x5667
```

Assembler Pseudo-ops

A pseudo-op looks very similar to an instruction since it has an opcode and operands.

Pseudo-ops can be easily recognized because they all begin with a period. The pseudo opcodes are:

```
    .byte
    .halfword
    .word
    .doubleword
    .float
    .string
    .skip
    .align
    .export
    .import
    .equ
    .begin
```

(In addition, there are several pseudo-ops associated with debugging; these are listed and discussed in a later chapter.)

The period is part of the opcode keyword. Spaces are not allowed after the period.

Machine and synthetic instructions are assembled into binary codes that, when executed, tell the processor what to do. Pseudo-ops are not translated into machine instructions to be executed at runtime. Instead, pseudo-ops are used to tell the assembler what to do and how to produce the object code.

Pseudo-ops are sometimes called "**assembler directives**".

.byte, .halfword, .word, .doubleword

The **.byte**, **.halfword**, **.word**, and **.doubleword** pseudo-ops are used to allocate 1, 2, 4, and 8 bytes, respectively. For example:

```
MyVar:    .doubleword    654321   # Allocate and initialize 8 bytes
```

A single operand (which is an expression) is required. The expression specifies an integer value which will be placed in memory before execution begins.

```
another: .doubleword    (789*5)<<6   # equal to 0x000000000003DA40
```

The expression may include values written in decimal or hex, as well as symbolic constants. The expression appearing in the operand field may use:

(	expression grouping
)	expression grouping
+	addition and unary plus
–	subtraction and unary minus
*	multiplication
/	integer divison
%	remainder after division
&	bitwise AND
|	bitwise OR
^	bitwise XOR
!	bitwise NOT
<<	shift left logical
>>	shift right logical
<<<	shift left arithmetic
>>>	shift right arithmetic

The expression will be evaluated and the value will be computed by the assembler and not at "run-time".

All expression evaluation will be performed using 64 bit signed integers. If the final value fails to be within the allowable range for the pseudo-op, the assembler will issue an error message.

	Size in bytes	Range of values
.byte	1	-128 ... 127
.halfword	2	-32,768 ... 32,767
.word	4	-2,147,483,648 ... 2,147,483,647
.doubleword	8	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807

If a label precedes a pseudo-op or instruction, that symbol will be associated with the address of the thing that follows. (More precisely, the symbol will be associated with the address of *the first byte of* the thing that follows.) The label may appear on the same line or on the preceding line.

For example, this

```
myVar:    .doubleword       0x0123456789abcdef
```

is equivalent to:

```
myVar:
          .doubleword       0x0123456789abcdef
```

The requirements for alignment in Blitz-64 are discussed elsewhere. In short, data should be properly aligned:

- Halfword data should be halfword-aligned
- Word data should be word-aligned
- Doubleword data should be doubleword-aligned

There is no alignment requirement for byte-sized data.

If the data is improperly aligned, an exception will be generated at runtime and the instruction will invoke an emulation routine. There will be a very heavy performance penalty for this. Therefore, the programmer should strive to ensure that all variables are properly aligned.

The **.align** instruction can be used for this purpose. One approach is to proceed each data variable with an **.align** instruction:

```
var1:     .byte             0x01
          .align            2
var2:     .halfword         0x0123
          .align            4
var3:     .word             0x01234567
          .align            8
var4:     .doubleword       0x0123456789abcdef
```

A simple programming trick is to place all doubleword data first, then all word data, then all halfword data, and finally all byte data. Only a single .align is required at the beginning:

```
        .align          8
var4:   .doubleword     0x0123456789abcdef
var3:   .word           0x01234567
var2:   .halfword       0x0123
var1:   .byte           0x01
```

The **.halfword**, **.word**, **.doubleword**, and **.float** instructions may be used at unaligned locations. The assembler will not issue warnings.

For the value of the **.byte**, **.halfword**,**.word**, and **.doubleword** pseudo-ops, the programmer may use either absolute value or may use addresses and symbols, which may be defined in the same file or imported from another **.s** source file. For example:

```
        .doubleword     MyLabel+4
        .doubleword     ExternSymbol
```

In such cases, the linker will not be able to compute the value and will defer to the linker, which will compute and fill in the final values.

The linker computes all values using 64 bits and may compute any value within this range. However, for **.byte**, **.halfword**, and **.word** pseudo-ops, there may be insufficient space to contain the value. Therefore, for **.byte**, **.halfword**, and **.word** pseudo-ops, the linker may report an error such as:

The computed value of a HALFWORD instruction is not within -32,768 ... +32,767 (i.e., 0x8000 ... 0x7FFF).

The linker will also print additional information, including filename, line number, symbol name, and the offending value.

.float

The **.float** pseudo-op is used to allocate 8 bytes and fill it with the IEEE representation of a double-precision (i.e., 64-bit) floating point number. The operand should be a floating point constant. Expressions are not supported.

Examples of floating point constants:

```
.float   123.456
.float   -3.4E-21
.float   +4.5e+21
```

It should be noted that Blitz-64 supports only double-precision floating point arithmetic; single-precision is not supported.

See the above comments regarding alignment. The assembler will not issue a warning when the **.float** instruction occurs on an improperly aligned address.

.string

The **.string** pseudo-op is used to place character data in memory.

Escapes (such as **\n**) can be used. These were described previously.

The string is not null-terminated. If desired, the null character can be included in two ways. For example:

```
str: .string          "Bye\0"
```

is equivalent to:

```
str: .string          "Bye"
     .byte            0
```

The characters are Unicode characters encoded in UTF-8. For example the following are equivalent. Since the UTF-8 encoding of "é" is the two byte sequence 0xC3A9, both will place 5 bytes in memory.

```
.string          "café"
.string          "caf\xc3\xa9"
```

Unicode and UTF-8 are described in a separate document titled "An Overview of Unicode".

.skip

The **.skip** pseudo-op causes the assembler to skip over a number of bytes, without specifying initial values. The operand is an expression which is evaluated at assembly time.

The bytes are guaranteed to be filled with zeros before execution begins.

If a label precedes the **.skip** pseudo-op, then that symbol is associated with the address of the first byte in the block of bytes allocated by the **.skip** pseudo-op.

Typically, a **.skip** instruction is used to define the memory region to be used for a large data structure, such as an array:

```
MyArray:    .skip       1000
```

However a **.skip** instruction can be used for any variable. In KPL, all variables are guaranteed to be initialized to zero values.

```
MyVarWord: .word       0
MyVarWord: .skip       4   ← equivalent
```

If the **.skip** instruction appears in a segment that is marked "zero-fill", then no bytes are actually stored in the object file. (The initializing zeros are generated at runtime when the program is loaded into memory.) Otherwise, the object file will contain N bytes, all filled with zero. Consequently, the programmer should normally place uninitialized variables in a zero-filled segment, particularly if they are large.

This expression used in a **.skip** instruction cannot rely on imported values or values that cannot be determined easily by the assembler.

[By "easily", we mean this. Several synthetic instructions depend on addresses and changes to addresses can, in some cases, change the number of machine instructions required for a synthetic instruction. The **.skip** instructions are evaluated before such synthetic instructions are evaluated. However, in some cases, the values of expressions can depend on addresses, and the translation of synthetic instructions may change addresses. The **.skip** pseudo-op cannot rely on values that cannot be determined early in the assembly. This is only an issue for pathological programs; normally the value for a **.skip** instruction is a simple integer.]

.align

The **.align** pseudo-op is used to insert padding bytes to force the next following thing to be aligned.

In the following example, the string may end on an improperly aligned address; the **.align** pseudo-op will insert as many bytes as necessary to guarantee that the variable "x" is properly aligned.

```
str: .string         "hello"
     .align          8
x:   .doubleword     0x0123456789abcdef
```

The padding bytes inserted by **.align** are guaranteed to be zero-filled.

The operand for **.align** may be 2, 4, 8, 16, or 32. The keyword "page" may be used as the operand, instead of an integer:

```
     .align          page
```

The ".align page" instruction will add padding bytes as necessary to round up to the next page aligned address, i.e., to an address that is a multiple of 16,384 (i.e., a multiple of 16 KiBytes and in which the least significant 14 bits are zeros).

The **.align** statement will insert only as many bytes as necessary. If the address is already aligned, then no bytes will be inserted. The inserted bytes are guaranteed to have value 0x00.

The **.align** statement is not normally preceded by a label. However, if a label is present, it will label the first padding byte. If no padding is inserted, the label will label will be associated with the address of the following byte.

In the following example, the value of "strX" will be 5 greater than the value of "str", regardless of how many bytes are inserted by the **.align**:

```
str: .string         "hello"
strX:
     .align          8
```

The assembler will keep track of all alignment up to word alignment and can fully process the following two types of **.align**. The assembler will effectively transform these into the necessary .skip instructions.

```
.align          2
.align          4
```

The following cannot be fully handled by the assembler and must be passed to the linker.

```
.align          8
.align          16
.align          32
.align          page
```

[When expanding synthetic instructions, the linker may move data and instructions in memory. However, the linker will always insert and move in multiples of 4. Thus, word-alignment will be preserved.]

Instructions must be halfword aligned. At runtime, the Blitz-64 program counter (PC) will always contain an even number by hardwiring the final bit to 0. Thus, alignment is forced and no exception is possible.

Since the assembler keeps track of halfword and word alignment, it can detect any attempt by the programmer to place an instruction at an odd (non-halfword aligned) address, and will issue a warning.

Often, the programmer will place an ".align 2" instruction directly before a code sequence (such as a function) to make certain the instructions in the function are aligned properly, regardless of what preceded the code sequence.

.export

This pseudo-op expects a single symbol as an operand. This symbol must be given a value in this file, either with an **.equ** instruction or used as a label. This symbol with its value will be placed in the object file and made available to other assembler source programs during linking.

For example:

```
                .export   MyFun
                .align    4
    MyFun:      add
                …
                ret

                .export   MyConstant
    MyConstant: .equ      100
```

The **.export** instruction may appear before or after the line that defines the symbol, as the programmer prefers. There must be no label on the same line as this instruction.

If a symbol is not exported, then that symbol may only be used within the assembly source code file in which it is defined. Other files are free to define, export, and import symbols with the same spelling. As long as the symbol is not imported in the current file, these other files will define and use separate, unrelated symbols that are not visible in the current file.

.import

This pseudo-op expects a single symbol as an operand. This symbol must not be given a value in this file; instead it will receive its value from another assembly source file during linking. All uses of this symbol in this file will be replaced by that value by the linker.

For example:

```
                .import   OtherFun
                call      OtherFun
```

The **.import** instruction may appear before or after lines that use the symbol, as the programmer prefers. There must be no label on the same line as this instruction.

A symbol must not be both imported and exported. Every symbol used in a given source code file will either be:

- imported
- exported
- local only

.equ

A symbol may be given a value with an "equate" instruction:

    symbol :   `.equ`   *expression*

The expression may give an absolute value or a relocatable address. For example:

```
Val_123:   .equ   MyConstant+23   ← specifies absolute value 123
MyAddr:    .equ   MyFun+8         ← specifies an address
```

The expression may use symbols that are defined later in the file.

A line containing an **.equ** instruction must begin with a symbol followed by a colon. (In all other situations, the symbol in the label field of an instruction will be given as its value, the address of the instruction. However, in the case of an equate, the symbol is being associated with the result of the expression evaluation.)

```
X:      .word     100      ← X = address of 4 bytes containing 100
Y:      .equ      100      ← Y = 100; no memory or addresses are involved
```

Some expressions may depend on the value of addresses:

```
Z:      .equ      X+4      ← Z = address of the bytes following variable X
```

In some cases, the value cannot be computed by the assembler and the evaluation of such expressions must be deferred to the link stage.

In the following example, the assembler is unable to transform a synthetic jump because the target location is defined in another file. The assembler cannot

determine whether to produce one or two machine instructions, so it leaves that task to the linker. As a result, the assembler is not able to determine the value to be associated with "W". This can cause an error if the symbol is used in a context where the assembler must know the value, such as an instruction which requires an immediate value. Since the assembler must be able to guarantee that the value will fit into the available space (i.e., a 16 bit immediate field), the assembler must be able to determine the value at assembly time. In practice, code sequences like this are unlikely to occur and will not be produced by the KPL compiler.

```
Addr1:     …
           jump    ExtLab        ← Could be one or two instructions
Addr2:     …

W:         .equ    Addr2-Addr1   ← Can't compute until link time

           addi    r1,r2,W       ← Error: must know the value at asm time
```

.begin

The **.begin** pseudo-op tells the assembler when to produce a segment of code and is used to associate several parameters with the segment.

Many programs will contain only a single **.begin** pseudo-op and the programmer will place it at the beginning of the assembly code source file.

Segments are described later, in a separate chapter.

The **.begin** pseudo-op has an operand field that can contain a number of comma-separated parameters.

```
.begin      parameter , parameter , parameter , parameter
```

For example:

```
.begin      startaddr=0x8000a0000,executable,writable
```

The following parameters are indicated by a keyword, which is either present or absent.

```
kernel
executable
writable
zerofilled
```

The programmer may also include a "**startaddr=**" parameter:

```
startaddr=value
```

The programmer may also include a "**gp=**" parameter:

```
gp=value
```

Segments are not given names and there must be no label on the **.begin** instruction. Any label directly preceding a **.begin** pseudo-op will be associated with an address in the previous segment.

As an example to illustrate this, the value of "strEnd" will be an address, namely "strStart+5". The value of "msg" will also be an address, but will very likely be different from "strEnd" since the linker will place the new segment at somewhere different. Perhaps the new segment will be placed directly following the bytes "hello", but this is not guaranteed. In any case, the assembler will treat "strEnd" and "msg" differently because they are in different segments.

```
strLen:    .equ    strEnd-strStart
strStart: .string  "hello"
strEnd:

        .begin
msg:

        .string  "world"
```

Chapter 3: Symbols and Expressions

Quick Summary

- A symbol may be defined in two ways:
    - A label on an instruction defines a new symbol.
    - The **.equ** pseudo-op equates a symbol to the value of an expression.
- The value of a symbol will either be an absolute value or relocatable address .
- The assembler can usually evaluate and determine relative offsets.
- Some expressions using addresses may require finalization by the linker.
- A symbol may also be imported, in which case its value is unknown.
- Use of imported symbols will require finalization by the linker.
    - Errors involving imported symbols may not be detected until link time.
- Expressions may use the usual operators: +, -, <<, >>, &, |, ...
- Operator precedence follows traditional languages (C, Java, ...).
- Expressions are used in instructions that take immediate values.
- Expressions are used in **.byte**, **.halfword**, **.word**, **.doubleword**, and **.skip**.
- All expression evaluation is done using signed, 64 bit integer arithmetic.
- In situations requiring fewer bits, the assembler will detect overflow errors.

Symbols

The assembler builds a symbol table, mapping identifiers to values. Each symbol is given exactly one value. There is no notion of scope or lexical nesting levels, as in high-level languages.

Each symbol is given a value which will be either:

    absolute
    relative
    external

An **absolute value** consists of a 64-bit quantity. A **relative value** consists of a 64-bit (signed) offset relative to either a segment or to an external symbol. An **external symbol** will have its value assigned in some other assembly file and its value will not be available to the code in this file until link time. However, an external symbol may be used in expressions within this file; the actual data will not be filled in until link time.

Symbols may be defined internally or externally. If a symbol is used in this file, but not defined, then it must be "imported" using the **.import** pseudo-op. If a symbol is defined in this file and used in other files, then it must be "exported" using an **.export** pseudo-op. If a symbol is not exported, then its value will not be known to the linker; if a symbol is imported in some files but never exported, then an "undefined symbol" error will be generated at link time.

If a symbol is neither exported nor imported, it will be entirely local to a single .s file. Another file may define another symbol with the same spelling without any confusion; it will be an entirely distinct symbol.

Within a file, a symbol may be defined either...

    as a label
    in an equate

A symbol may be defined by being used as a label, in which case it is given a value which consists of an offset relative to the beginning of whichever segment is current when the label is encountered. This is determined by whether the **.begin** pseudo-op was seen last, before the label was encountered. Each label occurs in a segment and names a location in memory. At link time, the segments are placed in their final positions in memory. Only at link time does the actual address of the location in memory become known. At this time, the label is assigned an absolute value by the linker.

When a symbol is defined using the **.equ** pseudo-op, it is given a value equal to the value of some expression, possibly involving other symbols.

Labels

The label on any instruction will define a new symbol, and the symbol will be given an offset relative to the beginning of the current segment.

Labels defined in the current file may be exported and labels defined in other files may be imported.

A label will name an address in memory, and as such a label cannot be given a final value until link time.

During the assembly of the current file, labels in the file are given offsets relative to the beginning of the segment in which they appear.

Equates

An **.equ** pseudo-op must contain a label and an expression. For example:

```
    MAX:      .equ      1000*8
```

The symbol defined in an equate may be exported.

The expression may involved various operations and other symbols, as in:

```
    SYM_2:  .equ     MAX + 0x18_0000
```

Expression Syntax and Evaluation

Instructions and pseudo-ops may use expressions as operands. Expressions may be occur in:

    .byte
    .halfword
    .word
    .doubleword
    .skip
    .equ
    various Blitz-64 instructions

The syntax of expressions is given by the following context-free grammar.

> *expr*   ::= *expr1* { "**|**" *expr1* }
> *expr1* ::= *expr2* { "**^**" *expr2* }
> *expr2* ::= *expr3* { "**&**" *expr3* }
> *expr3* ::= *expr4* { ("**<<**" | "**>>**" | "**<<<**" | "**>>>**") *expr4* }
> *expr4* ::= *expr5* { ("**+**" | "**-**") *expr5* }
> *expr5* ::= *expr6* { ("*****" | "**/**" | "**%**") *expr6* }
> *expr6* ::= "**+**" *expr6* | "**-**" *expr6* | "**!**" *expr6*
> | **ID** | **INTEGER** | **STRING** | "**(**" *expr* "**)**"

[In this grammar, the following notation is used. The characters enclosed in double quotes are terminals in the grammar. The braces { } are used to mean "zero or more" occurrences. The vertical bar | is used to mean alternation. Parentheses are used for grouping. The start symbol is "expr".]

This syntax results in the following precedences and associativities:

highest:	! unary+ unary-	*(right associative)*
	* / %	*(left associative)*
	+ -	*(left associative)*
	<< >> <<< >>>	*(left associative)*
	&	*(left associative)*
	^	*(left associative)*
lowest:	|	*(left associative)*

For example,

```
a + b * c
```

is equivalent to:

```
a + (b * c)
```

Likewise,

```
a + b >> – ! c & d * e
```

is equivalent to:

```
((a + b) >> (– (! c))) & (d * e)
```

If a string is used in an expression, it must have exactly 8 bytes. The string will be interpreted as a 64 bit integer, based on the ASCII values of the 8 characters, or the UTF-8 encodings for non-ASCII characters. With strings, Big Endian order is used: the first character will determine the most significant byte.

The following operators are recognized in expressions:

unary+	nop
unary-	64-bit signed arithmetic negation
!	64-bit logical negation (NOT)
*	64-bit multiplication
/	64-bit integer division with 64-bit integer result
%	64-bit modulo, with 64-bit result
binary+	64-bit signed addition
binary-	64-bit signed subtraction
<<	left shift logical (i.e., zeros shifted in from right)
>>	right shift logical (i.e., zeros shifted in from left)
<<<	left shift arithmetic (i.e., error if loss of significant bits)
>>>	right shift arithmetic (i.e., sign bit shifted in on left)
&	64-bit logical AND
^	64-bit logical Exclusive-OR
|	64-bit logical OR

With the shift operators (<<, >>, <<<, and >>>) the second operand must evaluate to an integer between 0 and 63. The logical shift operators (<<, >>) will shift in 0 bits. The right shift arithmetic operator (>>>) will shift sign bits in on the left. The left shift arithmetic operator (<<<) will treat the argument as a signed integer and will signal an error if significant bits are shifted out.

With the division operators (/ and %), the first operand must be non-negative and the second operand must be positive. (In the "C" language, the/ and % operators have machine-dependent results with negative operands.)

All operators except addition and subtraction require both operands to evaluate to absolute values, which can be determined by the assembler. All arithmetic is done with signed 64-bit values.

If the next two paragraphs are confusing, just look at the examples.

The addition operator + requires that at least one of the operands evaluates to an absolute value. The other operand may be an address. If one operand is an address, then the result will be relative to that location. Thus, the assembler will be unable to determine the value and the linker (which will place the segments in memory) will be required to determine the exact value.

For the subtraction operator, the first operand may be an absolute value or an address. If the first operand is an absolute value, then the second must also be an absolute value. If the first operand is an address and the second is an absolute value, then the result will be relative to that address. If both operands are addresses, the result will be an absolute value, which represents the difference in bytes between the two addresses.

```
Lab_1:  add     r1,r2,r3        ← Lab_1 is a relocatable address

        …
Lab_2:  add     r1,r2,r3        ← Lab_2 is a relocatable address

        …
max:    .equ    100             ← max is an absolute value

        …
u:      .equ    Lab_1 + 8       ← The value is a relocatable address
v:      .equ    8 + Lab_1       ← The value is a relocatable address
bad_1:  .equ    Lab_1 + Lab_2   ← Error, not allowed
w:      .equ    max + 8         ← The value is an absolute value
x:      .equ    max − 8         ← The value is an absolute value
y:      .equ    Lab_1 − 8       ← The value is a relocatable address
z:      .equ    Lab_2 − Lab_1   ← The value is an absolute value
bad_2:  .equ    8 − Lab_2       ← Error, not allowed
```

An attempt is made to evaluate all expressions at assembly-time. If the expression cannot be evaluated at assembly time, the problem is passed on to the link stage.

The following will prevent an expression from being evaluated at assembly time.

- The expression depends on symbols which are imported.
- The expression depends on the value of an address.

Here are the instructions which might depend on the value of an address:

```
movi
jump
call
bXXX
loadX
storeX
```

In most uses of the above instructions, the assembler will be able to determine the exact offset and produce the final machine code translations. However, in some cases, the assembler will be unable to complete the translation of the instruction and must pass the task on to the linker.

This happens whenever one of the above instructions uses an address and the assembler cannot determine the exact offset between the instruction and the target address. Whenever the assembler is unable to produce the final machine code, the linker will be required to complete the translation.

Pseudo-ops such as **.word** and **.doubleword** may also use expressions which contain values that cannot be determined until link time.

An expression may evaluate to either an absolute 64-bit value, or may evaluate to a relocatable value. A relocatable value is a 64-bit offset relative to some symbolic address. If the expression evaluates to a relocatable value (i.e., an address), its absolute value cannot be determined until link time.

At link time, the absolute locations of the segments will be determined and the absolute values of all symbols will be determined. At link time, the final, absolute values of all expressions will be determined by adding the offsets to the addresses assigned to the relocatable symbols.

The **.skip** pseudo-op requires the expression to evaluate to an absolute value.

In the case of the **.equ** pseudo-op, the expression may evaluate to either a relocatable address or an absolute value. In either case, the equated symbol will be given a relocatable or absolute value (respectively). The actual value may not be determined until at link time. Normally, the symbol would be used in other instructions, and the computed value will be placed in the appropriate bytes in memory at link time.

NOTE: At this time, all instructions except synthetic instruction of format S-1, ..., S-7 require expressions to have an absolute value that can be determined by the assembler before linking. Here are the instructions with formats S-1, ..., S-7:

```
movi
jump
call
bXXX
loadX
storeX
```

All other instructions require values that must be computable by the assembler alone.

NOTE: In the case of a subtraction expression where both operands are addresses, the assembler must be able to determine the relative offset between the two addresses. The computation will not be passed on to the link stage. While the assembler is not required to know the actual addresses involved in the subtraction, it must be able to determine the exact size of everything between the two addresses. This is because the assembler must be able to compute the difference between the addresses. This requires that all of the following conditions hold: (1) Both addresses must be in the same segment. (2) If any synthetic instructions fall between the two addresses, the assembler must be able to fully determine the length of the translations, if not their exact translations. (3) If any **.align** instructions fall between the two addresses, the assembler must be able to fully resolve them. This means that only halfword and word **.align** instructions may be used between the two addresses; larger **.align**s cannot be fully translated by the assembler and must wait for the link stage.

Chapter 4: Segments

Quick Summary

• The linker combines segments to produce an executable file.
• Each assembly source file will contain one or more segments.
• Segments are identified with the **.begin** pseudo-op.
• All the bytes in each segment are contiguous and placed in memory as a unit.
• Segments contain instructions and data bytes.
   • Every instruction and data byte is in exactly one segment.
• A segment may be pinned to a specific location or may be relocatable.
• A segment may be marked as executable or not.
• Each segment is either read-only or read-write.
• The "gp" register is used to make addressing faster.
   • Most accesses to static data can be done in a single gp-relative instruction.

Segments

The linker will place code and data into pages of memory. Each page of virtual address space will be marked either executable or not, and each page will be marked either writable or not. With the Blitz-64 hardware design, any page that is mapped into the virtual address space will be readable, so there is no such status as "present, but not readable".

Each assembly code source file consists of a sequence of "**segments**". Each segment starts with a "**.begin**" pseudo-op and consists of a sequence of instructions. The segments are listed one-after-the-other in the source code file. Thus, every line in the source file will belong to exactly one segment.

An assembly source file will typically contain just a couple of segments, and sometimes only a single segment. For example a given assembly source file might contain two segments: The first segment contains instructions and these bytes will

go into pages marked "executable" but not "writable". The second segment contains data and variables, and these bytes will go into pages marked "writable" but not "executable".

A segment may have any size, although the assembler will round each segment up to a multiple of 8 bytes, by appending padding zeros at the end as necessary. A segment size of zero is possible, but pointless.

[Note: When we referred to the "size of the segment" in the previous paragraph, we meant the size as the assembler determines it and the number of bytes it puts in the .o object file. Later, when the linker is processing a segment, the linker may insert bytes in the course of translating synthetic instructions and processing **.align** pseudo-ops. Thus, the size of the segment may be changed by the linker and may no longer be a multiple of 8 bytes. Although the assembler adds padding bytes to the segment, it would probably have been a better design if the assembler did not add those bytes. Instead, the assembler ought to add "padding bytes" to the .o object file *after* the segment data. These padding bytes would be to ensure that the following fields in the .o file are properly aligned, and would not increase the size of the segment itself.]

When placed in memory, each segment will be placed on a doubleword aligned address. A single page of memory may contain parts of several segments.

The term "segment", as used here, is a purely software concept used only by the assembler and linker; at runtime there is no such thing as a segment. (Other computer systems have used the term "segment" differently, e.g., for regions of memory supported by various hardware features.)

The purpose of the ".**begin**" pseudo-op is delineate segments and to specify some parameters that apply to the segment, like "writable" or "executable".

Below is a small, artificial example, representing a single assembly source code file containing three segments:

```
        .begin          executable
entry:  loadd           r1,myVar
        addi            r1,r1,300
        stored          myVar,r1
        ret

        .begin          writable
myVar:  .doubleword     12345
other:  .doubleword     200

        .begin
str:    .string         "Hello"
        .byte           0
        xor             r1,r2,r3
```

Each segment must start with a **.begin** pseudo-op. A segment runs from a **.begin** pseudo-op until just before the next **.begin** pseudo-op, or until the end-of-file. Every instruction and every other pseudo-op will be located in exactly one segment, based on where it is placed.

There is no requirement that an "executable" segment contains only machine instructions; it may contain data as well. There is no requirement that a "writable" segment contains only data; it may contain machine instructions as well.

In this example, the third segment is marked with neither executable nor writable. It contains a string and an XOR instruction. This segment is not executable and the XOR instruction cannot be executed.

The **.begin** pseudo-op has an operand field that can contain a number of comma-separated parameters.

```
    .begin      parameter , parameter , parameter , parameter
```

For example:

```
    .begin      startaddr=0x8000a0000,executable,writable
```

The following parameters are indicated by a keyword, which is either present or absent.

```
kernel
executable
writable
zerofilled
```

The programmer may also include a "**startaddr=**" parameter:

```
startaddr=integer
```

The programmer may also include a "**gp=**" parameter:

```
gp=integer
```

The "value" associated with a "**startaddr=**" or "**gp=**" parameter must be an integer; expressions are not allowed. Normally, this value is expressed in hex, but decimal is also okay. The following (in which "**undefined**" is a keyword recognized by the assembler) is also allowed:

```
gp=undefined
```

The parameters can be given in any order.

Segments are not given names and a source line containing **.begin** must not contain a label. Any label directly preceding a **.begin** pseudo-op will be associated with an address in the previous segment.

The job of the linker is to determine where in memory to place the segments. More specifically, the input to the linker will be a number of object files, each containing a number of segments. These segments must be placed into memory pages. One constraint is that two segments with different executable/writable attributes may not be placed in the same page. Another constraint is that segments may not overlap. The linker will attempt to pack segments close together in order to reduce the number of pages in the final memory image.

Normally, the linker will be free to choose the location of a segment. However, the programmer may demand that the linker place a segment at a given memory address. This is the purpose of the "**startaddr=**" parameter, which gives the starting address of the segment as an absolute value. This parameter forces the linker to

place a segment at a particular location in memory. The **startaddr=** value must be a doubleword aligned address.

If there is no starting address given for a segment, the linker is free to place the segment where it best fits. By default, the linker will place segments in the virtual address region, which starts at 0x8_0000_0000. The linker will more-or-less place segments one after another, filling up the virtual address space from 0x8_0000_0000 on up, within the previously mentioned constraints.

However, the presence of the "**kernel**" keyword will force the linker to place the segment in the lower, physical region of address space. Segments with this keyword will be placed in low memory, starting with 0x0_0000_0000 and going up.

The "**zerofilled**" keyword is used to indicate that a segment will contain only zeros. Thus, only the following are allowable in a "**zerofilled**" segment:

```
.byte           0
.halfword       0
.word           0
.doubleword     0
.float          0.0
.skip           <any>
.align          <any>
.equ            <any>
.import         <any>
.export         <any>
```

The data in zerofilled segments is not present in the object and executable files, since the pages can be created and initialized at the time the executable file is loaded into memory. Zerofilled segments are useful for large data structures (such as gigantic arrays, spaces for heaps, and so on), since these data structures would waste a large amount of space in the object and executable files.

For example:

```
        .begin      startaddr=0x900000000,writable,zerofilled
MyHeap: .skip       0x100000000      # 4 GiBytes
```

The assembler will round each segment up in size to a multiple of 8 bytes, by adding 1 to 7 bytes of 0x00, as necessary. The linker will place each segment on an aligned 8

byte address. Or, to put it another way, the linker will assign to each segment a doubleword aligned address, where the bytes will be placed when the executable is loaded at runtime.

Note that in Unix/Linux systems, segments are given names such as

```
.text
.data
.rodata
.bss
```

Blitz doesn't do it this way. Unix/Linux confuses segment attributes with the segment names. We see no good reason to name segments in the first place.

The Global Pointer Register, gp

Several of the synthetic instructions include an operand that can be an "address". Examples include:

```
beq         Reg1,Reg2,address
loadb       Reg1,address
storew      address,Reg2
call        address
jump        address
movi        Reg1,address
```

In the course of generating code, the assembler and linker must be able to translate memory addresses into the forms required by the machine instructions. For example, consider this line from an assembly source file:

```
loadb       r1,MyVar
```

Assuming the address of MyVar is within 0 ... 0x0_0000_7fff, then the above instruction can be assembled as:

```
load.b      r1,0x7fff(r0)
```

The virtual address space starts at 0x8_0000_0000. Thus, this optimization only works for programs running in kernel mode, since user programs cannot access data using addresses that are not in the virtual memory region.

However, the global pointer register (gp) is intended to be used for the same purpose, making a range of addresses in the virtual address region particularly quick to access.

For user programs running in a virtual address space, the default assumption is that the global pointer register (gp) will contain the value 0x8_0000_8000 at runtime.

In order for this to work, the gp register will be initialized either by the kernel during thread-creation or within the first couple of instructions at thread-startup, as part of the thread initialization prologue.

User programs typically place their data at the beginning of the virtual address space, i.e., at 0x8_0000_0000. If register gp contains 0x8_0000_8000 — which it normally will — then accessing any data within the first 64 KiBytes can be done with only one instruction.

For kernel code programs, the default assumption is that the global pointer register (gp) will contain the value 0x0_0001_0000. In combination with register r0, this makes accessing data in the first 96 KiBytes of memory especially efficient. (For the first 32 KiBytes, we use a positive offset from r0 and for the following 64 KiBytes we use an offset from gp.)

By assuming the gp register contains one of these known values, the assembler and linker can generate shorter code sequences when translating some synthetic instructions.

The "**gp=**" parameter tells the assembler and linker what value will be in the register gp at runtime.

The keyword "**undefined**" can also be used to override any assumption about the contents of the gp register. In this case, the assembler and linker will not make any assumption about the contents of the gp register for any instructions in that segment. This would be used for code in which the gp register (i.e., register r13) is used for an entirely different purpose.

```
.begin      gp=undefined
```

When the assembler is synthesizing a MOVI instruction and the value to be loaded is within a certain range of values, the assembler may use gp, as shown in the following example.

Consider the following code:

```
        movi            r1,MyVar            # Load address into reg
        …
  # Put data segment in the usual place:
        .begin          startaddr=0x800000000,writable
  Arr:    .skip          0x84D0              # Size = 34,000 bytes
  MyVar:  .doubleword   1234
```

The MOVI instruction is a synthetic instruction which moves the address of a variable into a register. The address of "MyVar" is

```
    0x8,0000,84D0
  = 0x8,0000,0000 + 0x84D0
  = 0x8,0000,8000 + 0x04D0
```

Since the assembler can assume that gp contains 0x8_0000_8000, it can translate the MOVI into a single ADDI instruction, exactly as if the programmer had coded this:

```
        addi            r1,gp,0x04D0
```

Without this assumption about gp, the assembler would be forced to use two instructions, such as:

```
        upper20         r1,0x80000
        addi            r1,r1,0x84D0
```

(This example was simplified. Actually XORI would be used and we failed to account for sign extension properly, but you get the idea.)

More precisely, positive offsets will be used for addresses above 0x8_0000_8000 and negative offsets will be used for addresses below that:

```
  8_0000_0000 … 8_0000_7fff        negative offset 8000 … ffff from gp
  8_0000_8000 … 8_0000_ffff        positive offset 0 … 7fff from gp
```

The assembler/linker can deal with arbitrary addresses, but addresses outside this range might require additional instructions or the use of the temp register "t". Therefore, the programmer is encouraged to place commonly used variables at the bottom of the virtual address space, in the first 64 KiBytes. The typical practice would be to place all static, non-stack data at the bottom of the virtual address space, with the code segments in pages following the data pages.

The above comments about register gp apply not only to MOVI but also to LOAD and STORE instructions. LOAD and STORE are used to access data in static, fixed memory locations. Thus, the gp-relative addressing scheme of Blitz-64 enables the vast majority of accesses to static data variables to be performed with a single instruction.

The BRANCH (Bxx), JUMP, and CALL instructions also use arbitrary addresses as targets. For them, PC-relative addressing is more common. However, the gp-relative addressing mechanism is still present and gp-relative jumps can be generated whenever the target address happens to be in low memory. As a consequence, it might make sense to place jump tables in low-memory, so the code can easily branch to various entries.

Kernel code will not be running in a virtual address space, so things are different. All addresses will be located in the physical memory region.

For kernel code, the gp register is assumed to be initialized to 0x0_0001_0000. This means that any address in the first 6 pages (i.e., the first 96 KiBytes of memory, 0 ... 0x0_0001_7fff) can be accessed with a single instruction:

```
0_0000_0000 ... 0_0000_7fff        offset 0 ... 7fff from r0
0_0000_8000 ... 0_0000_ffff        negative offset 8000 ... ffff from gp
0_0001_0000 ... 0_0001_7fff        positive offset 0 ... 7fff from gp
```

If the "**kernel**" keyword is present in the **.begin** pseudo-op, the default assumption is that register gp will contain the value 0x0_0001_0000. If the "**kernel**" keyword is not present, the assumption is that gp contains 0x8_0000_8000.

If, for some reason, the gp register will have a different value at runtime, the programmer can override the default assumption with the "**gp=**" parameter. If the programmer wants to prevent the assembler from producing code which relies on the value in in gp, then "**gp=undefined**" can be used on the **.begin** pseudo-op.

The MOVI / gp Exception There is one case where the assembler will not use the assumed value in gp: Whenever the destination register of a MOVI instruction is the gp register itself, the assembler will specifically avoid using any assumed value of gp. This exception makes it possible to initialize the gp register.

For example, it's likely that gp will need to be initialized right after any thread begins execution (in the "thread prologue") to contain its expected value of 0x8_0000_8000. To do this, the programmer might consider using the MOVI instruction. Because of this exception, the MOVI is safe to use for this purpose.

NOTE: The "**gp=**" parameter is not required on the **.begin** instruction. If missing, then the assembler will determine whether the "**kernel**" parameter is present. If this is a kernel segment, the assembler will assume the default value of 0x0_0001_0000. If this segment is not marked "**kernel**", then the assembler will make no assumptions, since the segment might go into kernel memory or into user memory. The assembler will defer to the linker, which will choose the correct default value.

    gp = value　　　　　　Programmer gives the value.
    gp = undefined　　　　　The gp register will not be used for synthetic instructions.
    kernel, *<no gp=>*　　　　A value of 0x0_0001_0000 will be assumed.
    <no kernel>, <no gp=>　Assembler assumes nothing.
                           Linker assumes:　**kernel (-k):**　**0x0_0001_0000**
                                             user:　　　　　**0x8_0000_8000**

Chapter 5: Synthetic Instructions

Quick Summary

• The assembler recognizes a set of synthetic instructions.
• Synthetic instructions are not implemented in hardware.
• The assembler translates each synthetic instruction into an equivalent machine instruction.
  • In most cases, the translation is to a single machine instruction.
  • In the other cases, a couple of instructions will be required.
• The technique of synthetic instructions expands the effective instruction set.
  • Hardware is simplified since only machine instructions are executed.
• In most cases, the assembler can perform the translation.
  • In some cases, the assembler will have to pass the task to the linker.
• The algorithm used by the assembler is complex.
  • The sizes of the translations (1, 2, 3, or 4 instructions) affect address values.
  • The values of addresses affect how many instructions are required.
  • Imported symbols introduce uncertainty, further complicating translation.

Introduction

The technique of using synthetic instructions yields a great enlargement of the instruction set while allowing the underlying hardware to remain very simple.

In some sense, no new functionality is added to the Instruction Set Architecture (ISA). But the presence of the synthetic instructions shows how the underlying machine instructions were designed in order to allow easy implementation of common operations.

The programmer need not use synthetic instructions, but they make programming much easier and the programs more readable.

By keeping the hardware design as simple as possible, we achieve the following:

- The processor core requires fewer transistors and wires.
- The circuit real-estate is smaller.
- More cores can be placed on a single die, leading to improved parallelism.
- The circuits are easier to design, debug, and verify.

The synthetic instructions are documented alongside the machine instructions in the document describing the Instruction Set Architecture (ISA). That document contains an entry for each synthetic instruction, specifying what it does and how it is used.

Simple Translations

A number of synthetic instructions are easy to translate. Such cases:

- Always translate to exactly one instruction
- Have no error conditions

Next, we list the easy translations and we will say nothing further about them.

Arithmetic Negation:

| Synthetic: | `neg` | *RegD,Reg1* |
| Translation: | `sub` | *RegD,r0,Reg1* |

Bit Negation (NOT):

| Synthetic: | `bitnot` | *RegD,Reg1* |
| Translation: | `xori` | *RegD,Reg1,−1* |

Logical Negation (0=False; other=True):

| Synthetic: | `lognot` | *RegD,Reg1* |
| Translation: | `testeq` | *RegD,r0,Reg1* |

Move (Register to Register):

Synthetic: `mov`   *RegD,Reg1*
Translation: `ori`   *RegD,Reg1*,0

Nop:

Synthetic: `nop`
Translation: `addi`   `r0,r0,0`

Call (Through Register):

Synthetic: `callr`   *Reg1*
Translation: `jalr`   `lr,0(`*Reg1*`)`

Jump (Through Register):

Synthetic: `jr`   *Reg1*
Translation: `jalr`   `r0,0(`*Reg1*`)`

Return:

Synthetic: `ret`
Translation: `jalr`   `r0,0(lr)`

CSR Write:

Synthetic: `csrwrite` *CSRReg,Reg2*
Translation: `csrswap` `r0,`*CSRReg,Reg2*

Test If Greater Than:

Synthetic: `testgt`   *RegD,Reg1,Reg2*
Translation: `testlt`   *RegD,Reg2,Reg1*

Test If Greater Than Or Equal:

Synthetic: `testge`   *RegD,Reg1,Reg2*
Translation: `testle`   *RegD,Reg2,Reg1*

Test If Greater Than (Floating):

Synthetic:	`fgt`	*RegD*,*Reg1*,*Reg2*
Translation:	`flt`	*RegD*,*Reg2*,*Reg1*

Test If Greater Than Or Equal (Floating):

Synthetic:	`fge`	*RegD*,*Reg1*,*Reg2*
Translation:	`fle`	*RegD*,*Reg2*,*Reg1*

Test If Equal To Zero:

Synthetic:	`testeqz`	*RegD*,*Reg1*
Translation:	`testeq`	*RegD*,*Reg1*,`r0`

Test If Not Equal To Zero:

Synthetic:	`testnez`	*RegD*,*Reg1*
Translation:	`testne`	*RegD*,*Reg1*,`r0`

Test If Less Than Zero:

Synthetic:	`testltz`	*RegD*,*Reg1*
Translation:	`testlt`	*RegD*,*Reg1*,`r0`

Test If Lass Than Or Equal To Zero:

Synthetic:	`testlez`	*RegD*,*Reg1*
Translation:	`testle`	*RegD*,*Reg1*,`r0`

Test If Greater Than Zero:

Synthetic:	`testgtz`	*RegD*,*Reg1*
Translation:	`testlt`	*RegD*,`r0`,*Reg1*

Test If Greater Than Or Equal To Zero:

Synthetic:	`testgez`	*RegD*,*Reg1*
Translation:	`testle`	*RegD*,`r0`,*Reg1*

Absolute Value

The translation of the "abs" instruction (which computes the absolute value of the contents of one register and moves the result into another register) is slightly more complex, since the translation results in three machine instructions.

However, since the translation always results in exactly three instructions and no error conditions can arise, it is fairly straightforward.

Absolute Value:

```
Synthetic:     abs        RegD,Reg1


Translation:   mov        RegD,Reg1
               bgez       Reg1,+8
               neg        RegD,Reg1
```

Note that the translation itself, as expressed above, uses synthetic instructions. When these are translated, we see the actual translation:

```
Translation:   ori        RegD,Reg1,r0
               b.le       r0,Reg1,+8
               sub        RegD,r0,Reg1
```

Branching Instructions

Recall that there are only four machine instructions which do a "test and branch" operation:

```
b.eq       Reg1,Reg2,offset16
b.ne       Reg1,Reg2,offset16
b.lt       Reg1,Reg2,offset16
b.le       Reg1,Reg2,offset16
```

where "*offset16*" is a 16 bit signed number (i.e., -32,768 … +32,767). The offset will be added to the address of the branch instruction (i.e., the current PC) to give the address of the branch target.

Out of these, the following synthetic instructions are constructed:

```
beq       Reg1,Reg2,Address
bne       Reg1,Reg2,Address
blt       Reg1,Reg2,Address
ble       Reg1,Reg2,Address
bgt       Reg1,Reg2,Address
bge       Reg1,Reg2,Address

beqz      Reg1,Address
bnez      Reg1,Address
bltz      Reg1,Address
blez      Reg1,Address
bgtz      Reg1,Address
bgez      Reg1,Address

bfalse    Reg1,Address
btrue     Reg1,Address
```

where "*Address*" is an arbitrary memory location.

In the first stage of the translation, the assembler will translate the above instructions into one of the following four synthetic instructions. The translation of these four instructions will be discussed in subsequent sections.

```
beq       Reg1,Reg2,Address
bne       Reg1,Reg2,Address
blt       Reg1,Reg2,Address
ble       Reg1,Reg2,Address
```

Here are those first-stage translations:

```
Synthetic:      bgt       Reg1,Reg2,Address
Translation:    blt       Reg2,Reg1,Address
```

Synthetic:	`bge`	*Reg1,Reg2,Address*
Translation:	`ble`	*Reg2,Reg1,Address*
Synthetic:	`beqz`	*Reg1,Address*
Translation:	`beq`	*Reg1,*`r0`*,Address*
Synthetic:	`bnez`	*Reg1,Address*
Translation:	`bne`	*Reg1,*`r0`*,Address*
Synthetic:	`bltz`	*Reg1,Address*
Translation:	`blt`	*Reg1,*`r0`*,Address*
Synthetic:	`blez`	*Reg1,Address*
Translation:	`ble`	*Reg1,*`r0`*,Address*
Synthetic:	`bgtz`	*Reg1,Address*
Translation:	`blt`	`r0`*,Reg1,Address*
Synthetic:	`bgez`	*Reg1,Address*
Translation:	`ble`	`r0`*,Reg1,Address*
Synthetic:	`bfalse`	*Reg1,Address*
Translation:	`beq`	*Reg1,*`r0`*,Address*
Synthetic:	`btrue`	*Reg1,Address*
Translation:	`bne`	*Reg1,*`r0`*,Address*

The Complex Translations

The remaining synthetic instructions are listed next. We group them into seven "formats" which we name "Format S-1" through "Format S-7".

In the following, "*Value*" can be any arbitrary 64-bit value, "*Address*" can be any 36-bit address, and "*Offset*" can be any 36-bit offset value.

<u>Format S-1</u>

```
movi      Reg,Value
```

<u>Format S-2</u>

```
beq       Reg1,Reg2,Address
bne       Reg1,Reg2,Address
blt       Reg1,Reg2,Address
ble       Reg1,Reg2,Address
```

<u>Format S-3</u>

```
call      Address
jump      Address
```

<u>Format S-4</u>

```
loadb     RegD,Address
loadh     RegD,Address
loadw     RegD,Address
loadd     RegD,Address
```

<u>Format S-5</u>

```
storeb    Address,Reg2
storeh    Address,Reg2
storew    Address,Reg2
stored    Address,Reg2
```

<u>Format S-6</u>

```
loadb     RegD,Offset(Reg1)
loadh     RegD,Offset(Reg1)
loadw     RegD,Offset(Reg1)
loadd     RegD,Offset(Reg1)
```

<u>Format S-7</u>

```
storeb    Offset(Reg1),Reg2
storeh    Offset(Reg1),Reg2
storew    Offset(Reg1),Reg2
stored    Offset(Reg1),Reg2
```

Addresses are typically specified symbolically. For example:

```
MyLabel:
        …
        jump        MyLabel
        …
        blt         r1,r3,MyLabel
        …
        call        MyLabel
```

In the case of data, addresses are often used like this:

```
        loadd       r1,MyVar
        …
MyVar:  .doubleword 1234
```

Addresses may also be specified as absolute values, as in:

```
        loadd       r1,MyVar
MyVar:  .equ        0x80000000c
```

Although unusual, addresses may also be specified using expressions, such as the following which offsets from relocatable symbolic address:

```
        jump        ExternLabel+8
        …
        .import     ExternLabel
```

Offsets are typically specified with numbers or symbols that are equated to integers:

```
varX:   .equ        12
        …
        loadd       r1,varX(sp)
        stored      16(r4),r2
```

However, the offset can be specified using an expression.

In the case of a "movi" instruction, the **Value** being loaded into the register can be specified in a number of ways:

```
          movi   r1,0x1234              An immediate value
          movi   r1,MyConst             An equated value
          movi   r1,MyVar               An address of data
          movi   r1,MyFun               An address of code
          movi   r1,(MyFun–MyVar)<<8    Complex expression

          …
  MyConst: .equ   0x1234
  MyVar:   .skip  8
  MyFun:   add    …
```

In the following sections, you will see that some of the translations make use of the **temporary register "t"** (i.e., r8). The programmer should be aware that the assembler and linker may produce code which silently modifies "t". Even though "t" does not appear in the assembly source code directly, any of the following instructions may result in a translation that involves "t".

b*XX*

jump

call

storeb

storeh

storew

stored

In the terminology used by compiler-writers, these instructions "kill" register "t". (Note that the translations for **movi** and **load*X*** or the other synthetic instructions will never silently use register "t".)

Format S-1: "movi   *RegD,Value*"

If *Value* is an absolute integer whose value can be determined by the assembler, then the translation selected will depend on the magnitude of the value involved.

If *Value* is -32,768 … +32,767, then the synthetic instruction will be translated as:

Translation: xori   *RegD*,r0,*Value*

Otherwise, if we know the value of gp and *Value* is within -32,768 ... +32,767 of gp:

Translation:	`addi`	*RegD*,`gp`,*offset16*

where *offset16* = gp - *Value*.

Otherwise, if *Value* is representable with a 36 bit number (-34,359,738,368 ... +34,359,738,367):

Translation:	`upper20`	*RegD*,*upper20*
	`xori`	*RegD*,*RegD*,*lower16*

where *upper20* and *lower16* are computed appropriately.

If *Value* is an address, then it is a 36 bit value within 0x0_0000_0000 ... 0xF_FFFF_FFFF. (In decimal, this is 0 ... 68,719,476,735). The linker will do something a little tricky for addresses in the upper half of this range, i.e., any and all addresses in the user address space. The linker will translate the MOVI using only two instructions, but since bit 35 is a 1 for addresses in 0x8_0000_0000 ... 0xF_FFFF_FFFF, the instructions will place a negative number in the registers. That is, the linker will implicitly sign-extend the address from 36 bits to 64 bits, which will make all addresses in the user address space negative. After this sign-extension, the value will lie within 0xFFFF_FFF8_0000_0000 ... 0x0000_0007_FFFF_FFFF (i.e., within -34,359,738,368 ... +34,359,738,367) which fits the requirements of the translation shown above.

Addresses are used in JUMP, CALL, Bxx, LOADx, and STOREx instructions. All of these instructions will ignore the upper bits, so it doesn't matter whether the upper bits are 0s or 1s.

KPL will represent all pointers with signed values. For code running in user space, addresses will always be negative values.

Some care must be taken by the programmer. As long as the programmer keeps pointers in 64 bit variables, the operations of comparison and incrementing will work fine. However, the programmer should remember that the pointers will usually be negative numbers.

One danger arrises when the programmer attempts to specify addresses by constants. For example:

```
var myPtr: ptr to int = …
…
if (myPtr == 0x80001234) …          Wrong; always false
if (myPtr == 0xFFFFFFFF80001234) …  Correct
```

Otherwise, if *Value* is within 52 bits (-2,251,799,813,685,248 … +2,251,799,813,685,247):

Translation:	upper20	*RegD,upper20*
	shift16	*RegD,RegD,shift16*
	xori	*RegD,RegD,lower16*

where *upper-20*, *shift-16*, and *lower-16* are computed appropriately.

Otherwise, it's the case that *Value* requires a full 64 bits:

Translation:	upper16	*RegD,r0,upper16*
	shift16	*RegD,RegD,shift16a*
	shift16	*RegD,RegD,shift16b*
	xori	*RegD,RegD,lower16*

where *upper16*, *shift16a*, *shift16b*, and *lower16* are computed appropriately.

If *Value* is a relocatable address, the assembler will not attempt to determine its value. Since segments are generally relocatable (i.e., not pinned with "startaddr=" in the .begin instruction), it would usually be impossible for the assembler to determine the exact address anyway.

However, it is much more likely that the assembler can determine the relative offset of *Address* from the current PC. If that offset is representable with a 20 bit number (i.e., -524,288 … +524,287), then the synthetic instruction will be translated as:

Translation:	addpc	*RegD,offset20*

In all other cases, the assembler will not produce a translation and will leave the task to the linker.

(If *Value* is an address but the assembler cannot determine its offset from the movi instruction, it must leave the task to the linker. If *Value* is an address and the assembler can determine the offset from the PC, but the offset exceeds 20 bits, the

assembler will leave the task to the linker. The linker will know the exact absolute value and may be able to find a translation of only one instruction. If *Value* involves an imported symbol, then the assembler will be clueless about its value and must defer to the linker.)

Note that any address can be loaded into a register with only two instructions, and many addresses will require only one instruction. In most cases, the assembler will be able to translate the register load with a single instruction. However, the linker will be required to handle the cases that involve two instructions. Also note, that almost all common small-ish constants (i.e., any number within -32,768 … +32,767) can be loaded into a register with only one instruction.

Format S-2: "`bxx` *Reg1,Reg2,Address*"

If a conditional branch is jumping to relatively close target location, then the translation will be a single instruction. Otherwise, two instructions will be used. Three instructions would be needed almost never, but can be used to cover all possible target locations.

[In Blitz-64, the instruction encoding was chosen so that the range for a single branch instruction is quite large (64 GiBytes). Conditional branches (which generally target a location within the same function or method) will almost always be translated with only a single instruction. Nevertheless, aberrant, extremal cases are also accommodated.]

If *Address* is within -32,768 … +32,767 from the instruction (i.e., if a 16-bit offset from the PC can be used):

```
    Translation of beq instruction:
        b.eq    Reg1,Reg2,offset16

    Translation of bne instruction:
        b.ne    Reg1,Reg2,offset16

    Translation of blt instruction:
        b.lt    Reg1,Reg2,offset16

    Translation of ble instruction:
        b.eq    Reg1,Reg2,offset16
```

If a single instruction cannot be used, then the translation takes a different approach. The condition is negated and we jump around one or even two instructions. The one or two instructions will then make the jump unconditionally.

Here's the idea:

> if x < y then goto Target

is equivalent to:

> if y ≤ x then goto L
> goto Target
> L:

Note that when the condition "*x < y*" is negated, we get "*x ≥ y*". There is no machine code to test for greater-than-or-equal. However, if we swap the order of the registers, this test becomes: "*y ≤ x*", and Blitz-64 has a machine instruction for this test.

In the following, note that we refer to the "offset from the PC". The instruction making the jump (JAL and JALR) is the location from which the offset will be calculated.

If *Address* is within -524,288 ... +524,287 from the jump instruction (a 20-bit offset from PC must be used):

Translation of **beq** instruction:
```
    b.ne    Reg2,Reg1,+8    The test is changed & the regs are swapped
    jal     r0,offset20
```

Translation of **bne** instruction:
```
    b.eq    Reg2,Reg1,+8    The test is changed & the regs are swapped
    jal     r0,offset20
```

Translation of **blt** instruction:
```
    b.le    Reg2,Reg1,+8    The test is changed & the regs are swapped
    jal     r0,offset20
```

Translation of **ble** instruction:

```
    b.lt    Reg2,Reg1,+8    The test is changed & the regs are swapped
    jal     r0,offset20
```

Otherwise, a 36 bit offset will be used:

Translation of **beq** instruction:
```
    b.ne    Reg2,Reg1,+12   The test is changed & the regs are swapped
    auipc   t,upper20
    jalr    r0,lower16(t)
```

Translation of **bne** instruction:
```
    b.eq    Reg2,Reg1,+12   The test is changed & the regs are swapped
    auipc   t,upper20
    jalr    r0,lower16(t)
```

Translation of **blt** instruction:
```
    b.le    Reg2,Reg1,+12   The test is changed & the regs are swapped
    auipc   t,upper20
    jalr    r0,lower16(t)
```

Translation of **ble** instruction:
```
    b.lt    Reg2,Reg1,+12   The test is changed & the regs are swapped
    auipc   t,upper20
    jalr    r0,lower16(t)
```

For the synthetic branch instructions (beq, bne, blt, ...), the target *Address* must be a relocatable address. The assembler does not accommodate absolute values. But keep in mind that this sort of branch is extremely rare.

For example, the following would cause an error message:

```
    blt     r3,r5,0xE47004    Absolute targets are not legal
```

In the unusual event that the programmer really needs to do a branch using a target location expressed as an absolute integer, the following code can be used. (Unlike the **bXX** instructions, the **jump** instruction will accept absolute integers for the target address.)

```
    ble        r5,r3,NewLabel        Note change in condition & reg swap
    jump       0xE47004
NewLabel:
```

The assembler will translate this as follows, achieving the desired effect:

```
    b.le       r5,r3,+12
    upper20    t,0x000E4
    jalr       r0,0x7004(t)
```

Format S-3: "`jump/call  Address`"

The two synthetic instructions in Format S-3 are:

```
    jump       Address
    call       Address
```

Recall that register r14 is the "link register" (also named "lr"). When calling a function, the JAL and JALR instructions will save the return address in register lr. A JUMP is identical to a CALL, except that the return address is not retained, and is sent to "r0" instead.

Jumps and calls to an address specified using an **absolute integer target address** (as shown here) are expected to be extremely rare. Nevertheless, the assembler will translate such a jump or call.

```
    jump       0x000e70400
    call       347810
```

If *Address* is an absolute value within 0 … +32,767, the translation will use a positive 16 bit offset from zero (i.e., register r0).

If the *Address* is within 0xF_FFFF_8000 … 0xF_FFFF_FFFF, then the translation will use a negative 16 bit offset from zero. Recall that the hardware always ignores the uppermost 28 bits of any 64 bit number, so we can address the upper bytes of the memory space with negative numbers.

[You can think of memory as "wrapping around" or, equivalently, taking all addresses "mod 0x0000_0010_0000_0000". For example, zero - 0x4000 = FFFF_FFFF_FFFF_C000; with truncation, we have 0xF_FFFF_C000. Note that address wrap-around makes the uppermost region of the virtual address space (above the stack) a reasonable place to put jump tables.]

```
Translation of jump:
    jalr      r0,immed16(r0)
Translation of call:
    jalr      lr,immed16(r0)
```

Otherwise, if *Address* is an absolute value within -32,768 … +32,767 of the value assumed to be in register "gp":

```
Translation of jump:
    jalr      r0,immed16(gp)
Translation of call:
    jalr      lr,immed16(gp)
```

Otherwise, if *Address* is any other absolute value (i.e., a full 36-bit address is required):

```
Translation of jump:
    upper20   t,upper20
    jalr      r0,lower16(t)
Translation of call:
    upper20   t,upper20
    jalr      lr,lower16(t)
```

It is much more likely that the address will be given as a **symbolic value**, as shown next. In many cases, the assembler will be able to determine the relative distance between the current PC (i.e., the address of the jump/call) and the target.

```
    jump      loop_exit
    call      MyFunction
```

(The assembler will never attempt to determine the absolute integer address of a symbolic label, but it can usually determine the relative offset between two locations, as long as both locations are within the same segment.)

If the assembler can determine the relative offset, and if the target *Address* is within -524,288 ... +524,287 from the jump/call instruction, then a 20-bit offset from PC will be used.

This is the common case. Most jumps and calls will be to targets that are specified as symbolic addresses that the assembler can determine are within 512 KiBytes from the location of the jump/call.

```
Translation of jump:
    jal        r0,offset20
Translation of call:
    jal        lr,offset20
```

If the assembler can determine the relative offset, but if the relative offset exceeds this value, then a 36 bit offset relative to the PC will be used:

```
Translation of jump:
    auipc      t,upper20
    jalr       r0,lower16(t)
Translation of call:
    auipc      t,upper20
    jalr       lr,lower16(t)
```

If the assembler cannot determine the target *Address* or cannot compute a relative offset between *Address* and the current PC, the task of translation will be passed on to the linker.

Format S-4: "loadX RegD,Address"

The memory location in the LOAD and STORE instructions can be specified in two ways, as shown in these examples:

```
        loadd     r7,Address          ← Move data from memory to register
        stored    Address,r7          ← Move data from register to memory

        loadd     r7,Offset(r5)       ← Add immed. value to reg to give address
        stored    Offset(r5),r7       ← Add immed. value to reg to give address
```

[Earlier, we said that the opcode exactly and uniquely determines the format of the operands. This isn't quite true. The LOAD and STORE instructions are exceptions to this and they are the only exceptions.]

All synthetic LOAD operations — regardless of whether the operand has the form "*Address*" or "*Offset*(*Reg*)" — are translated using the following machine instructions. (Note that the period "." in the opcode differentiates between synthetic instructions and machine instructions.)

```
load.b    RegD,offset16(Reg1)
load.h    RegD,offset16(Reg1)
load.w    RegD,offset16(Reg1)
load.d    RegD,offset16(Reg1)
```

In this section we'll use the notation **load***X* where *X* stands for **b**, **h**, **w**, or **d**. Likewise, we'll use the notation **load.***X* as shorthand for **load.b**, **load.h**, **load.w**, or **load.d**.

Format S-4 includes the four synthetic LOAD instructions that have an address as an operand.

```
loadb    r7,Address        ← Fetch a byte
loadh    r7,Address        ← Fetch a halfword (16 bits)
loadw    r7,Address        ← Fetch a word (32 bits)
loadd    r7,Address        ← Fetch a doubleword (64 bits)
```

The LOAD instructions in which the memory address has the form "Offset(Reg)" are discussed later, under Format S-6.

The *Address* can be given in several ways:

```
loadb    r7,MyVar          ← Symbolic, relocatable location
loadb    r7,MyVar+100      ← Symbolic plus/minus integer
loadb    r7,0x00E70040     ← Absolute Address
```

If the assembler can compute difference between the LOAD and the target location, then it will produce code using PC-relative addressing. If a symbolic label is used but the assembler is unable to determine the relative offset, then the task will be passed on to the linker. (This happens whenever the symbol is externally defined, or

whenever the source and target are in different segments, or whenever there is something of unknown size between the source and target locations.)

Otherwise, (i.e., if *Address* is specified as an absolute integer value known to the assembler), the translation will use the translations shown next.

If *Address* is an absolute value within 0 ... +32,767, the translation will use a positive 16 bit offset from zero (i.e., register r0).

If the *Address* is within 0xF_FFFF_8000 ... 0xF_FFFF_FFFF, then the translation will use a negative 16 bit offset from register r0. [Memory "wraparound" and the use of negative offsets was discussed in the section "Format S-3: jump/call".]

> Translation: `load.X`   *RegD*,*immed16*(`r0`)

If *Address* is an absolute number within -32,768 ... +32,767 of the value assumed to be in register "gp":

> Translation: `load.X`   *RegD*,*immed16*(`gp`)

If *Address* is an absolute number of any other value:

> Translation: `upper20`   *RegD*,*upper20*
>   `load.X`   *RegD*,*lower16*(*RegD*)

If *Address* has a PC-relative value that the assembler can determine:

> Translation: `auipc`   *RegD*,*upper20*
>   `load.X`   *RegD*,*lower16*(*RegD*)

Note: In some cases, the assembler will be able to determine a relative offset from the PC but unable to determine the absolute location. In such cases, the assembler will produce the two instruction sequence just shown. If it happens that the target location is also within ±32 KiBytes of zero (r0) or register gp, then a single instruction would have sufficed, although two instructions were generated.

This situation is rare and not considered likely to occur in practice because the typical programming practice is to put variables and data in one segment (marked "writable") and code in another segment (marked "executable"). The data segment

---

will typically be placed where it can be conveniently accessed with the default value assumed to be in register gp (0x8_0000_8000):

```
        .begin          writable,startaddr=0x800000000
Var_1:  .doubleword     1234
Var_2:  .byte           0x34
```

User code cannot access low memory so positive offsets from register r0 are only usable by the kernel. Kernel code will place its data in a segment located at the beginning of memory:

```
        .begin          kernel,writable,startaddr=0x0
Var_1:  .doubleword     1234
Var_2:  .byte           0x34
```

Since the source LOAD instruction and the target data address are in different segments, the assembler will be unable to generate a PC-relative address. The assembler will be forced to pass the task off to the linker. The linker will determine exact addresses and will generate a one-instruction sequence whenever possible.

This comment also applies to the STORE instructions.

Format S-5: "`loadX` *RegD*`,`*Offset*`(`*Reg1*`)`"

A second form of the LOAD and STORE instructions allows the address to be computed by adding a fixed constant value to the contents of a register. This form is particularly useful for accessing variables stored in a stack frame, which is commonly done for variables local to a function or method.

```
loadd       RegD,local_x(sp)
stored      8(sp),RegD
```

Another important use of the "*Offset(Reg)*" addressing form is to access the fields in an object. Each field (i.e., "data member") is located at a known offset within the object and the object itself is pointed to by a register.

Next, we describe the translation of:

```
loadb       RegD,Offset(Reg1)
loadh       RegD,Offset(Reg1)
loadw       RegD,Offset(Reg1)
loadd       RegD,Offset(Reg1)
```

If *Offset* is an absolute integer value within -32,768 … +32,767, i.e., if it can be represented with a signed 16 bit immediate value:

Translation: `load.X`   *RegD,immed16(Reg1)*

If *Offset* is an absolute integer value within -2,147,483,648 … +-2,147,483,647, i.e., if it can be represented with a signed 32 bit value:

Translation: `upper16`   *RegD,Reg1,upper16*
                `load.X`   *RegD,lower16(RegD)*

Otherwise, *Offset* requires a full 36 bits:

Translation: `upper20`   *RegD,upper20*
                `add`   *RegD,RegD,Reg1*
                `load.X`   *RegD,lower16(RegD)*

The assembler is unable to handle that case where *Offset* is given by a relocatable label, as shown below. Such cases will be passed off to the linker.

```
MyArr:   .skip    100000
    …
         loadd    r7,MyArr(r3)
```

It is often the case where the programmer has an address in a register and wishes to use that address directly, without any offset. To do this, the programmer can code it as illustrated by the following example.

```
         loadd    r7,0(r3)
```

which will be assembled identically to the following machine instruction:

```
         load.d  r7,0(r3)
```

We considered adding additional synthetic forms to accommodate shorthand such as shown in the following examples. But we decided against it because it violates the fundamental Blitz-64 design goal of avoiding complexity.

```
loadd  r7,(r3)        ← Syntax error
stored (r3),r7        ← Syntax error
```

Additional Detail We have called the expression "*Offset*" and implicitly assumed that the register contains a "base" address. The effective address will be "base+offset". This is typical of addressing fields in an object, where the register contains a pointer to the object and the literal, immediate value is the offset of some field in that object.

However, the literal, immediate expression might supply the base address and the register might contain an offset. This is common for accessing arrays that are located at statically determine addresses. The address of the array is coded directly into the instruction.

In this comment, we discuss the range of legal values for the literal, immediate value "*Expression*".

In general, the *Expression* may be any address (i.e., any value within 0x … 0xF_FFFF_FFFF) in which case the value to be used will be adjusted to a signed 36-bit value (i.e., within 0x8_0000_0000 … 0x7_FFFF_FFFF). This is equivalent; the lower-order 36 bits are identical, and there are no more bits than that in the address calculations performed in hardware.

The *Expression* may also be an offset, in which case it is reasonable to allow a negative value down to -0xF_FFFF_FFFF (i.e., 0xFFFF_FFF0_0000_0001). For example, consider the case where the programmer has placed a very high address (such as 0xF_FFFF_1234) in register r1.

An offset of -0xF_FFFF_1231 can be used to address location 0x0_0000_0003. Working through this example, -0xF_FFFF_1231 is a negative number and is represented as 0xFFFF_FFF0_0000_EDCF. This offset will be truncated to 36 bits and sign-extended, giving 0x0_0000_EDCF. This is the value that will go into the machine instructions. At runtime, adding 0x0_0000_EDCF to 0xF_FFFF_1234 gives 0x10_0000_0003, which will get truncated by the hardware to 0x0_0000_0003, exactly the address that is desired.

Thus, the linker will accept any value for *Expression* within -0xF_FFFF_FFFF .. 0xF_FFFF_FFFF (i.e., -68,719,476,735 ... + 68,719,476,735) without complaint or warning. Any value for *Expression* outside this range will result in an error message.

This scheme allows location 0 to be reached from the highest address (0xF_FFFF_FFFF) and it allows the highest address (0xF_FFFF_FFFF) to be reached from address 0. Since LOAD and STORE instructions are designed for memory access, any *Offset* value beyond 36 bits must be in error.

For example, consider reaching address 0 from address 0xF_FFFF_FFFF. This requires an offset of -0xF_FFFF_FFFF. Expresses as 36 bits, this value is 0xFFFF_FFF0_0000_0001 = 0x0_0000_0001. Adding, we get 0x0_0000_0000, as desired.

However, note that the assembler will only accept values for *Expression* within a more limited range of 0xFFFF_FFF8_0000_0000 ... 0x0000_0007_FFFF_FFFF (i.e., -34,359,738,368 .. +34,359,738,367 which is -0x8_0000_0000 ... +0x7_FFFF_FFFF). If the assembler can determine the value and this value is outside this range, the assembler will generate an error and fail. In the next paragraph, we explain why this should never be a problem.

If the programmer uses a memory address for *Expression*, the assembler will always defer instruction synthesis to the linker. So, in the only case where the assembler might potentially generate an error, we can assume that the "base" address must be placed in the register, and *Expression* is an "offset". In other words, the given *Expression* must be an offset from an address, not an address itself. For user mode programs we can assume the address in the register must be an address in the user address space. The range limit for the offset expression will still allow any address in user space to be reached from whatever address was in the register. Likewise, for kernel mode programs, we assume that any address calculation will be from an address in the kernel space to another address in the kernel space. Thus, the range limit imposed by the assembler should never be a problem, regardless of what address will be in the register and what offset was supplied. However, in the event that an *Expression* outside the assembler's limit is desired and the assembler is balking, there is a simple work-around. The large value can be placed in a separate .s file, assembled independently, and exported to the source file needing it. Since the assembler will not have access to the value of the *Expression*, it defer synthesis to the linker, which accommodates the full range of offsets.

Of course, any value beyond the linker's range make no sense. The value will be added to the contents of the register and will be used as an address for a LOAD or STORE instruction. Since the hardware addition is limited to 36 bits, the upper bits are pointless.

Format S-6: "storeX   Address,Reg2"

The memory location in the STORE instructions can be specified in two ways, as shown by these examples:

```
stored     Address,r7        ← Move data from register to memory
stored     Offset(r5),r7     ← Add immed. value to reg to give address
```

This section discusses instructions using the first form.

All synthetic STORE operations — regardless of whether the operand has the form "*Address*" or "*Offset*(*Reg*)" — are translated using the following machine instructions:

```
store.b    offset16(Reg1),Reg2
store.h    offset16(Reg1),Reg2
store.w    offset16(Reg1),Reg2
store.d    offset16(Reg1),Reg2
```

In this section we use the notation **storeX** where *X* stands for **b**, **h**, **w**, or **d**. Likewise, we'll use the notation **store.X** as shorthand for **store.b**, **store.h**, **store.w**, or **store.d**.

Format S-6 includes the four synthetic STORE instructions that have an address as an operand.

```
storeb     Address,Reg       ← Store a byte
storeh     Address,Reg       ← Store a halfword (16 bits)
storew     Address,Reg       ← Store a word (32 bits)
stored     Address,Reg       ← Store a doubleword (64 bits)
```

The STORE instructions in which the memory address has the form "Offset(Reg)" are discussed later, under Format S-7.

The *Address* can be given as an absolute integer value or as a relocatable symbol.

If *Address* is an absolute value within 0 … +32,767, the translation will use a positive 16 bit offset from zero (i.e., register r0). If the *Address* is within 0xF_FFFF_8000 … 0xF_FFFF_FFFF, then the translation will use a negative 16 bit offset from register r0.

    Translation: store.X   *immed16(r0),Reg2*

If *Address* is an absolute number within -32,768 … +32,767 of the value assumed to be in register "gp":

    Translation: store.X   *immed16(gp),Reg2*

If *Address* is an absolute number of any other value:

    Translation: upper20   t,*upper20*
                    store.X   *lower16(t),Reg2*

If *Address* has a PC-relative value that the assembler can determine:

    Translation: auipc   t,*upper20*
                    store.X   *lower16(t),Reg2*

Note: The temporary register "t" is used in some of the translations for STORE, although "t" is never used for LOAD instructions.

When translating LOAD, the assembler can use the target register for any address calculation, since it will obviously be available for use as a work register directly before the load.X instruction. However, there is no such free register for use in the translation of STORE instructions. Instead, register "t" will be used.

The programmer should not forget that a synthetic STORE instruction may result in code that overwrites the register "t".

Format S-7: "`storeX Offset(Reg1),Reg2`"

Next, we describe the translation of instructions in which the target memory address is computed by adding a fixed constant value to the contents of a register.

```
storeb       Offset(Reg1),Reg2
storeh       Offset(Reg1),Reg2
storew       Offset(Reg1),Reg2
stored       Offset(Reg1),Reg2
```

If *Offset* is an absolute integer value within -32,768 … +32,767, i.e., if it can be represented with a signed 16 bit immediate value:

Translation:	`store.X` *immed16*`(`*Reg1*`),`*Reg2*

If *Offset* is an absolute integer value within -2,147,483,648 … +-2,147,483,647, i.e., if it can be represented with a signed 32 bit value:

Translation:	`upper16` `t,`*Reg1*`,`*upper16*
	`store.X` *lower16*`(t),`*Reg2*

Otherwise, *Offset* requires a full 36 bits:

Translation:	`upper20` `t,`*upper20*
	`add` `t,t,`*Reg1*
	`store.X` `lower16(t),`*Reg2*

The assembler is unable to handle that case where *Offset* is given by a relocatable label, as shown below. Such cases will be passed off to the linker.

```
MyArr:   .skip    100000
    …
            stored  MyArr(r3),r7
```

It is often the case where the programmer has an address in a register and wishes to use that address directly, without any offset. To do this, the programmer can code it as illustrated by the following example.

```
            stored  0(r3),r7
```

Chapter 6: The Linker

Quick Summary

- The linker tool is called "**link**" and is run after the assembler.
- The linker takes a "**.o**" object file as input and produces an executable file.
  - — The linker can combine several object files into one executable.
- The linker can also take library files as input.
  - — The linker will pull out any object module that is referenced.
- The linker determines memory locations for each segment.
- The linker matches all imported and exported symbols.
  - — An error is reported if an imported symbol is not exported exactly once.
  - — This is the "undefined symbol" error.
- The linker determines the exact values for all symbols.
- The linker translates all remaining synthetic instructions into machine code.
- The linker inserts bytes as necessary for **.align** pseudo-ops.
- The linker algorithm is complex and is described in an appendix.

Using the Linker

The Blitz-64 linker tool is named "**link**". In the simplest use, it converts a single object file into an executable file:

```
link  hello.o  –o hello
```

The linker can combine several object files into a single executable:

```
link  file1.o  file2.o  file3.o  –o myPgm
```

The executable filename must always be given. It doesn't default to "a.out", but you can always say:

```
link  file1.o  file2.o  file3.o  –o a.out
```

The linker can also be suppled with library files as input. There can be zero or more library files as input:

```
link hello.o MyLib1.lib MyLib2.lib MyLib3.lib –o hello
```

Typically, the object files have a filename extension of ".o" and the library files have an extension of ".lib", however this is not enforced by the linker. The linker ignores the extension and determines whether the input file is an object file or a library by looking at the contents of the file. Object files and library files begin with "magic numbers" and these are used to determine what type of file is actually present.

Concerning the names of library files, the filename is given directly, just as for other command lines. (In Unix/Linux, something like "-lm" can indicates a file with name "libm.a" and/or "libm.so". Furthermore, this can result in a search of the directory hierarchy. This complex behavior is absent in Blitz-64.)

Error Messages

The linker will sometimes print **errors** and/or **warnings**.

In all cases, an error will cause the EXIT_FAILURE code to be returned from the linker command to the shell that invoked it. No executable file will be produced. If only warnings are generated, an executable file will be produced.

The error and warning messages are printed on **stderr**. For some messages, additional information will be printed on **stdout**.

Here are the most important **error messages**, all of which arise from programming mistakes:

```
***** LINK ERROR: Undefined Symbol: xxx was imported on line xxx in
   module "xxx" (file "xxx"/"xxx"). No matching export can be found.
   *****
```

***** LINK ERROR: Symbol "xxx" is equated to "xxx" which is imported.
   However, no matching export can be found. (line xxx from file
   "xxx"/"xxx") *****

***** LINK ERROR: The symbol "xxx" was used on line xxx of module
   "xxx" (file "xxx"/"xxx"). This symbol was imported but no matching
   export was found. *****

***** LINK ERROR: This program contains no bytes. *****

***** LINK ERROR: Every program must have an exported symbol "_entry"
   *****

***** LINK ERROR: Symbol "_entry" is not a valid address within this
   program *****

***** LINK ERROR: The EQU symbols xxx (from module "xxx") and xxx
   (from module "xxx") are cyclicly defined *****

***** LINK ERROR: When synthesizing this LOADx instruction, an offset
   value that was not in -0xF,FFFF,FFFF ... +0xF,FFFF,FFFF was
   encountered. *****

***** LINK ERROR: When synthesizing this STOREx instruction, an
   offset value that was not in -0xF,FFFF,FFFF ... +0xF,FFFF,FFFF was
   encountered. ****

***** LINK ERROR: Symbol xxx is exported multiple times (from module
   "xxx" in library "xxx" and module "xxx" in library "xxx") *****

***** LINK ERROR: Symbol xxx is exported multiple times (from module
   "xxx" from file "xxx" and module "xxx" from file "xxx") *****

***** LINK ERROR: These segments have different (executable,
   writable) attributes but try to occupy the same page. *****
   (Segments are also printed)

***** LINK ERROR: In computing the value of the EQU symbol
   "xxx" (from module "xxx"), overflow has occurred *****

Additional Errors

There are a large number of additional error conditions which are less common. Each of these conditions will cause an immediate termination of the linker after printing the error message.

These errors fall into these classes:

- **Invalid Command Line** Note that the command line option **-h** is always valid and will do nothing but produce some help info about what command line options are expected.

- **Problems with the Format of an Input File** The linker performs a number of tests and checks on the format of object and library files. If something seems wrong with an input file, the linker will terminate immediately. Such a message is likely to be the result of a bug in the assembler or createlib tools.

- **Memory Allocation Failure** There is not enough memory for the linker to allocate its internal data structures.

- **I/O Error** A problem was reported by the host OS during a system call to read input files or write output files.

- **Failure of the Linker to Find a Placement for the Segments** The algorithm used by the linker is reasonably clever but may, for some extreme cases, fail to find a solution. That is, when placing the segments in memory, the linker was unable to find legal locations for all the segments. Since each program has a 0x8_0000_0000 byte (i.e., 32 GiByte) address space, any program causing such a failure would have to be extraordinarily large.

- **Program Logic Error** The linker performs a large number of internal consistency checks. If any test fails, the linker will print a message and halt.

Warning Messages

The following messages are not fatal but probably indicate programmer errors:

```
***** LINK WARNING: When synthesizing this Bxx instruction, an
    illegal target address was encountered. (Use -w1 to suppress this
    warning.) *****

***** LINK WARNING: When synthesizing this JUMP/CALL instruction, an
    illegal target address was encountered. (Use -w1 to suppress this
    warning.) *****

***** LINK WARNING: When synthesizing this LOADx instruction, a
    target address that was not in 0x0 ... 0xF_FFFF_FFFF was
    encountered. (Use -w1 to suppress this warning.) *****

***** LINK WARNING: When synthesizing this STOREx instruction, a
    target address that was not in 0x0 ... 0xF_FFFF_FFFF was
    encountered. (Use -w1 to suppress this warning.) *****

***** LINK WARNING: For reasons that are too complicated to explain,
    a NOP was inserted following the translation for this synthetic
    instruction. This shouldn't hurt anything, but in the interest of
    full disclosure, it may slightly degrade performance. (Command
    option -w2 will suppress this warning.) *****
```

Chapter 7: Support for Runtime Debugging

Quick Summary

- There are a number of assembler **pseudo-op instructions** for debugging support.
- The debugging pseudo-ops allow the compiler to provide information to the debugger.
- The compiler will add debugging pseudo-ops to the **.s** file.
  - — Human assembly programmers will not typically use these pseudo-ops.
- The debugging pseudo-ops are not necessary for execution and only play a role when the debugger is activated.
- The debugging pseudo-ops direct the assembler to add debugging information to the **.o** file.
- The linker will process the debugging information and add it to the executable file.
- The debugging information will be placed at the end of the executable file.
- The debugging information will be ignored when the program is loaded for execution.
- If a debugging tool is used, it will read the debugging info from the executable file.
- The debugging information describes:
  - — Function and method names
  - — Local variable names, types, and locations
  - — Global variable names, types, and locations
  - — Source level statement types and locations
- The debugger will use it to display information in a way that is more human-readable.
- The debugging information includes source file name and line numbers which can be displayed to assist the programmer during debugging.

Debugging Pseudo-ops

The following pseudo-op are used to convey debugging information to the debugger.

```
.sourcefile
.function
.endfunction
.regparm
.local
.global
.stmt
.comment
```

These pseudo-ops are normally produced by the compiler and inserted into the **.s** assembly code file it produces.

These pseudo-ops will not in any way influence how the program executes and which error and exception conditions can occur.

Normally, human assembly language programmers will not bother to use any debugging pseudo-ops. Presumably, an assembly programmer thinks more in terms of labels and and machine instructions, so these pseudo-ops are not always meaningful for programs.

However, nothing prevents the human assembly language programmer from using the debugging pseudo-ops. The assembler and linker tools will perform error checking designed to catch egregious errors that might cause problems with the assembler, linker, and debugger tools. However, nothing prevents the human programmer from making minor mistakes that cause the debugger general confusion and to print out gibberish in its attempt to display debugging information in human-friendly terms.

A label is not allowed on any of the debugging pseudo-ops. Most of them require additional operands, which will be discussed later.

The .sourcefile Pseudo-op

The **.sourcefile** pseudo-op is used to associate a source file name with all the code in the **.s** file. It requires two operands, both of which must be strings. The strings will be passed on to the debugger and will be associated with all other debugging information in the file.

```
.sourcefile     "Filename","OtherInfo"
```

Here is an example usage.

```
.sourcefile     "MyPackage.c","KPL v1.0; Compiled 25-12-2019 19:30"
```

The **.sourcefile** pseudo-op must be placed near the top of the **.s** file, before any other debugging pseudo-ops. If the file contains any debugging pseudo-ops, then it must contain a **.sourcefile** pseudo-op. The **.s** file must not contain multiple occurrences of this pseudo-op.

The **Filename** string is passed through to the debugger, but is not otherwise examined by the assembler or linker. This string is required but may be empty. The debugger will display the **Filename** to the programmer, since a line number alone is insufficiently meaningful. The **Filename** should be the file within which the line numbers have meaning.

The **OtherInfo** string is intended to contain any additional documentation information, such as the nature of the tool that produced the **.s** file and perhaps the date and time at which the file was created. This information is passed through to the debugger, but is not otherwise examined by the assembler or linker. This string is required but may be empty.

The .function Pseudo-op

The **.function** pseudo-op is used to associate a source name with a function or method. (For code bodies, the debugging information makes no distinction between functions and methods.)

```
.function       "SourceName", line=NNN, framesize=NNN
.endfunction
```

Here is an example usage:

```
P_Foo_34:
     .function        "foo", line=57, framesize=32
     store.d          -8(sp),lr
     addi             sp,sp,-32
     …
     addi             sp,sp,32
     load.d           lr,-8(sp)
     ret
     .endfunction
```

The **.function** and **.endfunction** instructions act as pair to indicate which instructions originated from a single source code function or method. The **.function** should be placed directly before the first instruction of the entry prologue and the **.endfunction** should be placed directly after the last instruction of the function.

A function may contain several RETURN statements and the compiler may elect to include several copies of the exit epilogue in the code. Regardless of how many the compiler includes, there must be exactly one **.endfunction** and it must be placed after the last instruction that belongs to the function.

The **.function** pseudo-op requires a *SourceName* string, which is the name of the function or method, as it appeared in the original source file. Due to name mangling, the label in the assembly file may not match the original name chosen by the human.

The **.function** pseudo-op requires the number of the line number on which this function was defined. A zero value is legal and indicates missing information.

The **.function** pseudo-op requires the size of the stack frame (i.e., the activation record) and this is given in bytes. Since frames are always a multiple of 8 bytes in size, this number must be, too. It may be zero, but may not be negative. A leaf frame will always have a frame size of zero; a non-leaf frame will have a frame size of at least 8.

For leaf functions, the debugger will assume that the return value of the current function is in register **lr**. For non-leaf functions, the debugger will assume that the return value of the current function is at offset **-8(fp)**. Here, we use **fp** (frame pointer) to mean the address of the caller's frame. The debugger will compute **fp** as **sp - framesize**.

In the example above, note that the frame size in **.function** (i.e., 32) is the same number used in the entry prologue and exit epilogue. This should always be true, or else the debugger may become confused when looking at the stack.

The **.function** and **.endfunction** instructions form a bracket. Any **.stmt**, **.comment**, **.local** or **.regparm** that occurs between them will be associated with that function. Every **.stmt**, **.comment**, **.local**, and **.regparm** must occur between a **.function** and an **.endfunction** pseudo-op.

The byte range given by the **.function** and **.endfunction** instructions are all associated with that function.

During debugging, if execution is halted, the debugger can look at the current value of the PC to determine whether execution was halted within a known function.

There is no requirement that *SourceNames* for functions be unique; due to renaming in different packages, the same name may be used for different things. An empty string is legal and indicates missing information.

The .global Pseudo-op

Consider a KPL variable definition that occurs outside any function or method code. Thus, the variable is a "global" variable:

```
var myVar: int = 123
```

The purpose of the **.global** pseudo-op is to associate debugging information with the memory locations that will store this variable's runtime value.

The general form is:

```
.global        "SourceName",line=LineNumber,type="TypeCode"
```

For example:

```
P_MyPack_MyVar_19:
    .global        "myVar",line=24,type="I"
    .doubleword    123
```

The **.global** should be placed immediately before the variable as shown above, so as to associate the *SourceName* with the correct memory address.

As before, the *SourceName*, and the *LineNumber* associate attributes with a memory location. There is no requirement that *SourceNames* for global variables be unique; due to renaming in different packages, the same name may be used for different things.

The **TypeCode** is a string which gives the debugger information about the KPL type of the variable. From this, the debugger will determine how many bytes the variable occupies as well as how best to display the variable's value.

The following type codes are used:

> *Easy Types:*
> | | | |
> |---|---|---|
> | I | **int** | 64-bit signed integer |
> | W | **word** | 32 bit signed integer |
> | H | **halfword** | 16 bit signed integer |
> | C | **byte** (C = Char) | 8 bit signed integer or ASCII char |
> | L | **bool** (L = Logical) | TRUE / FALSE, 8 bits |
> | D | **double** | 64 bit double-precision floating point |
> | S | **String** | Ptr to array of bytes |

> *Hard Types:*
> | | | |
> |---|---|---|
> | P | **ptr** | Pointer to anything, 64 bits |
> | A | **array** | |
> | O | **object** | |
> | R | **struct** (R = Record) | Size and types of fields is unspecified |
> | U | **union** | Size and types of fields is unspecified |

A **String** is a pointer to an array of bytes. These are commonly used in KPL to to represent UTF-8 encoded Unicode strings. Strings are common enough to warrant their own type code. A String object can be printed by the debugger, although the debugger should make no assumptions about whether the bytes are UTF-8 codes.

Each **object** carries a dispatch pointer at runtime and this pointer points to a jump table which also contains a pointer to a Class Descriptor. The debugger may be able to extract some information from these data structures in the target program's address space so that it can print out some info about the object.

For **pointers**, **arrays**, **structs**, and **unions**, the debugger is provided with no further information. Thus, it can't display the value of such variables, other than as a sequence of bytes.

Possible Extensions That Were Considered The one-letter type system described above is obviously limited and could be extended, as described here.

For all "easy" types, the type code string will consist of a single character.

For "hard" types, the idea is that the initial character may be followed by additional characters. That is, we will allow the type code string to contain additional characters beyond the first character.

These additional characters encode additional type information for some types. For example, the type

> ptr to XXX

can be encoded with the string

> `"P`*X*`"`

where X is the type code string for type XXX. For example:

type	**encoding**
ptr to int	`"PI"`
ptr to ptr to word	`"PPW"`

This also works for arrays. For example:

type	**encoding**
array of int	`"AI"`
ptr to array of ptr to array of bool	`"PAPAL"`

If the additional characters are present, the debugger can use them for a more human-readable display of values. In the additional characters are missing, the debugger will be less adept at printing values for these types.

At this time, there is no proposal for additional characters following these codes.

O   **object**
R   **struct** (R = Record)
U   **union**

Even the extension proposed is not able to fully accommodate the KPL type system. Consider this KPL code:

type MyType = ptr to MyType
var x: MyType

The type code for **x** would be "PPPPP...". While this example is contrived, it shows the existence of a deeper problem.

We could propose an encoding that addresses the problem of an object/struct/union that contains a pointer to same type, but it will be complicated.

However, a complex type system is just not needed and will violate our fundamental goal of keeping Blitz simple.

The .local and .regparm Pseudo-ops

Consider a KPL variable that is local to some function or method code. This could be a parameter or a local variable:

```
function foo (myParm1: bool, myParm2: MyClass)
  var myLocal: int = 123
  …
endFunction
```

The purpose of the **.regparm** pseudo-op is to associate debugging information with a register that will be used to pass a parameter.

The purpose of the **.local** pseudo-op is to associate debugging information with stack locations that will store the values or parameters and local variables.

The general forms are:

```
.regparm       RegNum,"SourceName",line=LineNumber,type="TypeCode"
.local         Offset,  "SourceName",line=LineNumber,type="TypeCode"
```

For example:

```
P_Foo_34:
      .function        "foo", line=57, framesize=32
      .regparm         1, "myParm1", line=57, type="L"
      .local           32, "myParm2", line=57, type="O"
      .local           8, "myLocal", line=58, type="I"
      store.d          -8(sp),lr
      addi             sp,sp,-32
```

Typically, the **.regparm** and **.local** instructions will be placed immediately after the **.function** as shown above.

The **RegNum** is a number (1 .. 7) which tells which register the parameter is passed in: **r1** ... **r7**.

The **Offset** tells where in the stack a parameter or local can be found. Note that the offsets are relative to the stack top pointer after the function or method prologue. In a leaf function, the **sp** register will not be changed, so this doesn't make any difference. However, for non-leaf functions it matters. In this example, the frame size is 32 bytes and the function prologue adjusts register **sp** by this amount.

Parameter **myParm2** is located at the very top of the stack at the time of the CALL instruction, so it has offset 0 upon entry. However, after the prologue, the offset of **myParm2** is +32.

The **SourceName**, **LineNumber**, and **TypeCode** work as described earlier.

The **.local** pseudo-op may also be included by the compiler for any temporary variables the compiler creates which have no human-created **SourceName**. A empty **SourceName** is legal but is discouraged. A simple name of "_temp" is acceptable, but names like "_temp_23" are better.

There is no requirement that **SourceNames** be unique, even within a single function.

WARNING: The compiler is free to move globals, locals, and parameters into registers. The compiler will take great effort to keep them in registers as much as possible.

As such, the values stored in memory will often be out of date and _memory may contain obsolete values_.

The programmer should never forget this when using the debugger.

At the entry to a function or method, the parameters will always be where they are expected to be. Thus, the **.local** and **.regparm** info will be correct and an attempt to see their values at the beginning of a function or method will display their correct values. However, once the function or method gets underway, the current values may be placed in registers in ways that are likely to confuse a human. Since the debugger doesn't know about how the compiler has choosen to use the registers, the debugger may printout incorrect or out-of-date values.

Likewise, the compiler may keep a global variable in a register, instead of writing it out to memory immediately. [In the case of "shared" variables, the compiler is forced to write out the values as soon as possible whenever they change and to read from memory whenever the value is needed.]

But for most global variables, the compiler will defer writing the values to memory and may use register copies to avoid memory reads.

The result is that, by examining variables with the debugger, it is easily possible that the programmer will see obsolete values. The natural response is to ask why the variable is incorrect and to focus mental effort debugging something which is, in fact, not an error at all.

Even when the value in memory and the value in a register happen to be the same, the compiler may make all accesses and updates to the register copy, not to memory. Thus, if the programmer uses the debugger to make a change to a variable's value as stored in memory (where the debugger thinks it is), the actual code may ignore this value and continue to use the value cached in a register.

So again, the programmer should be very aware that examining or reading a variable's value (other than a "shared" global or a parameter at the very beginning of a function or method entry) is fraught with danger.

The .stmt Pseudo-op

Consider a KPL assignment:

```
x = y + 123
```

The purpose of the **.stmt** pseudo-op is to associate debugging information with the range of instructions that implements a single source level statement.

The general form is:

```
.stmt          StatementType, line=LineNumber
```

There are a number of **Statement Types**.

Each **.stmt** pseudo-op must be placed directly before the sequence of instructions to which it applies. The range of instructions continues until the next **.stmt** or **.endfunction** pseudo-op. The **.stmt** pseudo-op may only occur between a **.function** and an **.endfunction** pseudo-op.

```
.function      …
…
.stmt          assign,line=63
loadd          r1,16(sp)
addi           r1,r1,123
stored         32(sp),r1

.stmt          if,line=64
…
.endfunction
```

Here is the list of statement types.

```
 0  <.comment>      COMMENT
 1  assign          ASSIGNMENT statement
 2  if              IF statement
 3  then            THEN statement
 4  else            ELSE statement
 5  call            FUNCTION CALL
 6  send            SEND statement
 7  while_expr      WHILE LOOP (expr evaluation)
 8  while_body      WHILE LOOP (body statements)
 9  do_body         DO UNTIL (body statements)
10  do_expr         DO UNTIL (expr evaluation)
11  break           BREAK statement
12  continue        CONTINUE statement
13  return          RETURN statement
14  for_init        FOR statement (before initialization)
15  for_body        FOR (body statements)
16  for_incr        FOR (before increment)
17  for_expr        FOR (before test)
18  switch          SWITCH statement
19  case            CASE
20  default         DEFAULT
21  try             TRY statement
22  throw           THROW statement
23  catch           CATCH clause
24  free            FREE statement
25  debug           DEBUG statement
26  init_arr        INITIALIZE ARRAY statement
27  init_obj        INITIALIZE OBJECT statement
28  set_arr_sz      SET ARRAY SIZE statement
```

The .comment Pseudo-op

The **.comment** pseudo-op is used to associate an arbitrary comment string with a memory address. Here is the general form:

```
.comment            "CommentString"
```

Here is an example usage:

```
.comment            "Reg 4 contains X"
```

This string is associated with the memory location. When that memory location is examined using the debugger, the debugger may display that information.

The **.comment** pseudo-op is designed to help break apart complex statements. The **.comment** instruction can be inserted by the compiler to explain what it is doing or to document something not covered by the **.stmt** pseudo-ops. A prime example would be to document a function or method invocation within a larger expression.

Example Consider this KPL source code, which contains a function call (**foo**) and a message send (**bar**) within an expression.

```
…
i = foo (i) + x.bar (k)
if (a >= b)
…
```

The compiler might produce the following assembly code. We assume the compiler is smart enough to insert a **.stmt** pseudo-op before the code for each statement and a **.comment** before each function or method invocation. (I've highlighted in bold the debugging pseudo-ops inserted by the compiler. I've also added comments to explain what the code is doing, although it is unlikely the compiler will provide such useful comments.)

```
        …
        .stmt          "AS",line=87
        loadd          r1,16(sp)              # argument i
        .comment       "call foo"
        call           P_Foo_34               # perform call
        stored         32(sp),r1              # save in temp
        loadd          r1,40(sp)              # receiver x
        loadd          r2,48(sp)              # argument k
        .comment       "send message bar"
        loadd          t,0(r1)                # perform send
        jalr           lr,88(t)               # .
        loadd          r2,32(sp)              # retrieve temp
        add            r1,r1,r2               # perform addition
        stored         16(sp),r1              # save in i
        .stmt          "IF",line=88
        blt            r3,r4,_Label_97        # if (a >= b) ...
        …
```

Next, assume the program is executed and an error has occurred at runtime. Assume the debugger tool is activated and the programmer wishes to use the "disassemble" command to display the contents of memory.

Here is how the debugger might display memory contents. Using the debugging information, the debugger is able to display the debugging information (highlighted in bold). This additional information makes a straight memory dump comprehensible.

```
 ...
ASSIGNMENT (line 87)
 00000AB00: 1E0010F1  load.d    r1,16(sp)  # offset = 0x10
call foo
 00000AB04: 190013CE  call      P_Foo_34   # PC + 0x13C
 00000AB08: 220021F0  store.d   32(sp),r1  # offset = 0x20
 00000AB0C: 1E0028F1  load.d    r1,40(sp)  # offset = 0x28
 00000AB10: 1E0030F2  load.d    r2,48(sp)  # offset = 0x30
send message bar
 00000AB14: 1E000018  load.d    t,0(r1)    # offset = 0x0
 00000AB18: 1A00588E  jalr      lr,88(t)   # offset = 0x58
 00000AB1C: 1E0020F2  load.d    r2,32(sp)  # offset = 0x20
 00000AB20: 00010211  add       r1,r1,r2
 00000AB24: 220011F0  store.d   16(sp),r1  # offset = 0x10
IF (line 88)
 00000AB28: 12001434  b.lt      r3,r4,0x14 # if (r3<r4) goto _Label_97
 ...
```

Perhaps the compiler is clever and is able to generate a more descriptive string for the **.comment**. For example, the compiler might insert something like:

```
.comment     "call foo (i)"
.comment     "send message x.bar(k)"
```

Of course the more numerous the **.comment**s are and the more descriptive the strings are, the more space will be consumed in the object and executable files to contain this debugging information. Therefore, the compiler may elect to insert minimal debugging information. [Note that the assembler identifies identical strings and will represent each string only once. So if the same string is used repeatedly in many **.comment** pseudo-ops, no additional space will be required for subsequent uses of the same string.]

Chapter 8: Assembler Programming Conventions

Quick Summary

- Function calling conventions are described.
- Support for debugging is discussed.
- Representation for objects and classes is described.
- Method dispatching is described.
- Examples are given showing how code can be compiled into assembly.
    — Some common compilation patterns are given.
    — The fit of the Blitz-64 instruction set to the KPL language is discussed.

Function Calling Conventions

Whenever some code contains a "**call statement**" to invoke a function, we refer to that code as the "caller" or "calling code". The function being invoked is referred to as the "called" function or the "callee".

The caller and called functions are often compiled separately and the compiler has no knowledge of one function when compiling the other. Therefore, a set of "**function calling conventions**" is adopted and used for all functions. Assuming the code generated for the caller and for the called functions both respect these conventions, the function invocation and return will work properly.

The compiler will follow these conventions, but assembly language programmers are free to do anything they want. For hand-coded assembly functions that call compiler-generated functions, or for hand-coded assembly functions that are meant to be called by compiler-generated code, it is mandatory that the calling conventions are followed. For large assembly programs, the programmers would be well-advised

to follow the standard calling conventions. (Actually, nobody writes large assembly language programs any more, so this is a moot point.)

For convenience, we repeat the register usage conventions:

	Alternate Name	Function
r0		Zero
r1		Argument 1 / Return Value
r2		Argument 2
r3		Argument 3
r4		Argument 4
r5		Argument 5
r6		Argument 6
r7		Argument 7
r8	**t**	Temp register, used by assembler/linker
r9	**s0**	Work reg (caller-saved)
r10	**s1**	Work reg (caller-saved)
r11	**s2**	Work reg (caller-saved)
r12	**tp**	Thread data pointer
r13	**gp**	Global data pointer
r14	**lr**	Link register
r15	**sp**	Stack pointer

Consider a function named "foo"; we use the CALL and RET instructions to invoke the function. For example:

Source file of caller:

```
        …
        call    foo
        .import foo
        …
```

Source file of the called function:

```
    foo:
        .export foo
        … Code for foo …
        ret
```

If the caller and the called code are in the same source file, then we dispense with the **.import** pseudo-op instruction.

Next, we give the basic **function calling register conventions** including the **rules for passing arguments**.

Let's define an argument to be "**small**" if its size is 8 bytes or smaller. This means all arguments with a basic type—i.e., **int**, **word**, **halfword**, **byte**, **bool**, **double**, and pointer—are small. Some objects, structs, and unions may also be small. Every object requires at least an 8 byte header (the dispatch table pointer) which means the object would have no fields in order to be "small", but that might happen. Arrays are never "small" since they have a header of 8 bytes plus at least 1 element.

- The first 7 "small" arguments will be passed in registers **r1**, ... **r7**.

- A small argument that is passed in a register will be sign-extended whenever it is of less than 64 bits. For example, an argument of type "byte" will occupy the entire register.

- If there are fewer than 7 small arguments, they will be passed in registers **r1** ... **r**_N_. For example, if arguments 1, 2, 5, and 9 are the only small arguments, they will be passed in registers **r1**, **r2**, **r3**, and **r4**. The remaining registers (**r5**, **r6**, and **r7**) will contain garbage, by which we mean they contain the remnants of previous computations by the caller.

- All remaining arguments are passed from caller to callee by being placed in memory on the runtime stack, as described later.

- If there is a return value and it is "small", it will be returned in register **r1**. If there is a return value but it is not small, it will be returned on the runtime stack.

- Upon return, registers **r2** ... **r7** will contain garbage. Register **r1** will also contain garbage, unless the function returns a small value, in which case **r1** is used to return that value.

- Register **r8** (i.e., register **t**) is the "**temporary work register**". Upon invocation it will contain garbage and the callee can make no assumptions about its value. The register may be used by the callee, as needed. Upon return, the register is garbage and the caller can make no assumptions about its value.

- Registers **r9**, **r10**, **r11** (i.e., **s0**, **s1**, **s2**) are known as the "**work registers**". Upon entry, they will contain garbage and the callee is free to use them as needed.

- Registers **r1** though **r11** (i.e., **r1**…**r7**, **t**, **s0**…**s2**) are said to be "**caller-saved**". The caller must not assume their values will be preserved across the call. If they contain important information to the caller, then that function is responsible for saving their contents before the call and restoring them after the called function returns. Thus, the callee is free to use these registers without saving their contents first.

- Register **r12** (i.e., **tp**) is the "**thread pointer**". Register **tp** is typically fixed and unchanging throughout the execution of a program. It is used to point to a region of memory that is specific to an individual thread. In this way, a function can determine in which thread it is executing and can access any per-thread data. This register is said to "callee saved" in the sense that it must not be modified by the callee. If, for some strange reason, the callee changes its value, it must first save and then restore that value before returning.

- Register **r13** (i.e., **gp**) is the "**global pointer**". This register typically contains a fixed value which is used to making accessing static data (i.e., global variables) easier. This register typically remains unchanged throughout the entire program execution. This register is said to be "callee saved" in the sense that it must not be modified by the callee. If, for some strange reason, the callee changes its value, it must first save and then restore that value before returning.

- Register **r14** (i.e., **lr**) is the "**link register**" and is used directly by the CALL and RET instructions. This register is loaded with the return address by the CALL instruction so, upon invocation of a function, this register contains the return address. The RET instruction depends on this register containing that return address. If the called function intends to call other functions, it must first save the contents of register **lr** and then restore **lr** before executing its own RET instruction.

- Register **r15** (i.e., **sp**) is the "**stack pointer**" register. It is callee-saved and must not be modified. More precisely, anything pushed onto the runtime stack must be popped before return, so there must be no net change to this register.

By "the register will contain garbage", we mean that it will contain some undetermined, unspecified value. Upon invocation, the caller may have left the results of some previous computation in the register. However, the caller will no longer need that value, so the callee need not save that value and is free to use the

register. Upon return, the caller must assume that "garbage" registers contain undefined values. The caller cannot assume that these registers contain whatever the caller put in them before the function invocation.

When arguments are passed in registers, the register will contain these values:

Arg Type	Register Contains...
int	64 bit signed integer
pointer	36 bit address; the upper 28 bits are undefined
double	64 bit floating point value
word	32 bit signed integer; upper 32 bits will be sign extension
halfword	16 bit signed integer; upper 48 bits will be sign extension
byte	8 bit signed integer; upper 56 bits will be sign extension
bool	64 bits (0=FALSE, 1=TRUE)
object	The object, which must be exactly 64 bits in size
struct/union	The struct/union, which must be ≤ 64 bits in size

When a value smaller than 64 bits is passed in a register, the value will be sign-extended.

Whenever the processor uses the contents of a register as an address, the upper 28 bits are ignored. We never care about the upper 28 bits of an address. Generally, the upper bits of a pointer are zeros.

For other values, it is critical that the upper bits are sign-extended. Consider how a **byte** value of -1 might be passed in a register:

```
0xFFFF_FFFF_FFFF_FFFF     Correct
0x7FFF_FFFF_FFFF_FFFF     Incorrect
```

Imagine the code in the called function wishes to add +1 to the value. If the register contains the sign-extended value, then it works correctly, yielding 0. If the register contains the incorrect value, an Arithmetic Exception is erroneously generated.

The following is the meaning of the 8 bits stored in a boolean variable are:

    0 = FALSE
    anything else = TRUE

Typically, the value 1 is used for TRUE. (The compiler always makes the comparison against 0, and never against 1. However, when comparing two **bool** values, the

compiler is allowed to use a single EQ test. This guarantees a correct result as long as TRUE is always represented with 1 and other non-zero values are avoided.)

Commentary Concerning the design choices for register calling conventions, there are several questions:

- How many registers shall be devoted to argument passing?
- Shall some registers be declared to be "callee-saved" and how many?
- Shall some registers be "caller-saved work registers" and how many?

We decided to devote a lot of registers to argument passing.

Note that almost all functions have 7 or fewer arguments. Passing arguments in registers is very important for efficiency and 7 covers almost all cases. Also note that any register set aside for arguments that is not needed for that purpose, automatically becomes a "work register" for the callee function. So in many cases, 7 registers will suffice for all arguments and a few work registers. Notice that the argument numbers 1, 2, 3, ... coincide with the register numbers **r1**, **r2**, **r3**, ...

The thinking here is that every argument has to be "marshaled" (i.e., the argument expression must be evaluated and the result placed somewhere where the callee can find it). In order to perform this marshaling, each argument must at least be moved into a register in the caller's code. Moving the argument to memory is an additional step which may or may not be necessary. The Blitz-64 strategy is to try to avoid these STORE instructions.

The caller can't know which arguments are best kept in memory; only the callee can. So the idea is to delay saving the arguments to memory. This allows the callee to save whichever arguments to memory it chooses. Leaving all arguments in registers gives the maximal freedom to the callee to determine which arguments to keep in registers and which to move into memory.

For functions with fewer than 7 arguments, there will naturally be left-over registers which can be used as "work" registers by the callee. For functions with 7 or more arguments, all registers will be in use upon function entry. Presumably some arguments will be needed immediately, but the caller cannot know which. If the callee needs additional work registers beyond those otherwise available, it will be required "spill" some registers to memory. But only the callee can choose the best registers to spill. With up to 7 arguments in registers, we are effectively giving the decision making to the callee, where it can be made more effectively.

The **t** register is a very local temporary work register, frequently used in synthetic instructions, so its use is fixed.

The registers **tp**, **gp**, **lr**, and **sp** have dedicated uses.

This leaves 3 registers: **s0**, **s1**, and **s2**.

Initially, we defined **s0**, **s1**, and **s2** to be callee-saved, but reversed this decision and made them caller-saved.

Either choice has pitfalls: In one case, the caller must save them every time a function call is made, even though many callees may ignore them, which is a waste. In the other case, the callee must save them if they will be needed and restore them, even if they don't contain any valid caller data; again a waste.

It probably makes sense to have a few callee-saved registers. The compiler can look at each function "f" and make decisions about which registers to use. If "f" contains many functions calls, it makes sense to keep data in callee-saved registers, with the hope that the callees will be able to avoid using these registers. If there are few function calls in "f", then it makes better sense to keep data in caller-saved registers, since this allows "f" to avoiding saving the registers, with the hope that "f"s caller does not use the register.

Note that if a function "f" is small-ish, then it often won't need extra registers. Furthermore, if "f" is small, it is also more likely to be in-lined, in which case the issue is moot. On the other hand, if "f" is large-ish, then it is likely "f" will need the extra registers. And since "f" is large, it is likely its execution will require a lot of time. So it makes sense for "f"s caller to save the registers, if necessary, relieving "f" of the need to spill registers to memory.

We chose to make **s0**, **s1**, and **s2** caller-saved but not contain arguments because there are few functions requiring more than 7 arguments. In those few cases where there are, we still need a couple of work registers available for computation, or else we'll have to immediately spill the arguments to memory, which defeats putting them in registers in the first place.

This is all pretty sketchy reasoning and this may be an open research question deserving serious experimentation. Perhaps Blitz-64 can be used to try variations of the calling conventions, to try to locate the optimum balance between caller-saved and callee-saved registers. It is unclear how much performance potential awaits discovery.

The Runtime Stack

A runtime stack is maintained and the **sp** register points to the "top" of this stack.

The stack grows downward, from high memory addresses towards location 0.

The "top" of the stack is thus "below" the items deeper in the stack, in terms of memory addresses.

The **sp** register is decreased in value for a "push" operation and increased in value for a "pop" operation.

The **sp** register points to the first byte of the item at the top of the stack. The remaining bytes of the top item can be accessed with positive offsets from register **sp**. Items below the stack top (that is, deeper in the stack) are also accessible with positive offsets.

When referring to stacks, we use the words "top", "above" and "upper" to mean those items which are closest to the stack top. Since the runtime stack grows downward, these terms can be confusing since those items actually have "smaller" addresses and are located "lower" in memory. [This can be confusing: When item *x* is said to be "above" item *y*, it can mean item *x* is closer to the stack top and thus has a *smaller* address, or it can mean that item *x* has a *larger* address and is thus farther from the stack top. The best approach is to be careful to say "larger or smaller *addresses*", or "closer to the stack top" and "deeper in the stack".]

The **sp** register will always be an even multiple of 8. In other words, whenever an item is pushed onto the stack, that item will be rounded up in size to an integral number of doublewords.

Upon entry to a function, the stack top register **sp** will always point to the top item in the stack, or more precisely, to the first byte of the top item. After returning from the function, there will be no net changed to the stack. In particular, the **sp** register will be unchanged at the time of the RET instruction. Furthermore, there will be no changes to items already in the stack (with a couple of exceptions discussed later). In other words, the bytes with addresses equal and greater than **sp** will be unchanged by the invocation and return of a function.

However, there is no such guarantee about bytes above the top of the stack, i.e., the bytes with addresses lower than the value in **sp**. The called function is free to push items onto the stack (thereby overwriting whatever was in those bytes), as long as every item pushed is also popped before return.

While we said that register **sp** points to the first byte of the item at the top of the stack upon function *entry* and function *exit*, there is no constraint that bytes with addresses below **sp** cannot be used *during* the function.

A "**leaf function**" is a function that does not invoke any other functions. Many functions are not leaf functions because they may call other functions. In other words, a leaf function does not contain any CALL instructions, and a function that contains CALL instructions is not a leaf function.

Since a leaf function will not call any other functions, it will not need to use register **lr**. Thus, the leaf function can leave its own return address in **lr**. There is no need for the leaf function to save its return address. On the other hand, a non-leaf function must save its own return address before calling other functions. A non-leaf function must save the value of **lr** and must restore it before returning. Thus, a non-leaf function will require more instructions on entry and on return than a leaf function.

Functions are free to make use of memory locations above the stack top (i.e., at addresses that are less than register **sp**). This is important for leaf functions.

Since a leaf function will not be calling other functions, it does not need to worry about another function pushing data onto the stack. Therefore, the leaf function is free to use memory "above" the top of the stack (i.e., at memory addresses less than the **sp** register) to store its temporary and local variables.

A leaf function does not need to decrement **sp** upon function entry or increment **sp** upon function exit. It can simply use negative offsets from sp for the storage of its data. This saves two additional instructions upon the entry and exit of the leaf function.

However, it is important to note that the OS kernel can not rely on the **sp** register to delimit the runtime stack. The OS kernel may not make the assumption that only bytes with addresses greater than or equal to the **sp** register contain valid data. Because leaf functions are using bytes "above" the stack top, this assumption is incorrect.

Obviously, the OS kernel or any additional interrupting code cannot push information onto the stack using the **sp** register and expect a return to the interrupted code to be possible. Since the interrupted code could have been a leaf function, such an interrupting process that uses bytes beyond the stack top may possibly alter or overwrite bytes that were in use by the interrupted code.

Argument Locations and the Parameter Block

As mentioned previously, the first 7 arguments of basic types are passed in registers and all remaining arguments are passed in memory. Next, we describe this in detail.

The remaining argument values are passed on the runtime stack and will be at the top of the stack upon function entry. The following diagram shows the stack and **sp** register upon function entry, just before the first instruction is executed.
The caller will allocate space for all arguments in the **parameter block**. The called function will rely on the space being allocated exactly as described here.
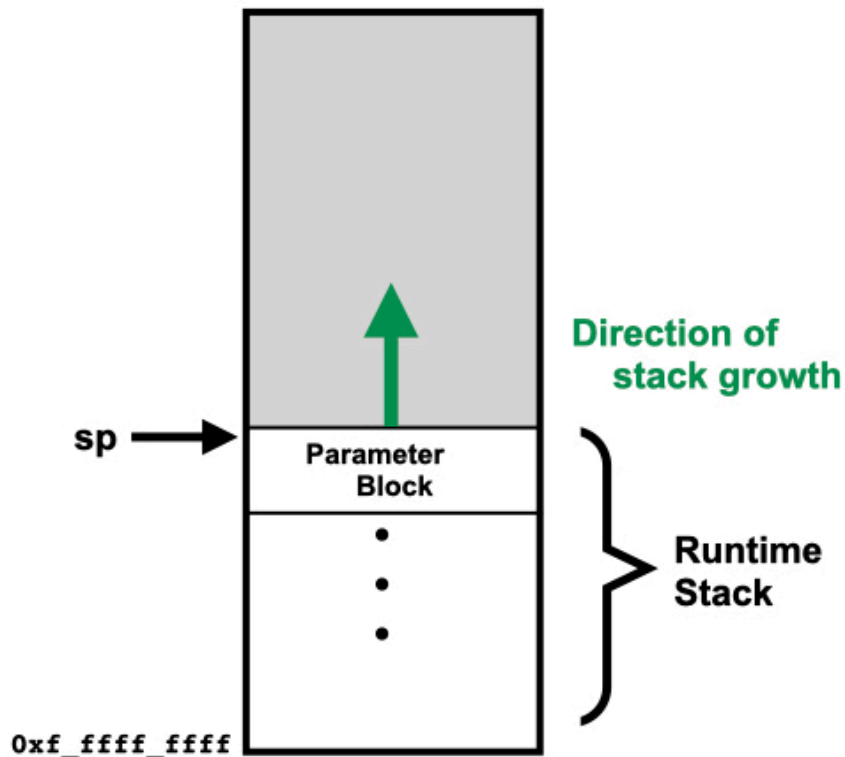
The parameter block will include space for both arguments that are passed in registers and for arguments that must be passed on the stack. For arguments that are not passed in registers, the values will be placed in the parameter block by the caller.

For arguments that are passed in registers, space will also be allocated in the parameter block. The space will be present, but will contain no useful data. The called function is free to use that space as a place to store the argument values if it wishes.

If there is a returned value of 8 bytes or smaller, it will be returned in register **r1**. If larger than 8 bytes, the function will place it at offset 0 in the parameter block.[2] In any case, the caller must assume that all argument values stored in the parameter block before the call are lost / overwritten / trashed by the called function.

(Note that allocating extra uninitialized bytes in the parameter block has a zero performance cost. The caller is not a leaf function, so it must allocate a stack frame regardless. Adding several bytes to the size of the stack frame only changes the value by which sp must be decremented when the stack frame is created, and incremented when the function returns. Since the bytes are uninitialized, no additional instructions are required.)

The parameter block will occur at the top of the stack and will contain space for each argument. The arguments will be placed in the order in which they appear in the source code. Padding bytes will be inserted, as required to meet the alignment requirements for each argument.

To illustrate, here is a function prototype:

```
function foo (
    i1, i2:    int,
    p3, p4:    ptr to …,
    c5:        MyClass,
    b6, b7:    bool,
    a8:        MyArray,
    i9, i10:   int,
    c11:       MyClass,
    h12:       halfword,
    w13,w14:   word,
    h15:       halfword,
    w16:       word,
    d17:       double,
    b18:       bool,
    b19, b20:  byte,
    h21:       halfword )
```

The layout of the parameter block is shown next.[3]

---

[2] In the event that the returned value is larger than all argument values combined, the size of the parameter block will be increased as necessary to accommodate the returned value.

[3] We assume that objects of **MyClass** are 16 bytes in size and arrays of type **MyArray** require 80 bytes.

The first 7 arguments that are 8 bytes or shorter will be transmitted in registers, as shown. All other arguments will be placed by the caller on the stack. Upon entry to the called function, the arguments will be found at the indicated offsets from the stack top, **sp**.

Offset		**Size**	
0	r1	8	i1: int
8	r2	8	i2: int
16	r3	8	p3: ptr to …
24	r4	8	p4: ptr to …
32		16	c5: MyClass
48	r5	1	b6: bool
49	r6	1	b7: bool
50		6	...*padding*...
56		80	a8: MyArray
136	r7	8	i9: int
144		8	i10: int
152		16	c11: MyClass
168		2	h12: halfword
170		2	...*padding*...
172		4	w13: word
176		4	w14: word
180		2	h15: halfword
182		2	...*padding*...
184		4	w16: word
188		4	...*padding*...
192		8	d17: double
200		1	b18: bool
201		1	b19: byte
202		1	b20: byte
203		1	...*padding*...
204		2	h21: halfword
206		2	...*padding*...

The total size of this parameter block is 208 bytes; the parameter block will always be a multiple of 8 bytes in size.

The called function will probably not be a leaf function, so it will itself need its own stack frame. Upon entry, the called function will begin by pushing a new frame by decrementing **sp** by some amount. This will, of course, alter the offsets it must use to access the parameter block.

For example, if function foo needs a stack frame of (say) 3000 bytes, then it will subtract 3000 from **sp** within its entry prologue. Then, in order to access an

argument such as "w16" at offset 184 in the parameter block, the called function will need to use offset 3184 from **sp**.

Debugging Support

Bugs occur and programs must be debugged. A program called a "**debugger**" is used to assist the programmer in finding bugs.

In Blitz, the debugger will be invoked immediately as a result of an error occurring. At the moment the debugger becomes active, the program is frozen. Its virtual memory is still intact, along with other state information such as the values of the registers.

In this section, we will discuss how the code generated by the compiler interacts with the debugger.

The debugger is itself a program, separate from the program being debugged. There are several possible organizations:

(1) The debugger will is integrated with the target program and inhabits the same virtual address space as the program being debugged.

(2) The debugger is integrated within the kernel and is a part of the kernel.

(3) The debugger is a separate user-level process which makes use of special features of the kernel to access the target program's memory.

(4) The code is being emulated and the debugger is part of the emulator.

As of this writing, the last option is fully implemented and is used to debug programs written in KPL and assembly language.

In KPL, all errors result in "throwing" an error. The program itself may catch the error, in which case the program may take appropriate actions. But if not caught, the default action is to invoke debugging.

The first task of the debugger is to determine where execution was when the error occurred. For many types of error, there will be additional information about the

error. For example, if an array index is out of range, we want to capture and make the (incorrect) index value available.

In Blitz, errors are detected in either of two ways. First, some types of error will cause a runtime exception. Second, the compiler will insert code that will explicitly test for other types of errors.

In the first case, errors caught by runtime exceptions are checked by the hardware and involve no overhead, since there are no additional instructions. As part of the exception processing, registers (including the **PC**) will be saved and an error handling function will be invoked.

In the second case, errors caught with explicit tests will cause a CALL to be made to error handling code. (Typically, the code generated by the compiler will test for an error condition and will branch around a CALL instruction.) The CALL instruction will be executed only if the error happens and, as normal for any CALL, the return address will be saved.

Regardless of how the error handler was invoked, the value of the **PC** register at the time of the error will be captured and used to locate where in the code the error arose. Also, any other pertinent information (such as an invalid array index) will be captured and saved by the error handler function.

Unfortunately, the value of **PC** is a memory address, i.e., a binary number not likely to be meaningful to a human. To help the human, this address must be translated into meaningful information, such as a line number within some source code file.

Also, the programmer may wish to examine the contents of variables and parameters. These will be stored at various offsets from the stack top. To assist the programmer, the debugger will need to know which function was executing and what offsets were used for various parameters and local variables.

In other words, the debugger will need some information about the program being debugged. Some debugging information is specified with pseudo-ops such as **.function**, **.local**, and **.stmt**.

But where is this information to be stored?

Information about the program (which will be used by the debugger) is stored in two places:

- Within the executable file
- Within memory, alongside of the program code and data

Blitz stores most of the debugging information in the executable file, but stores some information in memory with the target program's instructions.

The debugging info derived from pseudo-ops (such as **.function**, **.local**, and **.stmt**) is stored in the executable file. The KPL compiler automatically generates the debugging pseudo-ops so all programs carry the necessary information in their executable files.

There is also a concern with hand-coded assembly language routines. However, it is not a significant burden to include debugging pseudo-ops in hand-coded assembly functions.

The KPL compiler will place information about types and objects directly in memory in the form of **dispatch tables** and **class descriptors**. This is done because this information may be needed at runtime for other (non-error) operations, such as the **isKindOf** and **isInstanceOf** functions.

Generally speaking, storing the debugging information in the executable file is preferred over placing information in the program itself. Placing the information in memory at runtime increases the program size and increases the time to load the program, as well as enlarging the program's memory footprint.

However, placing the debugging information in the executable file requires participation by the assembler and linker. Since the debugging information contains information about the placement of code and variables in memory, the assembler and linker are required to carry this information through from the **.s** file and add it to the executable file. Also, at the time of an error, the debugger must read in the executable file, parse it, and build an internal representation.

We consider it mandatory that the debugger must always be invoked for any program that has an error. This means the hooks for error handling must be present in every program. The programmer must never be required to recompile the program with special options or rerun a faulting program.

In Blitz, the debugger is always invoked on error and begins by accessing the original executable file from which the program was loaded to obtain the necessary debugging information.

One issue concerns the question of locating the executable file from which the program was loaded.

(It is possible that the executable file will get modified or deleted between the time the program is loaded and the time the debugger is invoked. We place the burden of guarding against this on the programmer who is using the debugger.[4])

The key question the debugger must answer is:

Which source statement was executing at the time of the error?

The debugger must determine which source level statement was executing and within which function.

To accomplish this, the debugger builds a reverse mapping from **PC** values to source statements. From the **PC** value captured by the error handler, the debugger can search and determine the source statement and the identity of the function that contains that statement.

Experience has shown that naming the error and simply identifying the source statement line number is incredibly useful in debugging. This cannot be overstated.

This reverse mapping will fail if a bug causes a program to make a jump to a "random" location. In that case, the **PC** value is garbage.

But how likely is such a random jump? And how can it occur in KPL code?

We assume that user-level code is always kept in read-only pages so it can never be overwritten. Jump tables (e.g., dispatch tables or switch jump tables) are also kept in read-only pages. Therefore, these are not a source of random jumps.

Consider a program working with values of type "**ptr to function**". While a mistake may cause incorrect output, the KPL type checking system will prevent the program

---

[4] If the debugging information had been stored in memory alongside the code, this would not be a problem; the debugging information is already there when needed.

One approach is to disallow the debugger to be used on a program that was loaded in the past. In other words, to debug a program, the programmer must restart the program from within the debugger. But this has the shortcoming of making it difficult to debug transient errors. The bug may not manifest itself upon restarting the program. We must be able to begin debugging a failed program immediately, without having to restart it.

---

from taking a random jump which could confuse the debugger. However, if the programmer uses an "unsafe" operation on a function pointer, this could cause a program to take a random jump.[5]

This leaves return addresses stored in stack frames. Of course a bug can corrupt the stack and result in a RET instruction jumping to a random location.[6]

Random jumps are, in fact, almost non existent.

In practice, the Blitz debugger reports the location of errors very reliably.

Function Prologue and Epilogue

Often a function needs a stack frame to be pushed on the stack, in which to store local variables. The **sp** register is used to point to the current top of the stack.

In some processors, a second register is devoted to pointing to and accessing the stack frame. This register might be called the "frame pointer" (or "fp" register). The Blitz-64 architecture is designed so a second register is not needed. In Blitz, there is no "fp" register. Instead, the **sp** register is used to access the stack frame, as we describe next.

The sequence of instructions occurring at the beginning of a function is called the **function prologue**. The sequence of instructions occurring at the end of a function (executed directly before returning) is called the **function epilogue**.

The prologue creates and pushes a stack frame on to the stack when the function begins execution. The epilogue pops the stack frame off the stack before returning.

The same approach can be used for methods, as well as functions, so these sequences are sometimes called the **method prologue** and **method epilogue**. In this discussion, we'll just talk about functions, although the same works for methods. Sometimes, the terms **entry code sequence** and **exit/return code sequence** are used.

---

[5] Programs that perform unsafe pointer manipulations on function pointers are extremely rare and weird.

[6] Let's not forget that another source of random jumps in is the presence of a compiler error.

Of course, the programmer can place a **return statement** anywhere within a function and the function can contain many returns. In the following, we will place the function epilogue as if there is only a single return statement at the bottom of the function.

Most likely, the compiler will place a copy of the epilogue sequence at every place where a return statement occurs.[7]

Leaf Functions

A **leaf function** is defined as a function that does not call other functions. As such, the return address — which is in register **lr** on entry to the function — can remain in **lr** and does not need to be saved on the stack.

Here is the code that will be used for the entry and return in a leaf function.

```
# Leaf function
foo:
        …
        ret
```

There is zero prologue and epilogue overhead for a leaf function.

Note there is no need to touch or access memory, as long as all arguments and work variables are kept in registers.

---

[7] An alternative is for the compiler to include a single copy of the epilogue statements. The compiler will insert a JUMP to the epilogue sequence wherever a return statement is used. Since the epilogue is about 3 statements, inserting a JUMP instruction is generally considered too much overhead.

If the leaf function needs additional storage for locals and temporary variables, it can place these on the stack, ***above the stack top***, i.e., using negative offsets from register **sp**.
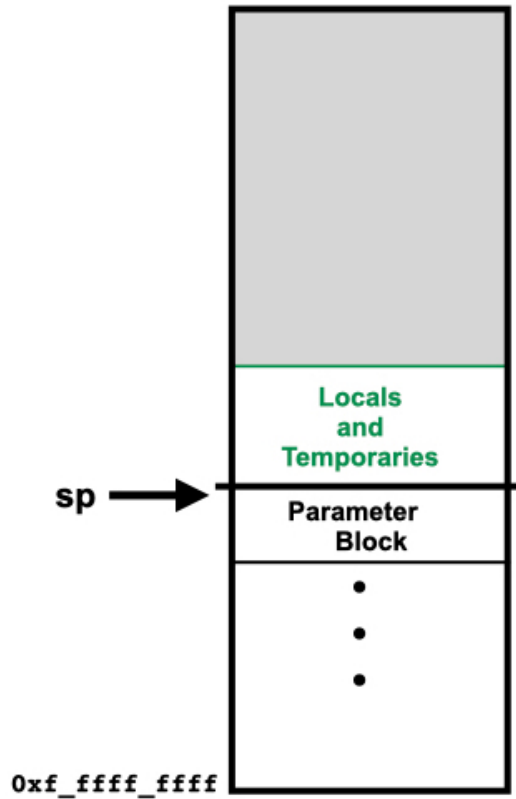
Non-Leaf Functions

If a function calls other functions, we call it a **non-leaf function**.[8]

For a non-leaf function, the code must save register **lr** and adjust the stack top pointer to push a new stack frame onto the stack:

```
# Non-leaf function
foo:
        store.d  -8(sp),lr
        addi     sp,sp,-FRAME_SIZE
        …
        addi     sp,sp,FRAME_SIZE
        load.d   lr,-8(sp)
        ret
```

In the above code, "FRAME_SIZE" is an integer which gives the size of the frame. The frame size and layout will be computed by the compiler. The compiler must compute the size needed to store parameters for each of the functions that "foo" invokes (the maximum size needed for all functions will become the size of the parameter block). The compiler will also determine the amount of storage needed for locals and temporaries within foo, plus 8 bytes in which to store the return address.

---

[8] The KPL compiler will often insert error checking tests and, if triggered, the code will execute a CALL to an error handler function. While the source code may not call any functions explicitly, any such implicit error-related CALLs will render the function a non-leaf function.

Here is what a stack frame looks like:
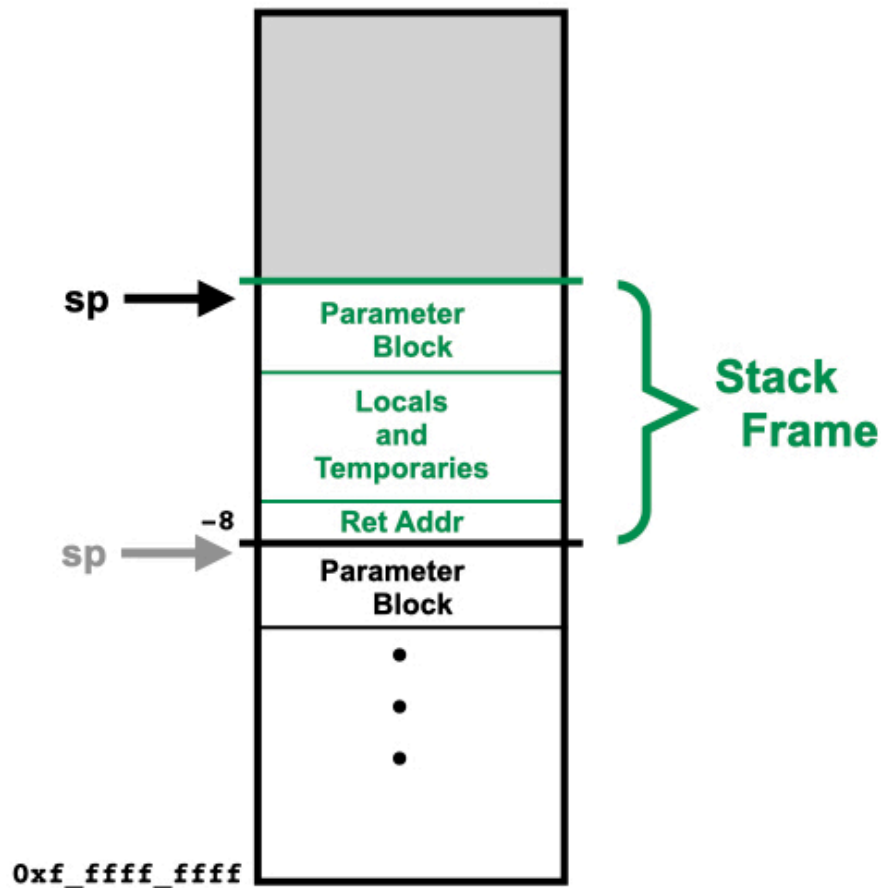
Within foo, the local and temporary variables will be accessed with positive offsets from **sp**. Access to the arguments to foo will also be made using positive offsets to **sp**. The exact offsets to the arguments can only be determined after the size of foo's frame has been determined.

The above code sequences will need a slight modification if FRAME_SIZE exceeds 32,767 since the ADDI instruction has that limit.[9]

Reconstructing the Call Stack

Note that this organization provides enough information for the debugger. After an error occurs, the debugger is given only:

[9] For larger frames, the compiler will need to generate an additional instruction for the prologue and an additional instruction for the epilogue. See the commentary in the ISA Reference Manual immediately following the description of the UPPER16 instruction for more information.

PC   The address at which the error occurred

sp   A pointer to the stack top at the time of the error

From the PC, the debugger will use the reverse mapping (described elsewhere in this document) to determine which source statement was executing and, from that, which function was currently active. From the function information, the debugger can determine the size of the stack frame, which will allow it to locate the slot containing the return address. Then, it can compute the stack top on entry to the function and the statement from which the function was called.

In this way, the debugger can work backwards through the stack, showing the entire call history.

Object Representation

Consider the following class definition:

```
class MyClass
    i: int
    b: bool
    w: word
    p: ptr to MyClass
    h: halfword
endClass
```

Every object will be located on a doubleword aligned address and all fields within the object will be properly aligned, according to their individual requirements. For example, the offset of the **word** field **w** will be an even multiple of 4, ensuring that it will be word aligned.

Each object of the class **MyClass** will have the five fields shown above, along with a hidden field, known as the "**dispatch table pointer**".

Every object will contain a dispatch table pointer, which will always be the first field in the object, i.e., the pointer will always be at offset 0 of the object. This pointer will be a 64-bit field containing the address of a "**dispatch table**".

Objects described by the above definition will be laid out as:

field	type	offset	size
<dispatch pointer>		0	8
i	int	8	8
b	bool	16	1
<padding>		17	3
w	word	20	4
p	ptr	24	8
h	halfword	32	2
<padding>		34	6
		40	*size of object*

There will be 0-7 bytes of padding added to force the size of every object up to a multiple of 8 bytes.

If the class is a subclass of another object, then all the fields of the superclass will be placed before the fields of the subclass. The size of the superclass will be a multiple of 8, which will ensure that the fields of the subclass (which follow) will be properly aligned.

There will only be one dispatch pointer and it will always be at offset 0.

The compiler will know the offset of every field in an object and these fields will always be properly aligned. Thus, the LOADx and STOREx synthetic instructions can be used directly to retrieve and update fields.

For example, assume that register **r1** contains a pointer to an object of type MyClass:

To retrieve the "int" field at offset 8:
```
        loadd    …,8(r1)
```

To update the "word" field at offset 20:
```
        storew   20(r1),…
```

Note that LOADx and STOREx are synthetic instructions. Any offset can be specified in the assembly code, up to the full range of memory. The assembler will generate only as many machine instructions as required. For any object under 32,767 bytes in size, a single instruction will suffice. Since it is unusual for objects to be this large, in most cases a single instruction will be used. However, notice that extremely large

objects will be automatically accommodated without additional measures or exceptions.

Method Invocation and Dynamic Dispatching

For every class definition, the compiler will produce a single dispatch table. The dispatch table will begin with a 64 bit field called the "**class pointer**". This pointer will be followed by a number of 64 bit fields, called "**jump slots**". Each jump slot will correspond to one message that objects of this class understand. The dispatch table will contain a jump slot for each message defined in the class, as well as a jump slot for each message defined in superclasses.

Each jump slot will contain a JUMP instruction. The JUMP instruction is a synthetic instruction that will be expanded to either one or two machine instructions. This expansion will be done by the linker, after it has determined the exact address of the target location.

Thus, the JUMP will be either 4 or 8 bytes. All jump slots are 8 bytes and, for JUMPs that require only 4 bytes, the linker will insert padding bytes.

The target of the JUMP will be the code for the corresponding method. That is, the JUMP will branch to the first instruction of the "entry prologue" sequence.

A message is very similar to a function. In fact, the code for a message is identical to the code for a function, with the exception that there is an additional argument. This argument is always the first argument and is a pointer to the receiving object itself.

Thus, the "**self variable**" is a pointer to the receiver and will be in register **r1** upon method entry. The first normal argument to the message will go into **r2**, with remaining arguments in **r3** … **r7**. In other words, arguments are passed to method exactly the same way they are passed to functions, with the addition of an additional argument (the self pointer) inserted before the other arguments.

Likewise, the remaining calling conventions and parameter passing rules are identical for both functions and methods.

The only difference is in the caller's code that invokes the method. When invoking a function named "foo", the caller's code looks something like this:

```
# Function Invocation
        mov     r1,…                    # Evaluate argument 1
        mov     r2,…                    # Evaluate argument 2
        mov     r3,…                    # Evaluate argument 3
        call    foo
        mov     …,r1                    # Retrieve returned value
```

Now let's consider invoking a method named "meth".

For each method, the compiler will determine the offset into the dispatch table. The code will jump indirectly through this table. We do this because the compiler must perform dynamic dispatching. The compiler cannot know the exact class of the object. Thus, the compiler doesn't know which dispatch table will be used or which method implementation will be executed. The compiler only knows the offset into the dispatch table where a JUMP to "meth" will be found.

Let us assume that the offset into the dispatch table for "meth" is some number "xxx". Then the following code sequence will perform message sending.

```
# Method Invocation
        mov     r1,…                    # Evaluate ptr to receiver
        mov     r2,…                    # Evaluate argument 1
        mov     r3,…                    # Evaluate argument 2
        loadd   s0,0(r1)
        jalr    lr,xxx(s0)
        mov     …,r1                    # Retrieve returned value
```

The LOADD instruction will move a pointer to the dispatch table into register **s0**. The JALR instruction will save the return address in the linker register **lr** and jump directly to an entry in the dispatch table. This entry will be the jump slot for "meth" and will contain a jump to the appropriate code. In other words, this code performs a "call" to the jump slot itself. Then, immediately, a jump is made to the first instruction of the appropriate method.

Thus, the overhead for a message send, above what is required for a function call is typically only two additional instructions:

Function call:
```
        call/jal
```

Message send (typical):
```
        loadd           # load ptr to dispatch table
        jalr            # jump to jump slot
        jump/jal        # jump to method prologue
```

[In comparing a method invocation to a function invocation, we are ignoring the additional code to load the pointer to the receiver object. If we are using a method instead of a function, then the assumption is that there is some object involved (i.e., the receiver object) and this object would have been passed as a normal argument had the programmer coded this as a function. In any case, a single instruction will often be used to load register **r1** regardless of whether it is a function or a method.]

A CALL instruction will normally expand to a single JAL instruction, but in some cases it may expand into two instructions.

Recall that the JALR instruction contains a 16-bit immediate field, ranging -32,768 … +32,767. The above code sequence for a message send will work as long as the offset into the dispatch table doesn't exceed this number. (In particular, the dispatch table cannot contain more than 4,094 jump slots, plus the class pointer.) It is unlikely that any class will have (or inherit) this many methods. But if so, the compiler will have to insert an additional UPPER16 instruction.

Normally, the jump slot will contain a single JAL instruction, which can branch up to -524,288 … +524,287 bytes relative to the jump slot's location. The compiler will typically place the dispatch table and the methods it references in the same segment, so they will end up near each other in memory. So in most cases the jump slot will contain only a single instruction, but in some cases it may contain two.

Thus, the very **worst case scenario** is that a message send requires four more instructions than a function invocation.

Function call:
```
        call/jal
```

Message send:
```
        loadd   s0,0(r1)            # load pointer to dispatch table
        upper16 t,s0,xxx            # call to jump slot
        jalr    lr,xxx(t)           # .
        upper20 t,yyy               # jump to method prologue
        jalr    lr,yyy(t)           # .
```

But keep in mind that the CALL itself might have a long distance target and require two instructions.

Object Initialization

In KPL, objects must be initialized before being used. The initialization is nothing more than initializing the dispatch table pointer. Without a valid dispatch table pointer, methods cannot be invoked on the object.

The KPL compiler will insert a test to make sure the object has been initialized. This test is inserted in every code sequence that invokes a method. This test requires an additional instruction to test the dispatch table pointer to make sure it is not null.

For clarity, this test was not shown in the above code examples.

If the dispatch table pointer is null, error handling will be invoked. In particular, an error will be thrown. The error is named **ERROR_UninitializedObject**. Perhaps the program will catch this error, but if not, it will result in the debugger becoming active.

[Without the explicit test, what would happen? Since the dispatch table pointer is missing, register **s0** will be loaded with zero. Then, using some offset (**xxx**), a jump will be made. This would result in a jump to absolute address **xxx**. Assuming this is user-mode code running in a virtual address space, this will cause a "Page Illegal Address Exception". Unfortunately, the location of the actual error would be lost. Would it be wise to add an option to the KPL compiler to give programmers the ability to leave these tests out? This was considered and rejected.]

Compilation Examples

In this section, we give some examples code fragments and suggest how a compiler might translate them into assembly language. The higher-level code is expressed in KPL, the programming language of Blitz-64, although any similar language (like "C" or "C++") could have been used.

These examples are intended to show how the Blitz-64 ISA can be used; they are not necessarily the way the KPL compiler actually works.

For the purposes of this appendix, we define "basic types" as:

int	64-bit signed integers
word	32 bit quantities
halfword	16 bit quantities
byte	8 bit quantities
bool	TRUE / FALSE, stored in a byte
double	64 bit double-precision floating point
ptr	Pointer to anything, stored in 64 bit doubleword

Non-basic types are defined as follows. Their sizes will vary:

arrays
structs / records
unions
objects
… anything else …

Access of Variables

Global variables (i.e., variables defined outside any function or method) will be allocated in fixed, unchanging locations in memory. This can be done with a single pseudo-op.

KPL:
```
var
   i: int
   w: word
   h: halfword
   c: byte
   b: bool
   d: double
   p: ptr to …
   a: array [ … ] of …
```

Assembly translation:
```
i:    .doubleword      0
w:    .word            0
h:    .halfword        0
c:    .byte            0
b:    .byte            0
d:    .double          0.0
p:    .doubleword      0
a:    .skip            …
```

(Global variables are called "static variables" by some people.)

In KPL, all variables are assumed to be initialized to zero values. The above translations work because **.skip** is guaranteed to fill the space with zeros.

If the programmer provides an initial value, this value can always be determined by the compiler and the translation will cause the global variable to be initialized when the program is loaded, before execution begins.

KPL:
```
i: int = MAX_SIZE-1
```

Assembly translation:
```
i:    .doubleword       99
```

The translation of a simple assignment involving a global variable of basic type will involve the use of a register, as in:

KPL:
```
i = i + 7
```

Assembly translation:
```
        loadd       r2,i
        addi        r2,r2,7
        stored      i,r2
```

NOTE: The LOADx and STOREx instructions are synthetic instructions. They can be used to access any location in memory. In many cases, the synthetic will expand to a single machine instruction, but for some harder-to-reach addresses, a second instruction will be automatically inserted by the linker. Thus there is no limit imposed by the ISA, assembler, or linker on global variable access.

Local variables are handled differently. In some cases, the compiler will be smart enough to place the variable in a register and avoid all memory references.

KPL:
```
function foo (…)
   var local: int
   …
   local = local + 7
```

Assembly translation:
```
        addi          r5,r5,7        # assumes "local" is in r5
```

In other cases, the local variable will be placed on the runtime stack. ("**Stack frames**" are often called "**activation records**".)

[Stack frames will be discussed later, but the basic idea is that a stack is maintained for the duration of program execution. This is a stack of "frames" and the top of the stack is pointed to by register **sp** (i.e., "**r15**"). When a function is called, a new stack frame is pushed onto the stack and when the function returns, the frame is popped off the stack. The sp register will point to the first byte of the stack frame (i.e., the byte with the lowest address). All locations within the frame will accessed using positive offsets. The "pushing" of a new stack frame is a quick and simple operation, requiring only that the **sp** register be decremented by the frame size. Likewise, "popping" is accomplished quickly by simply incrementing **sp** by the same amount.]

The compiler may determine that a local variable cannot be kept in a register. In such cases, it will allocate some space within the stack frame for the variable. This can be because:

- The variable is not a basic type.
- There are not enough registers available.
- Some code asks for the address of the variable (using the "&" operator in KPL).

By "**basic type**" we mean:

basic type	size in bytes
int	8
word	4
halfword	2
byte	1
bool	1
double	8
ptr to ...	8

KPL also supports the following types, which are "**compound types**":

array
object
struct / record
union

As an example, assume that variable **local** has been placed at offset **16** within the frame. Now the compiler will need to issue LOAD and STORE instructions to access the variable.

KPL:
```
function foo (…)
   var local: int
   …
   local = local + 7
```

Assembly translation:
```
        loadd      r5,16(sp)    # assumes "local" is in the frame
        addi       r5,r5,7
        stored     16(sp),r5
```

NOTE: The LOADx and STOREx instructions are synthetic instructions. They can be used to access any offset from **sp**. In most cases, the synthetic will expand to a single machine instruction. Occasionally a stack frame may exceed 32 KiBytes in size and a second instruction will be automatically inserted by the linker. Frame sizes above 2 GiBytes in size are not expected, but will be handled by the linker, which will automatically insert a third machine instruction. So there is no limit imposed by the ISA, assembler, or linker on frame sizes and offsets.

Parameters will be either passed in registers or placed on the stack. Details will be discussed later. But the accessing of the parameter variables will use these same instructions.

Arithmetic Computation

The Blitz-64 ISA and the KPL language have been designed together, to work together. The arithmetic and logical operators of KPL correspond exactly in semantics to the machine instructions in the ISA.

KPL:	**Machine Instruction**
i + j	add r1,r2,r3
i – j	sub r1,r2,r3
i * j	mul r1,r2,r3
i / j	div r1,r2,r3
i % j	rem r1,r2,r3
–i	neg r1,r2
i & j	and r1,r2,r3
i | j	or r1,r2,r3
i ^ j	xor r1,r2,r3
!(i)	bitnot r1,r2
!(b)	lognot r1,r2
i << j	sll r1,r2,r3
i >> j	srl r1,r2,r3
i <<< j	sla r1,r2,r3
i >>> j	sra r1,r2,r3
i == j	beq r1,r2,label
i != j	bne r1,r2,label
i < j	blt r1,r2,label
i <= j	ble r1,r2,label
i > j	bgt r1,r2,label
i >= j	bge r1,r2,label
b = (i==j)	testeq r1,r2,r3
b = (i!=j)	testne r1,r2,r3
b = (i<j)	testlt r1,r2,r3
b = (i<=j)	testle r1,r2,r3
b = (i>j)	testgt r1,r2,r3
b = (i>=j)	testge r1,r2,r3
d + e	fadd r1,r2,r3
d – e	fsub r1,r2,r3
d * e	fmul r1,r2,r3
d / e	fdiv r1,r2,r3
–d	fneg r1,r2

```
d == e                          feq        r1,r2,r3
d != e                          feq        r1,r2,r3
d < e                           flt        r1,r2,r3
d <= e                          fle        r1,r2,r3
d > e                           fgt        r1,r2,r3
d >= e                          fge        r1,r2,r3
```

In particular, the error and boundary cases are carefully designed to match exactly. For example, for many KPL operators, overflow is required to "throw an error". [The TRY-THROW-CATCH mechanism in KPL is discussed elsewhere.]

The Blitz-64 ISA specifies that the corresponding machine instruction will cause an exception. For example, KPL requires integer addition to throw an error in the case of overflow; likewise, the Blitz-64 ISA requires the ADD and ADDI instructions to signal an Arithmetic Exception when overflow occurs.

In the course of translating some arithmetic expressions, the compiler will need to store temporary results. In the following example, no temporary storage is needed:

KPL:
```
i = (i + j - k) * m
```

Assembly translation:
```
# Assume i: r1
# Assume j: r2
# Assume k: r3
# Assume m: r4
      add          r1,r1,r2
      sub          r1,r1,r3
      mul          r1,r1,r4
```

However, in the next example, the result of the addition must be kept in a temporary location, until after the subtraction is performed. In many cases, the compiler will be able to keep this temporary value in a register. In this example, the compiler has chosen to use register **t** (i.e., **r8**).

KPL:
```
  i = (i + j) * (k - m)
```

Assembly translation:
```
  # Assume i: r1
  # Assume j: r2
  # Assume k: r3
  # Assume m: r4
  # Assume temp: t
        add           r1,r1,r2
        sub           t,r3,r4
        mul           r1,r1,t
```

The compiler may be able to use a register to store the temporary result, as in the previous example. However, if no additional registers are available, the compiler will be forced to allocate space in the stack frame and store the temporary result there.

in the next example, the complier has set aside space in the stack frame at offset 24 to temporarily store the value of (i+j) until it is needed.

KPL:
```
  i = (i + j) * ((k - m) / (n + p))
```

Assembly translation:
```
  # Assume i: r1
  # Assume j: r2
  # Assume k: r3
  # Assume m: r4
  # Assume n: r5
  # Assume p: r6
  # Assume (i+j) is at offset 24 in stack frame
        add           t,r1,r2      # temp = i + j
        stored        24(sp),t     # save temp in frame
        sub           r1,r3,r4     # r1 = k - m
        add           t,r5+r6      # t = n + p
        div           r1,r1,t      # r1 = (k-m) / (n+p)
        loadd         t,24(sp)     # retrieve temp = i+j
        mul           r1,t,r1      # r1 = (i+j) * ((k-m) / (n+p))
```

In the above example, you will notice that all operations are done in the same order specified by the source code. The compiler maintains the same order to ensure that the overflow behavior at runtime will be exactly what the programmer expects.

For example:

```
(a + b) + c
```

may not overflow while the following will cause an overflow exception:

```
(a + c) + b
```

(This can happen when **a** and **c** are very large numbers and **b** is a very negative number.)

In some cases, the compiler may be able to perform some operations at compile time or may be able to re-order the operations with no fear of changing the overflow behavior. For example, the following:

```
(a + 123) + 456
```

will overflow in exactly the cases that the following will overflow:

```
a + 579
```

As long as there is no change in the behavior of the program, including exceptional and error behavior, the compiler is free to reorder the operations.

In most programming languages, wherever the programmer can specify a variable, he or she can insert a function call instead:

```
i + j + k
i + foo1(…) + foo2(…)
```

Whenever a function is called, it tends to involve a lot of register usage, forcing the compiler to move temporary results into "save" locations in the stack frame.

The KPL compiler avoids rearranging expressions since it does not always fully understand what the code is doing. In the above example, KPL guarantees that **foo1** will be called after the value of **i** is retrieved and before **foo2** is invoked. After all, **foo1** might have some side-effect that alters the behavior of **foo2**, or even the value of variable **i**.

Flow of Control Examples

Conditional statements can be translated as shown in this example:

KPL:

> **if** (*...condition...*)
>   *...Then statements...*
> **endIf**

Translation Idea:

```
        ...Evaluate condition...
        if true goto Then_label
        if false goto Endif_label
Then_label:
        ...Then statements...
Endif_label:
```

If there are "else statements", the general form is a little more complicated:

KPL:

> **if** (*...condition...*)
>   *...Then statements...*
> **else**
>   *...Else statements...*
> **endIf**

Translation Idea:

```
        ...Evaluate condition...
        if true goto Then_label
        if false goto Else_label
Then_label:
        ...Then statements...
        jump Endif_label
Else_label:
        ...Else statements...
Endif_label:
```

For example:

KPL:

    if (i < j)
       i = 23
    else
       i = j + 45
    endIf

Assembly translation:

```
# Assume i: r1
# Assume j: r2
        bge     r1,r2,_label_67
        movi    r1,23
        jump    _label_68
_label_67:
        addi    r1,r2,45
_label_68:
```

In order to translate flow-of-control statements, the compiler will often create new labels and give them automatically generated names, such as "_label_67".

Note the reversal of the condition testing in the above example. The "less than" test with a branch to the "THEN" statements is changed to a "greater-than-of-equal" test to the "ELSE" statements.

Condition		Reversed Condition	
==	beq	!=	bne
!=	bne	==	bge
<	blt	>=	bge
<=	ble	>	bgt
>	bgt	<=	ble
>=	bge	<	blt

Notice that , if done literally, the translation of:

    if (i < j) **then** ...

according to the general form:

```
        ...Evaluate condition...
        if true goto Then_label
        if false goto Else_label
    Then_label:
```

is this:

```
        blt     r1,r2,_label_66
        bge     r1,r2,_label_67
    _label_66:
```

But simple patterns like this can be reduced. In this case, the following is equivalent:

```
        bge     r1,r2,_label_67
```

With floating point numbers, we have the following instructions which implement operations directly.

<u>Condition</u>
```
==  feq
<   flt
<=  fle
>   fgt
>=  fge
```

Blitz-64 does not contain a FNE instruction. Equal and not-equals are logical opposites, so we use FEQ to implement !=. However, with floating point, note that < (FLT) and >= (FGE) are <u>not</u> opposites. Likewise, <=(FLE) and > (FGT) are not opposites. The difference arises when one argument is not-a-number (NaN). So the compiler must be careful not to switch FLT into FGE, or switch FLE into FGT.

There are a number of other types of conditional expressions and there are a number of specialized Blitz-64 instructions that are designed specifically to support them. For example, a boolean variable can be tested.

<u>KPL Example:</u>
    if (boolVar) …

<u>Relevant Assembly Instructions:</u>
```
        btrue           Reg1,Label
        bfalse          Reg1,Label
```

A pointer can be tested directly and these same instructions can be used for that. Note that these instructions compare against zero. Thus, non-null pointers will be interpreted as TRUE and null pointers will be interpreted as FALSE.

<u>KPL Example:</u>
    if (ptr) …

<u>Relevant Assembly Instructions:</u>
```
        btrue           Reg1,Label
        bfalse          Reg1,Label
```

When the source code compares an integer to a constant value, it will typically require an additional MOVI instruction, as in:

<u>KPL Example:</u>
    if (i == 123) …

<u>Assembly Translation:</u>
```
        movi            t,123
        beq             Reg1,t,Label
    or
        bne             Reg1,t,Label
```

However, if the comparison is against zero, there are specialized Blitz-64 instructions which can be used instead, avoiding the MOVI instruction.

KPL Examples:
    if (i == 0) ...
    if (i < 0) ...
    ...etc...

Relevant Assembly Instructions:
```
        beqz            Reg1,Label
        bnez            Reg1,Label
        bltz            Reg1,Label
        blez            Reg1,Label
        bgtz            Reg1,Label
        bgez            Reg1,Label
```

Sometimes the programmer will evaluate a conditional expression and want the result in the form of a boolean value, not in the form of branching. There are specialized Blitz-64 instructions which make that sort of operation easy:

KPL Examples:
    boolVar = (i >= j)
    return i<j

Relevant Assembly Instructions:
```
        testeq          RegD,Reg1,Reg2
        testne          RegD,Reg1,Reg2
        testlt          RegD,Reg1,Reg2
        testle          RegD,Reg1,Reg2
        testgt          RegD,Reg1,Reg2
        testge          RegD,Reg1,Reg2
```

According to the semantics of KPL, all subexpressions in a larger expression must be evaluated in the order in which they appear in the source. The following are not equivalent, and the code must perform the function invocation in the order given.

    if (foo(...) && bar(...)) ...
    if (bar(...) && foo(...)) ...

With the short-circuit AND operator (&&), whenever the first operand is evaluated and found to be FALSE, the second operand need not be evaluated, since the result will be FALSE regardless. The KPL language specifies that the second operand must definitely not be evaluated whenever the first is FALSE.

Likewise, with the short-circuit OR operator (||), whenever the first operand is evaluated and found to be TRUE, the second operand need not be evaluated, since the result will be TRUE regardless. The KPL language specifies that the second operand must definitely not be evaluated whenever the first is TRUE.

With the use of short-circuit operators, the evaluation of conditional expressions becomes more complex, as the next example illustrates.

KPL:

```
if  ( (i < j) && (i == k) ) || ( (k < m) && (i == m) )
    i = 23
else
    i = j + 45
endIf
```

Assembly translation:

```
# Assume i: r1
# Assume j: r2
# Assume k: r3
# Assume m: r4

        bge     r1,r2,_label_65
        beq     r1,r3,_label_66
_label_65:
        bge     r3,r4,_label_67
        bne     r1,r4,_label_67

# THEN STMTS...
_label_66:
        movi    r1,23
        jump    _label_68

# ELSE STMTS...
_label_67:
        addi    r1,r2,45

# ENDIF...
_label_68:
```

In the above example, the full benefit of the short-circuit operators is not demonstrated, since the operands are all simple variables that are read-only. But keep in mind that the programmer could substitute function invocations for each

operand, thus involving arbitrary computation. Thus, short-circuit behavior is required from && and ||.

The translation of a "**while loop**" follows this general form:

KPL:

> **while** (*...conditional...*)
>   *...statements...*
> **endWhile**

Translation Idea:

```
        goto Continue_label
Loop_label:
      ...statements...
Continue_label:
      if (...conditional...) goto Loop_label
Exit_label:
```

For example:

KPL:

> **while** (i < j)
>   *...BodyStatements...*
> **endWhile**

Assembly translation:

```
      goto    _label_35        # goto Continue_label
_Label_34:
      ...BodyStatements...
_Label_35:
      blt     r1,r2,_label_34   # If i<j  goto Loop_Label
_Label_36:
```

A loop containing a "**break statement**" will cause a jump to the "Exit_label". A loop containing a "**continue statement**" statement will cause a jump to the "Continue_label". For example:

KPL:

> **while** (*...Conditional...*)
>
> > *...*
> > **break**
> > *...*
> > **continue**
> > *...*
> **endWhile**

Translation Idea:

```
        goto    Continue_label
Loop_label:
        ...
        jump    Exit_label        # Break
        ...
        jump    Continue_label  # Continue
        ...
Continue_label:
        if (...conditional...) goto Loop_label
Exit_label:
```

KPL contains a "**do-until**" statement, which is similar to a "do-while" or "repeat-until" statement. The translation follows this general form:

KPL:

> **do**
> > *...statements...*
> **until** (*...conditional...*)

Translation Idea:

```
Loop_label:
        ...statements...
Continue_label:
        if !(...conditional...) goto Loop_label
Exit_label:
```

Here is an example of a "do-until" statement containing a short-circuit operator in the condition:

KPL:

> **do**
>   *...BodyStatements...*
> **until** (i < j) && (k == m)

Assembly translation:

```
_Label_34:
      …BodyStatements…
_Label_35:
      bge     r1,r2,_label_36    # If i>=j  goto Loop_Label
      bne     r3,r4,_label_34    # If k!=m goto Loop_Label
_Label_36:
```

The translation of a "**for loop**" follows this general form:

KPL:

> **for** (*...InitializationStmts...* ; *...Conditional...* ; *...IncrementStatements...*)
>   *...*
>   **break**
>   *...*
>   **continue**
>   *...*
> **endWhile**

Translation Idea:

```
      …InitializationStmts…
      goto Check_label
Loop_label:
      …
      jump    Exit_label        # Break
      …
      jump    Continue_label    # Continue
      …
Continue_label:
      …IncrementStatements…
Check_Label:
      if (…Conditional…) goto Loop_label
Exit_label:
```

There are several ways to translate a "**switch statement**". The simplest translation involves performing a series of tests.

KPL:

> **switch** (*…TestExpr…*)
>
> > *…*
> > **case** (*…ExprN…*):
> > > *…StatementsForCaseN…*
> >
> > *…*
> > **default**:
> > > *…DefaultStatements…*
> > **endSwitch**

Translation Idea:

> *…Evaluate TestExpr…*
>
> *…Evaluate Expr1…*
> ```
> if (TestExpr != Expr1) goto Case_1
> ```
> *…Evaluate Expr2…*
> ```
> if (TestExpr != Expr2) goto Case_2
> ```
> *…Evaluate Expr3*
> ```
> if (TestExpr != Expr3) goto Case_3
> ```
> *…Evaluate Expr4*
> ```
> if (TestExpr != Expr4) goto Case_4
> ```
>
> ```
> jump Case_Default
> ```

```
Case_1:
```
> *…StatementsForCase1…*
```
Case_2:
```
> *…StatementsForCase2…*
```
Case_3:
```
> *…StatementsForCase3…*
```
Case_4:
```
> *…StatementsForCase4…*

```
Case_Default:
```
> *…DefaultStatements…*
```
Exit_label:
```

Any "break" statement within any of the code blocks is just translated into a JUMP to "Exit_label". Any code block not ending with a "break" will simply fall through to the next code block.

As you can see, a translation based on this scheme will execute the switch by testing each possible value in turn. Of course, whenever there are more than just a couple of cases, this will result in poor performance. There are better translation schemes for the switch statement.

The decision about which translation scheme is best to use can depend on the number of cases and other factors. If the various case values all happen to fall with a small range of integer values, a superior translation approach is to create a "jump table" of indirect pointers. The code will first compute the value of "*TestExpr*" and then use that value as an index into the jump table. Then the code will branch directly to the correct statement block. For switch statements with hundreds of cases, this approach to translation is clearly superior. We will not discuss this translation technique any further here, although it is the key to making switch statements work well.

Chapter 9: Format of Object Files

Quick Summary

• Object files use the extension "**.o**".
• Running the assembler tool will produce an object file.
• The linker tool takes one or more object files as input.
• Running the linker tool will produce an executable file.
• Each object file contains the following:
     — Information about each segment
     — The data bytes of each segment
     — Information about each symbol
     — Information about each patch
     — Info to support the runtime debugging (optional)
• A "patch" is a relocation entry, telling how to modify the bytes in an instruction.
• The assembler creates a patch entry for every instruction it cannot complete.
• Each synthetic instruction will result in a single patch.
     — The assembler will fully translate some synthetic instructions,
        in which case no patch is necessary.
• The linker has more information available to it than the assembler.
• The linker will first place the segments in memory.
• Once placed, the value of every symbol will become known.
• The linker will process each patch, updating the bytes in memory.
• The linker will complete by creating an executable file.

Terminology and Files

This chapter describes the format of the object file. The object file generally ends with a ".o" extension. The format of the executable file is described in a different chapter.

For example, an assembly source file named "simple.s" would typically be used to produce an object file named:

simple.o

The object file is used as input to the Blitz-64 linker, which produces an "executable file". The linker will take one or more object files, and will produce a single executable file.

The executable file is often call the "a.out" file, although it is generally given a more meaningful name. Often the name of the executable file is the same as one of the original source files, after removing the ".o". For example:

simple

An extension is optional, but if present, **.exe** is recommended. For example, the output file might be given this named instead:

simple.exe

At some later time, the executable file will be loaded by an operating system and executed. Therefore, it must contain all that is necessary for executing the program.

The Blitz-64 assembler tool is called "asm" and the linker tool is called "link". Another Blitz-64 tool, called "dumpobj", can be used to print out, in a human readable form, either object or executable files.

The Object File

The object file has the following format. The file can be considered as series of fields. The length of each field is given in the left-hand column.

bytes   field description

The following fields constitute the header information…
8	Magic number "B64objct" (in hex: 0x4236_346F_626A_6374)
8	Version Number (0x0000_0000_0000_0001)
2	Blitz-64 ISA Architecture (e.g., 0x0002)
4	Number of segments
4	Number of symbols (0 … 2,147,483,647)
4	Source file name: number of characters (M); 0=source came from stdin
M	Source file name: the ASCII characters (no terminating \0)
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every segment…
4	Segment number (1, 2, 3, …)
4	Source file line number
8	Length of segment in bytes (possibly zero)
1	Is Kernel (0=user, 1=kernel)
1	Is Executable (0=not executable, 1=executable)
1	Is Writable (0=read-only, 1=read and write)
1	Is Zero-filled (0=normal, 1=all data is zero)
8	Starting address from "startaddr=" (-1 = floating)
8	Assumed value of "gp" from "gp=" (-1 = undefined, -2=default)

After all segments…
4	Zero to terminate (in hex: 0x00000000)
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every symbol...
  4   Symbol_number (1, 2, 3, ...)
  4   Source file line number
  1   Type:
             1 = imported
             2 = label
             3 = equate (definition appeared in .equ)

  If type = 1 (imported)...

  If type = 2 (label)...
    4   Segment number in which symbol was defined
    8   Offset into segment (where label occurred)
    1   Was this symbol exported (0 = local only, 1 = exported)

  If type = 3 (equate)...
    4   RelativeTo symbol number (0 = offset is an absolute value)
    8   Offset (from relativeTo symbol, or value if absolute)
    1   Was this symbol exported (0 = local only, 1 = exported)

  4   Symbol name: number of characters (L)
  L   Symbol name: the ASCII characters (no terminating \0)

After all symbols...
  4   Zero to terminate (in hex: 0x00000000)
  8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every patch...
  1   The patch type (1, 2, ...)
  4   Source file line number
  4   The segment where the patch must be made
  8   The location to be patched (i.e., offset into the segment)

  4   The target symbol (0 = absolute)
  8   Offset from target symbol (often zero)
             For patch type = "align", offset will be 8, 16, 32, or 16384

  1   Exact size of result in bytes (4, 8, 12, 16) or -1 if don't care
             Used for Formats S1,S2,...S7. For ALIGN this will be -1.

After all patch entries….
    1   Zero to terminate (in hex: 0x00)
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every segment…
    4   Segment number
    N   The data bytes, where N is the size of the segment in bytes
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields concern debugger information…
    4   Package name: number of bytes (M); 0 = No debugger info present
    M   Package name: the UTF-8 encoded characters (with terminating \0)
    4   The second string: number of bytes (N)
    N   The second string: the UTF-8 encoded characters (with terminating \0)
    4   The number of globals; 0 = none present / missing info
    4   The number of functions; 0 = none present / missing info
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every global…
    4   Global name: number of bytes (M); will be > 0
    M   Global name: the UTF-8 encoded characters (with terminating \0)
    4   Source file line number
    1   Type Code (One character code, e.g. 'I')
    4   Location: The segment number
    8   Location: Offset into segment

After all global entries…
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every function…
    4   Function name: number of bytes (M); will be > 0
    M   Function name: the UTF-8 encoded characters (with terminating \0)
    4   Source file line number
    4   Location: The segment number
    8   Starting Location: Offset into segment
    8   Beyond Location: Offset into segment (i.e., address of last byte + 1)
    4   Frame size (not negative; 0 = leaf function)

The following fields are repeated once for every register parameter…

4	Source file line number (>= 0)
1	Register number (1 … 15)
4	Parameter name: number of characters (M); will be > 0
M	Parameter name: the UTF-8 encoded chars (with terminating \0)
1	Type Code (One character code, e.g. 'I')

After all register parameters…

4	-1 to terminate
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every local variable…

4	Source file line number (>= 0)
4	Offset from stack top
4	Variable name: number of bytes (M); will be > 0
M	Variable name: the UTF-8 encoded chars (with terminating \0)
1	Type Code (One character code, e.g. 'I')

After all local variables…

4	-1 to terminate
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every statement…

4	Source file line number (>= 0)
4	Location of code: Segment number
4	Location of code: Offset into segment
1	Type Code (0=comment, 1=assign, …)

If and only if type code = comment, the following will be present…

4	Comment String: number of bytes (M); will be > 0
M	Comment String: the UTF-8 encoded chars (with terminating \0)

After all local statements…

4	-1 to terminate
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After all function entries…

4	Zero to terminate
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

Integers

All integers in the file are stored as signed binary values in Big Endian order, i.e., the most significant byte will appear first.

Integers of the following sizes are used:

	number of bytes	number of bits
byte	1	8
word	4	32
doubleword	8	64

Magic Number

The first eight bytes of the object file serve to identify it as a Blitz-64 object file. These bytes are the ASCII character codes for the letters "B64objct" (for Blitz-64 Object), namely the value 0x4236_346F_626A_6374.

The magic number idea is not a foolproof way to identify files. Although highly unlikely to occur by chance, there may happen to be other files that happen to begin with these same eight bytes. Although this techniques is by no mean secure, it is a good way for the linker to check that it is being given a meaningful file. Also, it allows a human looking at the file to guess what sort of data it contains. Although much of the file will contain bytes that are not interpretable as text data, the eight bytes of the magic number are human-readable, so they should give the reader a clue about the file's nature.

This technique is also used in other files:

	Magic Number	**ASCII Interpretation**
object file	0x4236_346F_626A_6374	"B64objct"
executable file	0x4236_3461_2e6f_7574	"B64a.out"
load-and-go file	0x4236_346C_642B_676F	"B64ld+go"
object library file	0x4236_346F_5F6C_6962	"B64o_lib"

The Version Number and ISA Architecture Fields

Following the magic number is a "version number". We understand that future changes may be required to the format of object files. This field exists to accommodate changes, updates, and extensions to this file format.

This document describes "version 1" of the file format. All files conforming to this specification will have the value 1 in this field. Any other value indicates that the remainder of the file will conform to a different specification.

At this time, there is only one version of this file format and the assembler and linker are is only capable of dealing with "version 1" files. Future versions may be capable of handling different versions.

Details about future version and compatibility between the tools must be documented in the future, obviously.

The "ISA Architecture" field specifies which type of machine this code is intended to be run on. This value must match the value from the version number in bits [30:16] of the CSR register **csr_version**. In other words, the numbers used in this field and the in **csr_version** are drawn from the same set and therefore have the same values and meanings.

At this time, the current version Blitz-64 Instruction Set Architecture (ISA) is

0x0002

In the future, changes and/or additions to the machine code instructions are likely. For example, we plan to specify and implement the compressed instruction set in the future. When changes are made to the ISA , the **csr_version** will be changed (incremented) to reflect a modified architecture.

Commentary We separate out the "file version number" and the "ISA architecture version" into two fields because these really track two different kinds of changes that can be made in the future. A change to the machine architecture may not require a change to the file format. Conversely, a change to the file format may be implemented even though there is no change to the ISA.

Separators (********)

As an internal consistency check, there will be 8 bytes of "separator" data placed at the indicated points in the file. These eight bytes are the ASCII character codes for the characters "********". That is, the separator doubleword is 0x2A2A_2A2A_2A2A_2A2A.

If there is some inconsistency between the text or data segment sizes and the actual number of bytes provided, then these separators may help to catch the error. The linker will check that the separator characters appear correctly at the places in the object file where they are supposed to appear, and print error messages if not.

Segment Information

Each segment in the object file is given a sequential number, starting with 1.

[Typically we expect the number of segments to be under 10. There will be at least one segment in the file and it is likely that other constraints will prevent the upper limit of 2,147,483,647 segments ever being reached.]

Each segment corresponds to a single **.begin** instruction in the source file.

A segment represents a block of bytes containing instructions and/or data. The segment will be loaded into memory by the operating system at the time the program is to be executed.

The block of bytes will appear in the object file and the linker will copy the block to the executable file. However, if the segment is marked "**zero-filled**", the block of bytes (which will all be 0x00) will not be stored in the object file or in the executable file. At execution time, the operating system will initialize the bytes of the segment to zeros, as it allocates memory pages for the process.

The linker will determine where in memory to place each segment. The following pieces of data will be used by the linker to determine where to place the segment:

- Length of segment in bytes
- Starting address from "startaddr="
- Is Kernel (0=user, 1=kernel)
- Is Executable (0=not executable, 1=executable)
- Is Writable (0=read-only, 1=read and write)
- Is Zero-filled (0=normal, 1=all data is zero)

The length of the segment is given in bytes and may even be zero, although why a programmer would create such a segment is hard to imagine.

The **.begin** instruction may include the "**startaddr=**" parameter. If so, the programmer has specified exactly where in memory to place the segment.

If the **startaddr=** parameter is undefined, the linker will rely the "**Is Kernel**" value. Kernel segments will be placed in low memory, as near to address 0x0_0000_0000 as possible. User segments will be placed in the virtual address region, which begins at address 0x8_0000_0000.

The linker will begin by placing the segments with predetermined addresses at their locations. Then the linker will place floating segments (i.e., segments without a **startaddr=** parameter) in the remaining area. (The placement algorithm is described elsewhere in this document.)

The linker is aware of pages and the fact that each page will either be marked "**writable**" or not, and that each page will either be marked "**executable**" or not.

The following kinds of pages are possible. The linker will determine how many pages are required at runtime and will only place like segments in any page.

	writable	**executable**
read-only	no	no
read-write	yes	no
code-only	no	yes
code-and-data	yes	yes

Each segment will also have a "**gp=**" parameter. This parameter will have as its value a 36 bit address (0x0 … 0x0000_000F_FFFF_FFFF). This parameter may also be "undefined", in which case the object file will contain the value -1

(0xFFFF_FFFF_FFFF_FFFF). This parameter may also be "default", in which case the object file will contain the value -2 (0xFFFF_FFFF_FFFF_FFFE).

The **gp=** parameter will be used by the linker when processing the patches. For example, one patch might indicate that a segment contains the following synthetic instruction:

```
loadb     r5,MySymbol
```

where "MySymbol" was imported. Since the value of the symbol will not be known until link time, the linker will be tasked with translating this synthetic instruction into one or more machine instructions.

Assuming that register "gp" contains a value such as 0x8_0000_8000 (which is typical for user programs) and that "mySymbol" has a value such as 0x8_0000_8123, the linker can replace the synthetic instruction with this machine instruction:

```
load.b    r5,0x123(gp)
```

However, if register gp happens to be undefined or has some other value, the linker will be required to use a different instruction.

After the symbol information in the object file, the segment data will actually appear.

The segments will be given in order. In other words, the data for segment #1 will come first, followed by the data for segment #2, and so on.

For zero-filled segments, there will be not be an entry with zero bytes; the entry will simply be missing.

The segment length is not constrained to be a multiple of anything and may be zero.

Symbols in the Object File

A single executable program may originate from several source files. Each source file will be assembled into an object file. These object files will then be combined in the linking phase to produce a single executable file.

The program is composed of several object files and each object file corresponds to a single source file. The linking process then combines the object files to produce the executable file.

Code in one object file may refer to addresses, instructions, data, and values defined in other object files. As an example, object file A may define a function called "printf" and object file B may call this function. When object file B is assembled, there is no information about where the "printf" function will be located or even what object file it will be in. As the linker processes all object files, it will modify the "call" instruction to fill in the final address of the "printf" function, in a process we call "patching". (Traditionally, this has also been called "relocation".)

Symbols are used to share such things as the address of the "printf" function across object file boundaries. Object file A would export the symbol "printf" and object file B would import "printf".

Ultimately each symbol must be assigned a value which will be a 64 bit signed integer. The linker will determine that value and will issue an error if it cannot determine the value of some symbol.

The value of each symbol will be either (1) the address of a location in some segment or (2) an absolute value, which is not the address of any location.

Since addresses are not determined until link time, any symbol which originates as a label (or an **.equ** to a label) will not have an actual value until link-time. Some symbols may have an absolute value known in the object file where it is defined, but this value will be unknown in any object file that imports the symbol.

The Symbol List

The next section of the object file consists of a number of symbols. For example, an object file may contain 100 symbols. Each of the 100 symbols is represented in the object file with a "**symbol entry**", which will have information such as "symbol number", "type", "relative to", and the characters of the symbol's name.

The symbols within each object file are sequentially numbered, starting with 1. These numbers are local to only that object file. The numbers are used in:

- The definition of other symbols
    A symbol can be given a value of "OtherSymbol + offset"
- The patch entries
    A synthetic instruction may use a symbol as its argument.

Each symbol has a name, which is a character string, and the symbol entry contains the string. The symbol name is used to match an exported symbol from one file with an imported symbol in another file. For this matching, symbol names are case-sensitive and must match exactly. The string is specified using length in bytes. No terminating character (\0 or \n) is used. ASCII encoding is used; only ASCII characters are allowed in symbols.

Each symbol in an object file has a **type code**, which indicates how that symbol was defined:

    1 = imported
    2 = label
    3 = equate

If a symbol is "**imported**", then the object file contained no definitions that symbol. Instead, the symbol is assumed to be exported by some other file. The linker must locate the definition (by matching the characters in the symbol name) and must tie the uses of the symbol in this object file to the definition in the other object file.

For a symbol of type "**label**", the symbol was defined by labeling an address in this object file. The definition consists of the segment that contained the label and the offset into that segment of the byte location that had the label.

Note that the linker will translate synthetic instructions into machine instructions. When translating a synthetic instruction which requires more than one machine instruction, the linker may be required to insert additional bytes into the middle of some segment. Whenever the linker inserts such additional bytes, it will update and shift the definition of all labels in that segment following the insertion.

The other way in which a symbol can be defined is with a **.equ** equate instruction. A symbol can either be equated to an absolute value that was known to the assembler or to some other symbol.

For any symbol given an absolute value (which will be a 64 bit signed integer), the symbol entry in the object file will be marked "**equate**" and will use the special value

of zero for the "**relativeTo**" field and the "**offset**" field will contain the actual value. (Perhaps an absolute value should be thought of as an offset from zero.)

For any symbol that is given a value in a **.equ** equate instruction where the value is not an absolute value known to the assembler, the definition will be of the form:

> *Symbol*:   .equ   *OtherSymbol + IntegerOffset*

The definition may not have that exact form, but it will be reduced to that. For example:

```
Sym43:     .equ    (-0x123 <<4 ) + Lab_98
```

The *OtherSymbol* may be defined in the current object file or may be an imported symbol. If it is defined in this file, then it will be a "label" type symbol. (If *OtherSymbol* had been an absolute value, then the assembler would have evaluated the expression, determined the value, and made this symbol an absolute value, not a "relative to" symbol.)

Regardless of how the symbol was defined (either as a label or in an equate), the symbol may or may not have been exported. Another field in the symbol entry will indicate whether or not the symbol is exported.

If a symbol is exported, then the linker will link it with any identically spelled symbol that is imported in another object file.

All exported symbols from all object files must be unique. It is an error for the same symbol to be exported from more that one object file. The linker will catch and report this error.

Symbols that are not exported are considered to be "**local**" to a single object file. Different object files may use identically spelled symbol names for different purposes; such symbols are completely unrelated and will have totally different definitions.

The purpose for including local symbols in the object file is that they can provide useful information to a debugger. Local symbols will be included in the executable file, but only for the purpose of debugging. They will not impact execution in any way.

For example, it is common for the programmer to create many local labels as targets for BRANCH, JUMP, and CALL instructions. It is very helpful when disassembling instructions in a debugger to be able to show the local labels to help the programmer get oriented and make sense of the disassembled instructions. As another example, unusual constants may be equated to local symbol names; displaying these symbolic names during debugging may make dissembled code easier to interpret.

The symbols are given in numerical order in the object file. After the last symbol, the list will be followed with a zero and a "********" separator. These will signal the end of the list.

Patch Entries

Next in the file will be a list of patch entries, each describing a patch that must be made by the linker.

Each patch entry will begin with a "type" code. The list of patch entries will be followed by a zero and a "********" separator. These will signal the end of the list.

Each path entry has this form (repeated from above):

<div style="border:1px solid black; padding:1em">

The following fields are repeated once for every patch...

1	The patch type (1, 2, ...)
4	Source file line number
4	The segment where the patch must be made
8	The location to be patched (i.e., offset into the segment)
4	The target symbol (0 = absolute)
8	Offset from target symbol (often zero)
	For patch type = "align", offset will be 8, 16, 32, or 16384
1	Exact size of result in bytes (4, 8, 12, 16) or -1 if don't care
	Only for Format S1,S2,...S7.

</div>

The "patch type" tells which synthetic instruction appeared in the source file, so the linker can know what instructions to generate.

The **.byte**, **.halfword**, **.word**, and **.doubleword** pseudo-ops can have as an operand an expression which has a value that cannot be determined until link time. There are 4 patch types, one for each of these.

The **.align** pseudo-op may also require linker attention and there is a special patch type for it, as well.

The "source line number" is used in error messages printed by the linker, but not otherwise used, with one interesting exception. Consider the following assembly code:

```
label:
        .align   16
```

The assembler will insert zero bytes for the **.align** pseudo-op, leaving the task to the linker. Thus, the label and the align will both be located at the same offset in the segment. As far as the object file goes, this code is indistinguishable from the following:

```
        .align   16
label:
```

But what happens if the linker is required to insert several bytes for the **.align** pseudo-op? These cases must be handled differently! In the first case, the label must be associated with the first padding byte; in the second case, the label must be associated with the first byte after the padding.

The line number on the symbol and the line number on the patch are used by the linker to distinguish these cases.

The "segment number" and "location to be patched" give the location that must be modified. The linker is required to change and/or insert bytes at that location.

For synthetic instructions that must be patched, there is always an "address" or "value" that could not be determined by the assembler. There are two cases:

An absolute number
> The "target symbol" will be zero and "offset" will contain the value

A symbolic address
> The "target symbol" will indicate which symbol was used.
> There may be an optional "offset" from the target symbol.

Once the linker determines the final address of the target symbol, the linker will add in the "offset", which is often zero. Then the linker can determine exactly which machine instructions are required and can modify the segment accordingly.

For synthetic instructions, the linker will be replacing the synthetic instruction by 1, 2, 3, or 4 machine instructions. Generally speaking, the assembler will either be unable to determine what the linker will do or will not care. In such cases, the "exact size" field will be -1 (i.e. "don't care").

However, in some cases, the size of the translation was important during assembly. The "exact size" field gives information about how many bytes the assembler has concluded will be needed for the translation.

Even though the assembler may have been able to determine that some instruction could be translated by a given number of bytes, it may have been unable to perform the actual translation. This might have occurred because the assembler was unable to know exactly what the linker would do for some other instructions somewhere else. However, the assembler may have depended on the translation for the instruction being some exact size. This size expectation is captured in the "exact size" field. The linker must ensure that its translation is the size expected by the assembler, but this will never be a problem since the assembler will only make such assumptions when it knows the linker can meet its size expectations.

For the BYTE, HALFWORD, WORD, and DOUBLEWORD patches, the exact size field will be 1, 2, 4, and 8, respectively.

In the case of ALIGN patches, all fields are present:

> target symbol — ignored (will be 0)
> offset — will be 8, 16, 32, or 16384
> exact size — ignored (will be -1)

How many bytes will be present in the file?

For synthetic instructions (i.e., S-1 through S-7), if the exactSize field is 4, 8, 12, or 16, then the object file will contain exactly that number of bytes. It the exactSize field is -1 (don't care), then the file will contain exactly 4 bytes.

If registers were used in the synthetic instruction, the first 4 bytes will contain the register identities in their proper places. To be more precise, when expressed in hex,

the first word will have the following format, where 3, 2, 1, and D symbolically represent the bit fields for encoding registers Reg3, Reg2, Reg1, and RegD, respectively.

```
0000321D
```

Since each register is encoded with 4 bits, the first word will have this format, expressed in binary:

```
0000 0000   0000 0000   3333 2222   1111 DDDD
```

For the BYTE, HALFWORD, WORD, and DOUBLEWORD patches, the object file will contain 1, 2, 4, and 8 bytes, respectively.

For all ALIGN patches, the object file will contain 0 bytes.

The Patch Types

There are 25 patch types, numbered 1 … 25:

Format S-1:
> **Patch Type 1**
>> MOVI (regD ≠ gp)
> **Patch Type 2**
>> MOVI (regD = gp)

Format S-2:
> **Patch Type 3**
>> BEQ   Reg1,Reg2,Address
> **Patch Type 4**
>> BNE   Reg1,Reg2,Address
> **Patch Type 5**
>> BLT   Reg1,Reg2,Address
> **Patch Type 6**
>> BLE   Reg1,Reg2,Address

Format S-3:

> **Patch Type 7**
> > JUMP   Address
> **Patch Type 8**
> > CALL   Address

Format S-4:

> **Patch Type 9**
> > LOADB   Regd,Address
> **Patch Type 10**
> > LOADH   Regd,Address
> **Patch Type 11**
> > LOADW   Regd,Address
> **Patch Type 12**
> > LOADD   Regd,Address

Format S-5:

> **Patch Type 13**
> > LOADB   Regd,Offset(Reg1)
> **Patch Type 14**
> > LOADH   Regd,Offset(Reg1)
> **Patch Type 15**
> > LOADW   Regd,Offset(Reg1)
> **Patch Type 16**
> > LOADD   Regd,Offset(Reg1)

Format S-6:

> **Patch Type 17**
> > STOREB   Address,Reg2
> **Patch Type 18**
> > STOREH   Address,Reg2
> **Patch Type 19**
> > STOREW   Address,Reg2
> **Patch Type 20**
> > STORED   Address,Reg2

Format S-7:

Patch Type 21
STOREB   Offset(Reg1),Reg2
Patch Type 22
STOREH   Offset(Reg1),Reg2
Patch Type 23
STOREW   Offset(Reg1),Reg2
Patch Type 24
STORED   Offset(Reg1),Reg2

Align:

Patch Type 25
"offset" contains the alignment requirement

Data:

Patch Type 26
BYTE — The linker will print an error if the value will not fit
Patch Type 27
HALFWORD — The linker will print an error if the value will not fit
Patch Type 28
WORD — The linker will print an error if the value will not fit
Patch Type 29
DOUBLEWORD

Order of patches within an object file:

Each patch applies to a location within a segment. The patches in an object file must be in proper order, as discussed next:

The patches for all segments must appear together. The segments must appear in order. For example, all patches for segment #1 must be placed before the patches for segment #2.

Furthermore, all patches for a given segment must be in order by the location to be patched. For example, a patch to offset 0x40 in segment #2, must come before a patch to offset 0x44 in that segment.

The following patch types are not used:

All instructions of format B, C, and D instructions require an immediate value as an operand. The assembler requires such an immediate value to be knowable by the assembler from the information in the assembly source file. Thus, for these instructions, the assembler will produce the final machine code and the linker will never need to modify them.

Therefore, the following patch types are not required, not used, and not implemented.

Format B:
> **Patch Type *XXX***
>> immed-16 — Errors would occur if the value will not fit

Format C:
> **Patch Type *XXX***
>> immed-16 — Errors would occur if the value will not fit

Format D:
> **Patch Type *XXX***
>> immed-20 — Errors would occur if the value will not fit
>> Normally UPPER20/AUIPC/JAL/ADDPC
>>> are the result of synthetic instruction translation.

Debugging Information - Header Info

The segments, the symbols, and the patches are not optional. Every **.o** object file will contain that information.

But after these, the file might or might not contain additional information to support the runtime debugger, which we describe next.

The debugging information is optional. If the **-nodebug** option was specified on the assembler command line, then no debugging information will be added to the object file. Also, if the file contained no debugging pseudo-ops, then no debugging information will added to the object file. Otherwise, the information will be included at the end of the file.

The debugging information begins with a header block of data. If debugging information is not included in the file, the the header information will contain the following values to signal this and the object file will include nothing further.

Field	Value
Package name, number of bytes	0
Package name string	*< no bytes >*
The second string, number of bytes	0
The second string	*< no bytes >*
The number of globals	0
The number of functions	0
Separator ("********")	0x2A2A_2A2A_2A2A_2A2A

Otherwise, the package name and the second string (which come from the **.sourcefile** pseudo-op will be present. These are null-terminated UTF-8 strings, and their sizes (in bytes, including the **\0**) are also given.

Since both strings will contain at least the **\0** character, their lengths will be greater than 0. The empty header can be differentiated by the first field, i.e., the number of bytes in the package name string.

Following the debugging header block there will be a number of globals and a number of functions.

Debugging Information - Global Blocks

For each appearance of a **.global** pseudo-op in the source file, there will be a single block of information. Each block will include these fields:

bytes	field description
4	Global name: number of bytes (M); will be > 0
M	Global name: the UTF-8 encoded string (with terminating \0)
4	Source file line number
1	Type Code (One character code, e.g. 'I')
4	Location: The segment number
8	Location: Offset into segment

The name of the variable (as given in the **.global** pseudo-op) is a null-terminated UTF-8 string; its size (in bytes, including the **\0**) is also given. This is followed by the

source code line number, as given in the **.global** following "**line=**". (This is presumably the line number from a KPL source code file and not the line number in the **.s** file.).

The type of the variable is indicated by a single character. This simple typing scheme doesn't match the richness of KPL's type system, but is enough to support debugging at the machine code level.

The **.global** pseudo-op should be placed in the **.s** file directly before the variable to which it applies. For example:

```
        .global         "myVar", line = 60, type = "I"
  P_MyPack_myVar_43:
        .doubleword     0
```

The name in the **.global** is what the KPL programmer used; the label in the **.s** file is the (presumably mangled) name generated by the compiler.

The address of the next thing following the **.global** will be associated with this debugging information. The "location" fields in the global data block give the segment and offset at which that thing will be located. During the linking step, the linker will place the segments in memory, and will determine the actual address at that time. The debugger will use this information to know that an integer (a signed 64-bit value) with name "myVar" (defined on source code line 60) is stored at the address.

The assembler and linker do not check whether the **.global** is placed before the correct instruction and do not check whether the type code is correct. For example, the assembler and linker will accept the following with no complaint. Obviously, the location will contain a couple of machine code instructions, not a pointer (P). This will trick the debugger, which will display "myVar" as a pointer, interpreting the machine code bits for these instructions as an address. (Since this would confuse anyone using the debugger, the compiler will only place a **.global** pseudo-op directly before the data bytes to which it applies.)

```
        .global         "myVar", line = 60, type = "P"
        add             r1,r2,r3
        xori            r3,r4,567
```

Type Codes Used for Debugging

Here are the single character codes used. The type code will be a single character. Compound types (such as "PI" for "ptr to int") are not supported.

I	**int**	64-bit signed integer
W	**word**	32 bit signed integer
H	**halfword**	16 bit signed integer
C	**byte** (C = Char)	8 bit signed integer or ASCII char
L	**bool** (L = Logical)	TRUE / FALSE, 8 bits
D	**double**	64 bit double-precision floating point
S	**String**	Array of bytes; UTF-8 encoded
P	**ptr**	Pointer, 64 bits
A	**array**	
O	**object**	
R	**struct** (R = Record)	
U	**union**	
?	**other / unknown /missing**	

The same type code characters are used in **.global**, **.regparm**, and **.local** pseudo-ops.

Debugging Information - Function Blocks

For each appearance of a **.function** pseudo-op in the source file, there will be a single block of information.

The **.function** pseudo-op is used to give the debugger information about a function or a method. The debugger treats methods and functions the same way. The receiver (i.e., "self") is always a pointer to an object and is always the first parameter, so it will be in register **r1**. Otherwise, the code for methods and functions is identical.

Each function block will include these fields:

bytes   **field description**
- 4   Function name: number of bytes (M); will be > 0
- M   Function name: the UTF-8 encoded string (with terminating \0)
- 4   Source file line number
- 4   Location: The segment number
- 8   Starting Location: Offset into segment
- 8   Beyond Location: Offset into segment (i.e., address of last byte + 1)
- 4   Frame size (not negative; 0 = leaf function)

Following each function block, will be zero or more "register parameter blocks" with one for each **.regparm** pseudo-op appearing after the **.function**.

Following the register parameter blocks, will be zero or more "local variable blocks" with one for each **.local** pseudo-op appearing after the **.function**.

Following the local variable blocks, will be zero or more "statement blocks" with one for each **.stmt / .comment** pseudo-op appearing after the **.function**.

The name of the function or method (as given in the **.function** pseudo-op) is a null-terminated UTF-8 string; its size (in bytes, including the **\0**) is also given. This is followed by the source code line number, as given in the **.function** pseudo-op following "**line=**". (This will be the line number from a KPL source code file and not the line number in the **.s** file.).

The function block tells where the function's code begins and where it ends, as determined by the placement of the **.function** and **.endfunction** pseudo-ops. These are given by offsets into a segment and these locations will be turned into addresses by the linker. The ending address is given as the location just past the end of the function (i.e., the location of the next thing following the function).

The framesize field gives the size of the stack frame in bytes and will always be a positive multiple of 8. A zero value indicates that this block describes a leaf function. The debugger needs this information in order to go down into the stack to retrieve information from buried stack frames.

Note that the framesize is the amount that register **sp** is adjusted whenever this function is invoked. Leaf functions will often use elements above the stack top (i.e., with lower addresses), but they must not adjust register **sp**, or else the debugger will become very confused. (This is because the debugger must be able to locate the

return address field in buried frames. From the return address, the debugger can deduce which function was executing and, from that, the debugger can deduce the frame sizes of buried frames, which it needs to know in order to work its way further down the stack.)

Debugging Information - Register Parameter Blocks

For each appearance of a **.regparm** pseudo-op following a **.function**, there will be a single block of "register parameter" information. Each block will include these fields:

bytes	field description
4	Source file line number (>= 0)
1	Register number (1 … 15)
4	Parameter name: number of bytes (M); will be > 0
M	Parameter name: the UTF-8 encoded string (with terminating \0)
1	Type Code (One character code, e.g. 'I')

The name of the parameter is a null-terminated UTF-8 string which comes from the **.regparm** pseudo-op. Likewise, the source code line number comes from "line=" the **.regparm** pseudo-op.

The line number will never be negative. The line number field is listed first and a value of -1 is used to terminate the list of register parameter blocks.

The register number will normally be 1 … 7 since the standard calling conventions use only registers **r1** … **r7** for passing parameters. The debugger may or may not be able to cope with values 8 … 15.

The type code character meanings were listed above.

All the register parameter blocks for a given function will occur in the object file directly after the function block and before the local variable blocks and statement blocks, regardless of their order in the **.s** file.

Debugging Information - Local Variable Blocks

For each appearance of a **.local** pseudo-op following a **.function**, there will be a single block of "local variable" information. Each block will include these fields:

bytes	field description
4	Source file line number (>= 0)
4	Offset from stack top
4	Variable name: number of bytes (M); will be > 0
M	Variable name: the UTF-8 encoded string (with terminating \0)
1	Type Code (One character code, e.g. 'I')

The name of the local variable is a null-terminated UTF-8 string which comes from the **.local** pseudo-op. Likewise, the source code line number comes from "line=" on the **.local** pseudo-op.

The line number will never be negative. The line number field is listed first and a value of -1 is used to terminate the list of local variable blocks.

The offset tells where in the stack frame the parameter or local variable will be located. The offset is in bytes, relative to the stock top. A more positive offset is buried deeper in the stack.

For parameters passed on the stack, the data in memory will be valid at the time the function is called. In other words, the calling conventions require that the argument be placed in the stack at the given offset before the CALL instruction is executed.

However, during the execution of any function (or method), parameters and local variables will often be cached in registers. Even though the compiler has included a **.local** pseudo-op to describe a parameter or a local variable, it is likely that the value will be cached in a register for much of the execution of the function, and the debugger will not know about this. Be aware of this.

The type code character meanings were listed above.

All the local variable blocks for a given function will occur in the object file directly after the register parameter blocks and before the statement blocks, regardless of their order in the **.s** file.

Debugging Information - Statement Blocks

For each appearance of a **.stmt** pseudo-op following a **.function**, there will be a single block of "statement" information. The **.comment** pseudo-op is handled as a special case of the statement block. For each **.comment** there will be a single statement block as described here. In other words, the following block of information describes either a **.stmt** or **.comment** pseudo-op, as differentiated by the typecode field.

Each statement block will include these fields:

bytes	field description
4	Source file line number (>= 0)
4	Location: Segment number
4	Location: Offset into segment
1	Type Code (0=comment, 1=assign, …)

If and only if type code = 0/comment, the following will be present…

4	Comment String: number of bytes (M); will be > 0
M	Comment String: the UTF-8 encoded string (with terminating \0)

The source code line number comes from "line=" on the **.stmt**. There is no associated line number for a **.comment** so this field will be 0 for **.comment** pseudo-ops. The line number will never be negative. The line number field is listed first and a value of -1 is used to terminate the list of statement blocks.

The **.stmt** or **.comment** applies to the thing that follows it. The location given here will be translated by the linker into an address.

Furthermore, the statement blocks for a given function are guaranteed to be in order by location. They will be in the same order they occurred in the **.s** file.

The type code are integer codes. For example:

```
0   comment         < from .comment pseudo-op >
1   assign          ASSIGNMENT statement
2   if              IF statement
3   then            THEN statements
4   else            ELSE statements
5   call            CALL statement
6   send            SEND statement
7   while_expr      WHILE LOOP (expr evaluation)
8   while_body      WHILE LOOP (body statements)
9   do_body         DO UNTIL (body statements)
    ... etc...
```

All the statement blocks for a given function will occur in the object file directly after the local variable blocks.

If, and only if, the type code is 0, then this statement block of data describes a **.comment** pseudo-op. For such a block, there will also be a string, which gives the comment information. The comment is a null-terminated UTF-8 string. The **.comment** does not have a "line=" field; the value of the source code line number will be 0.

Future Work

The fields of the **.o** object file are not properly aligned. This creates a potential performance problem for the file I/O performed by the **asm**, **link**, and **createlib** tools.

Typically, files are implemented with memory-mapped I/O. File READ and WRITE operations end up becoming nothing more than memory-to-memory data movement. Thus, proper alignment may speed up file operations, at the cost of increasing file size.

Perhaps the file format needs to be redefined so that all fields are properly aligned. This would require changes to the **asm**, **link**, **createlib**, and **dumpobj** tools.

Segment sizes are not constrained to be a multiple of any number. We ought to add a padding field (with 0 ... 7 padding bytes) to follow the data for each segment, in order to make sure that all subsequent segment data chunks are doubleword aligned.

The changes described here can be expected to have only a modest impact on the performance of the **asm**, **link**, and **createlib** tools. We think "modest" because these tools don't spend much time performing I/O. The bulk of processing for these tools is spent manipulating in-memory data structures.

On the other hand, the **.o** object file format has been designed to minimize file size, which also contributes to performance. Without empirical testing, it is not certain that performance would be significantly improved by redesigning the object file format.

Therefore, these proposed changes will not be pursued.

Chapter 10: Executable File Format

Quick Summary

- The linker produces an executable file.
- The executable file is loaded by the OS kernel at runtime.
- The format of the executable file is given, including:
    — Version and machine architecture identification
    — The number of pages, and addresses of the pages
        For each page, its "writable" and "executable" attributes are given.
    — A number of segments
        For each segment, the address, length, and data bytes are given.
    — The entry point, an address at which to begin execution
- The executable file also contains a "debugger info section".
    — The debugger info is used in reporting runtime error messages.
    — The debugger info will be used by the debugger tool.

Introduction

The linker tool takes one or more object files and combines them, producing an executable file. The executable file contains all the information needed by the OS kernel to load the program into memory and begin execution.

In this chapter we give the format of the executable file.

In Unix/Linux, executable files are sometimes called "a.out" files.

File Format

An executable file has two sections called the "**executable section**" and the "**debugger info section**". The first part of the file contains all information needed to load a virtual address space and commence execution. The second part of the file contains information that will only be needed if errors arise during execution or if a debugging tool is used.

Every file always has both sections. The debugger info section always follows the executable section. There are no bytes outside of the sections. In other words, the length of the file is simply the length of the executable section plus the length of the debugger info section. The sections are concatenated to create the complete file.

The debugger info section can be safely ignored for now. The debugger info section is discussed later in this chapter, after the description of the execution section.

The file can be considered as series of fields. The length of each field is given in the left-hand column.

The executable section of the file has the following format.

bytes   **field description**

The following fields constitute the header information...

bytes	field description
8	Magic number "B64a.out" (in hex: 0x4236_3461_2E6F_7574)
8	Version Number (0x0000_0000_0000_0001)
2	Blitz-64 ISA Architecture (e.g., 0x0002)
2	Padding (0x0000)
4	Number of pages (0 if this is a kernel program)
8	Lowest used address
8	Highest used address
8	Entry Point
4	Number of modules
4	Number of symbols
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every region...

    8   Starting Address (0x0 … 0xF_FFFF_C000)

    4   Number of pages

    1   Is Executable? (1=pages should be marked "executable")

    1   Is Writable? (1=pages should be marked "writable")

    2   Padding (0x0000)

    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After all regions...

    8   -1 to terminate (in hex: 0xFFFF_FFFF_FFFF_FFFF)

The following fields are repeated once for every segment...

    8   Starting Address (0x0 … 0xF_FFFF_FFF8). Will be a multiple of 8.

    8   Length in bytes (N). Will be a multiple of 8.

    4   Number of module from which this came

    4   Source code line number

    7   Padding (0x00_0000_0000_0000)

    1   Is zero-filled? (1=zerofilled; 0=data bytes are present)

    N   The bytes to load into memory. (Only if IsZerofilled=0)

    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After all segments...

    8   -1 to terminate (in hex: 0xFFFF_FFFF_FFFF_FFFF)

    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every module...

    4   Number of module (1, 2, 3, …)

    4   Name of .s source file: number of characters (L)

    L   Name of .s source file: the ASCII characters (no terminating \0)

After all modules...

    8   -1 to terminate (in hex: 0xFFFF_FFFF_FFFF_FFFF)

    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every symbol...

4	Number of module which defined this symbol (1, 2, 3, ...)
4	Source file line number
8	Value of this symbol
1	Is Label?

       0 = this value is probably not an address
       1 = this symbol derives from a label definition

4	Symbol name: number of characters (M)
M	Symbol name: the ASCII characters (no terminating \0)

After all symbols...

4	-1 to terminate (in hex: 0xFFFF_FFFF_FFFF_FFFF)
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After the Executable Section...
    (The Debugger Info Section, which is discussed later)

Magic Number

Every executable file begins with a special doubleword value. This value of this "magic number" can be interpreted as the ASCII encoding of the characters "B64a.out".

Since all valid executable files begin with this value and since this particular value is highly unlikely to occur in other files, this is a fairly good way to catch accidental user errors. For example, any attempt to execute a ".o" object file or to give an executable file as input to the linker tool will be caught by the magic number check.

The Version Number and ISA Architecture Fields

Following the magic number is a "version number". We understand that future changes may be required to the format of executable files. This field exists to accommodate changes, updates, and extensions to this file format.

This document describes "version 1" of the file format. All files conforming to this specification will have the value 1 in this field. Any other value indicates that the remainder of the file will conform to a different specification.

At this time, there is only one version of this file format and the linker is only capable of producing "version 1" files. Future versions of the linker tool may be capable of producing different versions.

Future versions of the Blitz kernel may or may not be able to load and execute files in the "version 1" format or other versions. Details about future compatibility must be documented in the future, obviously.

The "ISA Architecture" field specifies which type of machine this code is intended to be run on. This value must match the value from the version number in bits [30:16] of the CSR register **csr_version**. In other words, the numbers used in this field and the in **csr_version** are drawn from the same set and therefore have the same values and meanings.

At this time, the current version Blitz-64 Instruction Set Architecture (ISA) is

0x0002

In the future, changes and/or additions to the machine code instructions are likely. For example, we plan to specify and implement the compressed instruction set in the future. When changes are made to the ISA , the **csr_version** will be changed (incremented) to reflect a modified architecture.

Commentary We separate out the "file version number" and the "ISA architecture version" into two fields because these really track two different kinds of changes that can be made in the future. A change to the machine architecture may not require a change to the file format. Conversely, a change to the file format may be implemented even though there is no change to the ISA.

Commentary A "Fat Executable" file contains multiple copies of the executable code, each assembled for a different architecture. This effectively combines several executable modules into a single file. The benefit of doing this is that a single executable file can be run on different machines and is therefore, to this extent, portable. At this time, we avoid fat executables, but if this added in the future, the file format will need to be modified to accommodate multiple architectures. At that time,

the file format "version number" will be increased to reflect the changes to the file format.

Padding Bytes

Executable files contain a large amount of data that must be moved from the file into memory. It is crucial that loading an executable into memory be made as fast as possible, since load-time is consumed whenever a program is executed.

In order to speed up this copying, it is important for the data to be properly aligned.

To make sure subsequent fields are doubleword aligned, there are "padding bytes" inserted into the executable file in a couple of places. These byte should be zeros.

Number of Pages

Every executable is either a "kernel program" or a "user program". Kernel programs will be loaded into kernel memory (i.e., addresses within 0x0 … 0x3_FFFF_FFFF). User programs will be loaded into the virtual memory region (i.e., addresses within 0x8_0000_0000 … 0xF_FFFF_FFFF).

If this file contains a kernel program, the "number of pages" field will be 0 and there will be no regions. Otherwise, this field will indicate the number of memory pages that are required to run this program.

Typically, the OS kernel will allocate the required number of pages all at once, and then fill them in subsequently. (This is because allocating the pages piecemeal may result in a deadlock. Consider the situation in which some processes have grabbed some of the pages they need but are waiting to get additional pages.)

In order to know how many pages are required (so they can all be obtained before any are needed), this field tells how many will be required.

Lowest and Highest Used Addresses

These values give the full range of addresses that will be used by the program.

For user programs, the lowest address will always be a page-aligned address and the highest address will a page-aligned address, minus 1.

For kernel programs, the lowest and highest addresses used will not necessarily be page-aligned. These values are important for loading a kernel. The kernel program will be loaded by some form of "boot loader". Both the boot loader and the kernel will reside in the kernel address space. This check is important so that the boot loader can make sure that the material it is loading will not overwrite the boot loader itself.

Entry Point

Every program must define a value for and export the symbol "_entry". This value should be a legal address. Once loaded into memory by the kernel at runtime, execution will begin at this location. In the case of the kernel program, the boot loader program will end by jumping to this address.

If the linker is compiling a kernel program (i.e., if the -k command line option is present), the linker will ensure that the value is within the kernel address space, i.e., 0x0 ... 0x3_FFFF_FFFF. Otherwise, the linker will ensure the address is within the user address space, i.e., 0x8_0000_0000 ... 0xF_FFFF_FFFF.

The linker will not ensure that the address is within an allocated page or segment. If this entry address is not an allocated address, the program will presumably signal an unrecoverable page fault or addressing error immediately upon execution at runtime, if it is a user program. If it is the kernel program, an illegal instruction exception will probably be signaled.

Separators

The separators, which were discussed earlier, serve as a check to make sure the file format is followed.

List of Regions

There will be zero or more regions listed after the header information.

Kernel programs will have no regions; user programs will have one or more regions.

For each region, several fields will appear. The first field is "starting address", which will never be negative. Following the regions, a value of 0xFFFF_FFFF_FFFF_FFFF (i.e., -1) will appear. The -1 value will mark the end of the list of regions.

A region is a set of one or more pages, all of which are contiguous, i.e., placed sequentially in memory, one after the other, with no intervening gaps. The field called "number of pages" tells how large the region will be. You can multiply the number of pages by the page size to determine the size of the region in bytes.

A page can be…

- Either writable or read-only, and
- Either executable or not executable

The next two fields "Is Executable?" and "Is Writable?" tell how the pages should be marked before execution begins. All the pages in a region will have the same protection attributes. That is, all pages in the region are to be marked identically.

The collection of pages in the region list describes how the kernel should set up the virtual address space before the program begins. (The kernel will also add additional pages, e.g., for stack and environment variables).

Each region is ends with a separator.

List of Segments

After the header list there will be a list of segments. Each segment is described by a block of fields that begins with "starting address" and ends with a separator.

The starting address will never be negative. After the list of segment blocks, there will be a field with value -1 (i.e., 0xFFFF_FFFF_FFFF_FFFF). This -1 value will occur in place of the starting address of the next segment and is used to determine when the list of segment blocks ends.

A segment (as discussed in the context of the executable file format) gives the actual data bytes that must be loaded into memory. Each segment contains a starting address, a length in bytes, and a block of data. The "starting address" tells where to place the data and the "length in bytes" tells how big the block of data is.

Some segments are "zero-filled", which means that they contain nothing but zeros. To avoid storing long strings of zeros in the file, the segment is marked "zero-filled" and the block of data is not given. The field "Is Zerofilled?" is used to determine whether (A) a data block is present and must be moved into memory, or (2) no data block is present and the memory is to be zero-ed instead.

Every assembly language segment starts with a **.begin** pseudo-op and there is a one-to-one correspondence between **.begin** pseudo-ops and segments.

Every segment in an assembly language source file will result in one segment being placed into the executable file.

Caveat Segments in the assembly language file can actually have zero length. This is not an error, although a segment of zero length is meaningless and the product of sloppy programming. For a segment of zero size, nothing will be placed in the executable file.

So, more precisely: For every assembly code segment of length greater than zero, there will be a segment in the executable file. Segments from the assembly file will never be broken apart and correspond to no more than one segment in the executable file.

Furthermore, the linker may also insert additional segments that do not correspond to any **.begin** in the assembly source file. These segments will always be zero-filled. These extra zero-filled zones result from and fill the gaps between the segments.

Every byte in every page in every region included in the executable file will be given a value exactly once. Most bytes in the executable will come from the code and/or data specified in the assembly source file. However, any byte not explicitly specified is required (by the Blitz-64 design spec) to be initialized to zero.

When the linker places the segments into pages, there may be gaps. These gaps can occur because the programmer specified "startaddr=" values and these resulted in gaps between segments or resulted in unused space at the beginning of the page. Gaps will also occur whenever segments fail to completely fill a page.

To ensure that these gaps are properly initialized, the linker creates additional zero-filled segments that describe the areas that must be initialized at load-time.

Commentary Copying bytes and initializing bytes may perhaps be done by special DMA hardware outside the processor core. However, it is reasonable in many systems to perform these operations directly by machine code executing in the core.

Let's compare the cost of copying bytes versus zero-filling bytes.

To copy bytes, a loop such as this may be required:

```
# r1 = destination ptr
# r2 = source ptr
# r3 = stop value
loop:
        load.d      r4,0(r2)
        store.d     0(r1),r4
        addi        r1,r1,8
        addi        r2,r2,8
        blt         r1,r3,loop
```

This example omits a lot of details, including loop setup and boundary conditions. It also assumes the addresses are properly aligned. Regardless, this seems to be the minimal loop needed for the bulk of a large copy operation.

On the other hand, the code to initialize a large block of memory might depend on a loop such as this:

```
# r1 = destination ptr
# r3 = stop value
loop:
        store.d     0(r1),r0
        addi        r1,r1,8
        blt         r1,r3,loop
```

The bottom line is that zero-filling a large region of memory will be substantially faster than copying bytes into that region. And this doesn't even consider the cost of storing and reading in data from the executable file.

So there is good reason to accommodate zero-filled segments.

Modules and Symbols

The executable file contains:

- A list of all modules
- A list of all symbols

This information is not necessary to load and execute the program. Typically, the OS kernel will ignore and skip this information when a program is read and loaded into memory prior to execution.

This information is provided for use in disassembling and debugging a program. After loading a program into memory, a debugging tool can go back to the executable file and retrieve this additional information for use in the debugging process.

A single module is included in the file for every **.o** object module that was included by the linker in the executable file. The only information included is the name of the original **.s** source file from which the module came. This module information is included so that each symbol can be associated with the name of the file in which it was defined.

The executable file also includes a number of symbols. The information for each symbol includes both a reference to the module number (so the source filename in which this symbol was defined can be retrieved), along with the line number in that file.

Every symbol that is used as a label is included in the executable file. Every symbol that is defined with a **.equ** pseudo-op is included.

Each symbol has an associated value. The symbols do not appear in the executable file in any particular order. The are neither in alphabetical order nor in numerical order by value.

Each symbol has a flag to indicate whether it is thought to be an address or not. This information is not precise. For example, consider this code:

```
loc:    .equ    0x800000000
        …
        movi    r1,loc
        loadb   r7,0(r1)
        …
        loadb   r7,loc
```

As you can see from the way it is used, the symbol "loc" is clearly an address. Yet "loc" will not be identified within the executable file as an address.

In the following example, the symbols "**var1**" and "**var2**" will be identified as addresses in the executable file:

```
var1:   .word   0
var2:   .equ    var1+2
        …
        loadb   r7,var1
```

When displaying out the contents of memory, a debugger tool is free to use the symbols information when displaying information. Although the following example is only suggestive, it shows how a debugger might display the contents of memory and the value of having information about symbolic addresses during debugging.

```
      800000b80:   a4
      800000b81:   02
 var1:
      800000b82:   7c
      800000b83:   15
 var2:
      800000b84:   8f
      800000b85:   44
      800000b86:   28
      800000b87:   05
```

The Debugger Info Section

The initial portion of the executable file contains the information necessary to load and execute the program. The format of the initial section of the file was described above. After the initial **executable section**, the file includes a second section of data which is only used for error reporting and runtime debugging.

The second section is called the **debugger info section**. During normal, error-free executions of the program, the debugger info will never be read from the file. However, when a runtime error occurs or whenever the programmer wants to use the debugger tool, the debugger section will be read from the file.

The format of the debugging section is designed to promote simple and fast loading into memory. This is important because when an error occurs at runtime, the goal is to display an error message quickly.

A typical error message might look like this:

NULL POINTER EXCEPTION: Assignment stmt in "myFunction" in package "MyPack", line N

One purpose of the debugger info is to supply the underlined information:

Function Name — The currently executing function
Package Name — The source package name associated with this function
Statement Type — The most relevant statement information
Line Number — The line number of the statement

The error reporting will have only the current address (i.e., the PC) at which the error occurred. From this value, it must be possible to quickly determine the above information.

The error reporting code may be a part of the failing program. That is, the error handling code may be linked and loaded with the failing program and the program may be responsible for printing the error message itself. Or, it may be that the OS kernel will produce the error message and the failing program will not execute any more instructions.

In any case, the error reporting code will need the identity of the executable file, from which the debugger info section can be read. How that file is obtained is not discussed further here.

Another purpose of the debugger info is for use by a debugger tool. In that scenario, the debugger tool will read in the debugger info from the executable file upon startup. The performance constraints are not as important in this scenario. The tool is free to read in the information and build complex internal data structures that it will use during the debugging session.

Layout of Debugging Information

Here is the format of the debugger section:

bytes   **field description**

The following fields constitute the header...
4	Number of modules (K); will be > 0
4	Number of Global blocks included below
4	Number of Function blocks included below
4	Number of Statement blocks included below

The following fields are repeated once for every Module...
4	Module number (1, 2, 3, ..., K)
	Same module numbers as in executable section.
	These will be in numerical order.
	Every module will be represented here, even if there is no info.
	No info means that both strings are "\0".
4	Package name: number of bytes (M; will be > 0)
M	Package name: the UTF-8 encoded characters (with terminating \0)
4	The second string: number of bytes (N; will be > 0)
N	The second string: the UTF-8 encoded characters (with terminating \0)
4	The **.o** object filename: number of bytes (P; will be > 0)
P	The **.o** object filename: the UTF-8 encoded characters (with term. \0)
4	The **.s** source filename: number of bytes (R; will be > 0)
R	The **.s** source filename: the UTF-8 encoded characters (with term. \0)
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After all Modules...
4	Zero to terminate (in hex: 0x00000000)
8	Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every Global...
4	Module number (will be > 0)
4	Source line number
1	Type Code (One character code, e.g. 'I')
8	Address in memory
	(The globals are not in any order)
4	Global name: number of bytes (M); will be > 0
M	Global name: the UTF-8 encoded characters (with terminating \0)

After all Global entries….
  4   -1 to terminate (in hex: 0xFFFFFFFF)
  8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every Function…
  4   Source line number
  4   Module number (will be > 0)
  4   Frame size (not negative; 0 = leaf function)
  8   Starting Address in memory
            (The functions are not in any order)
  8   Beyond Address in memory
  4   Function name: number of bytes (M); will be > 0
  M   Function name: the UTF-8 encoded characters (with terminating \0)

    The following fields are repeated once for every Register Parameter…
      4   Source line number (>= 0)
      1   Register number (1 … 15)
      1   Type Code (One character code, e.g. 'I')
      4   Parameter name: number of characters (M); will be > 0
      M   Parameter name: the UTF-8 encoded chars (with terminating \0)

    After all Register Parameters…
      4   -1 to terminate
      8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

    The following fields are repeated once for every Local Variable…
      4   Source line number (>= 0)
      4   Offset from stack top
      1   Type Code (One character code, e.g. 'I')
      4   Variable name: number of bytes (M); will be > 0
      M   Variable name: the UTF-8 encoded chars (with terminating \0)

    After all Local Variables…
      4   -1 to terminate
      8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every Statement…
  4 Source line number (>= 0)
  8 Address in memory
      (The statements are not in any order)
  1 Type Code (0=comment, 1=assign, …)
 If and only if type code = comment, the following will be present…
  4 Comment String: number of bytes (M); will be > 0
  M Comment String: the UTF-8 encoded chars (with terminating \0)

After all Statements…
  4 -1 to terminate
  8 Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

After all Function entries…
  4 -1 to terminate (in hex: 0xFFFFFFFF)
  8 Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

Chapter 11: Object Libraries

Quick Summary

- A "library object file" is a binary file.
- The library file has an extension of "**.lib**".
    - The library file is identified by its own magic number ("**B64o_lib**").
- The library file begins with an index.
- Each index entry contains:
  — File name of original "**.o**" file
  — Where in the library file the module begins (Length is unnecessary.)
  — A list of all symbols exported by that module
- The index is followed by object modules.
    - There will be one or more modules in the file.
    - Each module has the same format as the object file it came from.
- Object modules may import symbols.
    - There is no checking to make sure the imported symbols are defined.
- A library is created by the "**createlib**" tool.
    - The tool takes one or more object files as input.
    - The tool adds all object modules to the newly created library file.
- The "**createlib**" tool will issue an error if the same symbol is exported by more than one object module.
- The linker tool will issue an error if the same symbol is exported in two different libraries or conflicts with an input object file.
- The "**dumpobj**" tool can be used to display the contents of an object library, as well as an object file.

The Format of a Library File

The library file has the following format. The file can be considered as series of fields. The length of the fields is given in the left-hand column.

bytes   **description**

The following fields constitute the header information...
    8   Magic number "B64o_lib" (in hex: 0x4236_346F_5F6C_6962)
    8   Version Number (0x0000_0000_0000_0001)
    8   Number of object modules
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following fields are repeated once for every object module...
    4   Name of original **.o** file: number of characters (M)
    M   Name of original **.o** file: the ASCII characters (no terminating \0)
    8   Offset into file of this object module

    Repeated once for every exported symbol in this module...
      4   Symbol name: number of characters (N)
      N   Symbol name: the ASCII characters (no terminating \0)

    To terminate the list of symbols...
    4   Zero
    8   Separator "********" (in hex: 0x2A2A_2A2A_2A2A_2A2A)

The following data blocks are repeated once for every object module...

    X   The object file contents (X bytes = size of original **.o** file)

Introduction and Motivation for Libraries

A typical program will use functions that have been written previously by someone else. Typically there exists a large collection of functions that are intended for re-use by many different, unrelated programs.

For example, consider a collection of math-related functions, such as:

sin, cos, sqrt, log, …

These functions are written once and used in many different programs. There may be hundreds of such math functions although a particular program will use, at most, only a few of them.

When creating an executable file, we need a way to include only the functions that are needed, without including code that is not needed. The simple solution of including all math functions in every program is unacceptable. This would lead to very large executable files and constitute a waste of memory.

A library is a single file containing the entire collection of all the functions. In this example, the "math library" is a single file containing all the math functions that are available for use. Since there are many math functions, this file may be quite large.

First, let's consider a program which does not use a library.

After writing and assembling a program, a ".o" object file will be created. In fact, a large program may have several assembly source files, and several ".o" object files may be needed. The linker will combine all the object files and produce the executable file. The linker will include all the code and data bytes in all the object files, regardless of whether it is necessary or not.

Next, we discuss how a library file is used.

When linking the program, the linker tool may also consult a library file. Typically, the input to the linker consists of a list of object files, as well as a library file to consult.

If the program makes use of some math function — say "cos" for example — the linker will include the code for that function in the executable. If a function is not used — for example, the "sqrt" function — the code for that function will not be included.

The linker understands the format of the library file and will extract from it only the functions that are needed.

So far, we have only mentioned a single library file. In our example, we discussed a library file containing all the math functions. There may be more than one library file. For example, a second library file might contain all the functions related to formatting output. A third library file might contain functions related to the graphical user interface.

The linker is capable of taking as input more than one library file. Whenever a function (such as "cos") is used but not defined, the linker will search all the library files in order to locate a module containing the "cos" function and will include it in the executable.

About the Library File

Each "**.o**" object file contains all the data and code bytes specified in a single "**.s**" assembly source code file.

Within that object file, there will typically be a number of symbols which are defined and exported. There may also be symbols which are imported and used, but not defined in the file. The object file may contain a single function or several functions and may contain data as well.

(As mentioned in previous chapters in this document, the code and data within a single object file is broken into segments, but we will ignore segments in this discussion.)

A library file consists of an index, followed by a number of "object modules".

An object module is nothing more than an object file: they have exactly the same format. We say object "module" instead of object "file" because — in this context — it is only a part of the library file, not a file on its own.

Another way to say this is:

A library file consists of a number of object files concatenated together, one after the other, with an index placed at the front. The size in bytes of the library file is exactly the sum of the sizes of all the object files that went into it, along with the size of the index.

A library file is created with the "**createlib**" tool. The input to this tool is a list of all the object files that are to be placed into the library file. The tool reads all the object files, creates the index, then copies the index and all the object files into the newly created library file.

The index contains an entry for each object module in the library. The entry lists the symbols which are exported by that object module.

For example, here is the command to create a library file. We assume that a number of files (such as "**sin.s**", "**cos.s**", "**sqrt.s**", "**log.s**", …) have been previously assembled. This command will create a new file, which is given a name following the "**-o**" output option.

```
createlib sin.o cos.o sqrt.o log.o … -o math.lib
```

Now assume that a programmer has created a program consisting of two assembler source code files called "**MyProg.s**" and "**MoreCode.s**", and wishes to create an executable file "**MyExe**".

```
asm MyProg.s          Creates "MyProg.o"
asm MoreCode.s        Creates "MoreCode.o"
```

In order to create an executable file, the linker tool will be used to combine the material from "**MyProg.o**" and "**MoreCode.o**". In addition, the library file called "**math.lib**" will be consulted, along with a second library file called "**output.lib**", which contains functions related to output formatting.

Here is the command to create the executable. The executable filename, "**MyExe**", is given after the "**-o**" output option.

```
link MyProg.o MoreCode.o math.lib output.lib -o MyExe
```

When the linker tool is used, it begins by reading the index for each and every library file that is to be consulted.

Then the linker will read every "**.o**" file and include that material in the executable file. After this step, if there are symbols which have been imported but not exported by any of the object files, the linker will consult the library indexes.

If the linker tool can locate an object module in one of the libraries that exports the needed symbol, the linker will include the material from that object module in the executable. If the linker cannot find any module that exports the needed symbol, it will issue an error message to the effect that "The symbol *xxx* is undefined; it is imported but not exported by any object file."

The linker will continue to add modules from the library files until all symbols have been defined.

An object module in a library may itself import a symbol. This will cause other object modules to be added to the growing executable file.

The order in which the object files and library files are listed on the linker command line does not matter. The material from all object files will be added to the executable file.

The linker is able to determine whether an input file is an object file or a library file. The linker ignores the "**.o**" and "**.lib**" extensions. These extensions are customary and useful for humans to know what is in the file, but the linker doesn't use them to determine what sort of file it is. Instead, the linker looks at the file contents directly. Object files can be distinguished from library files because each type of file begins with a different "magic number", which the linker uses to determine what the file contains.

The first 8 bytes of the file will be:

Magic number	As ASCII	Meaning
0x4236_346F_626A_6374	B64objct	This is an object file
0x4236_346F_5F6C_6962	B64o_lib	This is a library file

Typically, each object module in a library file will contain a single function, but this doesn't have to be the case. Next, we examine a more complex example in which a single object module may export several symbols and where an object module in a library can itself import a symbol from another object module.

A single object module may contain several functions. Such an object module would presumably export several symbols, one for each function it contains. In other words, whenever a single module in a library file contains several functions, the name of each function would presumably be exported. Each function will begin with a labeled instruction and those labels would be exported.

If any single symbol is used in the main program, it will cause the linker to pull in the entire module, with all the functions it includes, as well as all the symbols the module defines and exports. So, in the case where a single module contains several functions, the use of any one of those functions will cause all the functions in the module to be included in the executable.

Normally this behavior is not what is wanted, and the builder of the library will place each function in a separate module. Then, the inclusion of one function will not cause the other functions to be added, unless it specifically uses some other function (i.e., it imports the symbol naming some other function).

The object modules in a library need not contain only functions. They can contain arbitrary bytes.

As an example of modules that contain data, consider the implementation of the "sin" and "cos" functions. One possible implementation is to include a table of pre-computed values and compute the "sin" function by simply looking up the value in the table.

The value of sin(x), where x ranges from 0° to 90°, is sufficient to capture the shape of the entire sin curve, since sin(x) for all other values of x can be computed using simple identities.

A reasonable implementation is to include data points for (say) 10,000 values of x from 0° to 90° in a look-up table. Of course there are an infinity of values between 0° to 90°, but 10,000 seems like a reasonable number to include. For intermediate values of x not included in the table, the algorithm will look up the values for the nearest two points and perform a linear extrapolation. Using this general approach, very precise values for sin can be computed.

The shape of the cos curve is identical to the sin curve, only shifted in phase, and one implementation of cos might make use of the same table of values. The table is relatively large and we only want to include it in the executable file if either sin or cos is used.

In this example, there will be three separate modules: (1) the sin function, (2) the cos function, and (3) the table of values, which is needed by both functions. The use of either sin or cos will cause the module containing the table to be loaded. If both sin and cos are used, then only one copy of the table will be loaded.

This can be achieved as follows: The source file containing the table will export a single symbol, namely the label addressing the first element in the table. Both the source files for sin and cos will import this symbol.

[This example was somewhat contrived. It seems more likely that the sin function and the table would be combined into a single module, while cos would be implemented as a function that adds 90° to x and then calls the sin function to do all the work. I believe a better design would be to put sin, cos, and the table all into a single module.]

When a library file is created, the **createlib** tool combines a number of object files into a single file. Each object file will export one or more symbols and may import symbols, as well.

Each object module in a library must export at least one symbol, otherwise there is no way for that object module to be pulled in to the executable file. The **createobj** tool will check this, and issue an error message if necessary.

Two object modules in a single library must not export the same symbol. The **createlib** tool will check this, and issue an error message if necessary. Likewise, during linking, the same symbol must not be exported multiple times, from different object files or from modules brought in from different libraries. The linker tool will check this, and issue an error message if necessary.

However, we do allow an object module to export a symbol that is also exported from a module in a library file, as long as the library module is not brought in for inclusion in the executable file. The reason is this: It is possible that the library contains some function which just happens to have a common name that a programmer has coincidentally chosen for an unrelated meaning.

For example, a program concerned with computing the energy efficiency of a wind turbine might reasonably define a function named "power" to compute the wattage of a turbine. Unknown to the programmer, the math library might contain a module which happens to export the symbol "power", for example to compute the function x^n. No error will be reported since there is no ambiguity. The programmer need not ever know that he/she happened to choose a symbol spelling that coincided with a symbol in the math library.

Each object module in a library may import symbols. A symbol imported by one object module need not be exported by another module in that library. During linking, any imported symbol (regardless of whether imported by an object file or by an object module included from a library) must be exported exactly once by some other object file or object module. The linker tool will check this, and issue an error message if necessary.

The Version Number Field

Following the magic number is a "version number". This document describes "version 1" of the file format.

Note The object file format and the executable file format both contain an "ISA Architecture" field, in addition to the "version number". There is no "ISA Architecture" field in the library file header, since our approach is not dependent on the ISA. A modification or change to the ISA should never require a change to the library header.

However, note that the individual object modules each contain an "ISA Architecture" field, so any alteration to the architecture version will be represented in the library file, within the individual modules.

Appendix 1: Machine Instructions

Format A-0   *<no operands>*

ILLEGAL	Canonical form of illegal instruction
SYSRET	PC ← csr_prev; csr_status ← csr_stat2
SLEEP1	Enter light sleep state
SLEEP2	Enter deep sleep state
RESTART	Same as Power-On-Reset
DEBUG	
BREAKPOINT	
TLBCLEAR	Invalidate all TLBs for current ASID
FENCE	

Format A-1   *Reg1*

CHECKB	r1	Trap if reg not within -128 … +127
CHECKH	r1	Trap if reg not within -32768 … +32767
CHECKW	r1	Trap if reg not within 32 bit range
PUTSTAT	r1	CSR_STATUS [9:3] ← Reg1 [9:3]
TLBFLUSH	r1	Invalidate TLB for virtual address in Reg1

Format A-2   *RegD,Reg1*

ENDIANH	r7,r1	Reorder bytes: 76543210 → 67452301
ENDIANW	r7,r1	Reorder bytes: 76543210 → 45670123
ENDIAND	r7,r1	Reorder bytes: 76543210 → 01234567
SEXTB	r7,r1	Sign extend byte to 64 bits
SEXTH	r7,r1	Sign extend 16 bits to 64 bits
SEXTW	r7,r1	Sign extend 32 bits to 64 bits
FNEG	r7,r1	
FABS	r7,r1	
FSQRT	r7,r1	
FCLASS	r7,r1	RegD ← classify(Reg1) || FLOAT_STATUS
FCVTFI	r7,r1	Convert: floating-point ← int

```
        FCVTIF      r7,r1              Convert: int ← floating-point
```

Format A-3   *RegD,Reg1,Reg2*

```
    ADD         r7,r1,r2
    ADDOK       r7,r1,r2
    SUB         r7,r1,r2
    MUL         r7,r1,r2
    DIV         r7,r1,r2
    REM         r7,r1,r2
    AND         r7,r1,r2
    OR          r7,r1,r2
    XOR         r7,r1,r2
    SLL         r7,r1,r2
    SLA         r7,r1,r2
    SRL         r7,r1,r2
    SRA         r7,r1,r2
    ROTR        r7,r1,r2
    TESTEQ      r7,r1,r2
    TESTNE      r7,r1,r2
    TESTLT      r7,r1,r2
    TESTLE      r7,r1,r2
    FEQ         r7,r1,r2
    FLT         r7,r1,r2
    FLE         r7,r1,r2
    FADD        r7,r1,r2
    FSUB        r7,r1,r2
    FMUL        r7,r1,r2
    FDIV        r7,r1,r2
    FMIN        r7,r1,r2
    FMAX        r7,r1,r2
```

SLA — Shift-left-arithmetic; checks for overflow

ROTR — Rotate right; no overflow check

TESTEQ — $RegD \leftarrow (Reg1 = Reg2) ? 1 : 0$

TESTNE — $RegD \leftarrow (Reg1 \neq Reg2) ? 1 : 0$

TESTLT — $RegD \leftarrow (Reg1 < Reg2) ? 1 : 0$

TESTLE — $RegD \leftarrow (Reg1 \leq Reg2) ? 1 : 0$

FEQ — $RegD \leftarrow (Reg1 = Reg2) ? 1 : 0$ (float compare)

FLT — $RegD \leftarrow (Reg1 < Reg2) ? 1 : 0$ (float compare)

FLE — $RegD \leftarrow (Reg1 \leq Reg2) ? 1 : 0$ (float compare)

Format A-4   *RegD,Reg1,Reg2,Reg3*

```
    ADD3        r7,r1,r2,r3     Reg3 ← Reg1+Reg2+Reg3 (unsigned)
    ALIGNH      r7,r1,r2,r3     Reg3 (unaligned addr) gives shift amount
    ALIGNW      r7,r1,r2,r3     Reg3 (unaligned addr) gives shift amount
    ALIGND      r7,r1,r2,r3     Reg3 (unaligned addr) gives shift amount
    INJECT1H    r7,r1,r2,r3     RegD ← Reg1; inject Reg2 per addr in Reg3
```

```
INJECT2H   r7,r1,r2,r3      RegD ← Reg1; inject Reg2 per addr in Reg3
INJECT1W   r7,r1,r2,r3      RegD ← Reg1; inject Reg2 per addr in Reg3
INJECT2W   r7,r1,r2,r3      RegD ← Reg1; inject Reg2 per addr in Reg3
INJECT1D   r7,r1,r2,r3      RegD ← Reg1; inject Reg2 per addr in Reg3
INJECT2D   r7,r1,r2,r3      RegD ← Reg1; inject Reg2 per addr in Reg3
FMADD      r7,r1,r2,r3      RegD ← (Reg1 × Reg2) + Reg3
FNMADD     r7,r1,r2,r3      RegD ← (-(Reg1 × Reg2)) + Reg3
FMSUB      r7,r1,r2,r3      RegD ← (Reg1 × Reg2) - Reg3
FNMSUB     r7,r1,r2,r3      RegD ← (-(Reg1 × Reg2)) - Reg3
MULADD     r7,r1,r2,r3      RegD ← (Reg1 × Reg2) + Reg3
MULADDU    r7,r1,r2,r3      RegD ← (Reg1 × Reg2) + Reg3 (unsigned)
INDEX0     r7,r1,r2,r3      Reg1=arrayPtr, Reg2=header, Reg3=index
INDEX1     r7,r1,r2,r3      .  RegD ← Reg1 + 8 + (Reg3 * scale)
INDEX2     r7,r1,r2,r3      .  Reg2=header=[ArrayMAX||ArrayCURR]
INDEX4     r7,r1,r2,r3      .  Trap if (Reg3 < 0) or (Reg3 ≥ ArrayCURR)
INDEX8     r7,r1,r2,r3      .           or (ArrayMAX = 0)
INDEX16    r7,r1,r2,r3      .
INDEX24    r7,r1,r2,r3      .
INDEX32    r7,r1,r2,r3      .
CAS        r7,r1,r2,r3      Compare and Swap: If *r1=r2 then *r1←r3
```

Format A-5   _Reg1,Reg2_
  <no longer used>

Format A-6   _Reg2_
```
  <no longer used>
```

Format A-7   _RegD,Reg1,Reg2_
```
  CSRSWAP   r7,csr,r2        Reg1 encodes CSR; RegD ← CSR; CSR ← Reg2
```

Format A-8   _RegD,Reg1_
```
  CSRREAD   r7,csr           Reg1 encodes CSR; RegD ← CSR;
```

Format A-9   _RegD_
```
  GETSTAT   r7,csr           RegD ← CSR_STATUS & 0x0000…03f8
```

Format B-1   _RegD,Reg1,immed-16_
```
  ADDI      r7,r1,0x1234
```

```
ANDI       r7,r1,0x1234
ORI        r7,r1,0x1234
XORI       r7,r1,0x1234
TESTEQI    r7,r1,0x1234
TESTNEI    r7,r1,0x1234
TESTLTI    r7,r1,0x1234
TESTLEI    r7,r1,0x1234
TESTGTI    r7,r1,0x1234
TESTGEI    r7,r1,0x1234
UPPER16    r7,r1,0x1234
SHIFT16    r7,r1,0x1234
CONTROL    r7,r1,0x1234
CONTROLU   r7,r1,0x1234
```

`TESTEQI   r7,r1,0x1234`	RegD ← (Reg1=immed) ? 1 : 0
`TESTNEI   r7,r1,0x1234`	RegD ← (Reg1≠immed) ? 1 : 0
`TESTLTI   r7,r1,0x1234`	RegD ← (Reg1<immed) ? 1 : 0
`TESTLEI   r7,r1,0x1234`	RegD ← (Reg1≤immed) ? 1 : 0
`TESTGTI   r7,r1,0x1234`	RegD ← (Reg1<immed) ? 1 : 0
`TESTGEI   r7,r1,0x1234`	RegD ← (Reg1≥ immed) ? 1 : 0
`UPPER16   r7,r1,0x1234`	RegD ← (immed<<16) + Reg1
`SHIFT16   r7,r1,0x1234`	RegD ← (Reg1+immed) << 16

Format B-2   *RegD,immed-16(Reg1)*

`LOAD.B`	`r7,offset(r1)`	Value is sign-extended to 64 bits
`LOAD.H`	`r7,offset(r1)`	. May cause unaligned exception
`LOAD.W`	`r7,offset(r1)`	. No overflow check on addr calculation
`LOAD.D`	`r7,offset(r1)`	
`JALR`	`lr,offset(r1)`	RegD ← return addr; Target ← offset+Reg1

Format B-3   *RegD,Reg1,immed-3*

`CHECKADDR r7,r1,5`	Reg1 = virt addr; RegD ← except. code or 0

Format B-4   *immed-10*

`SYSCALL   123`	`immed-10` selects one of 1,024 syscalls

Format B-5   *RegD,Reg1,immed-6*

`SLLI      r7,r1,5`	
`SLAI      r7,r1,5`	Shift-left-arithmetic checks for overflow
`SRLI      r7,r1,5`	
`SRAI      r7,r1,5`	
`ROTRI     r7,r1,5`	Rotate right; no overflow check

Format B-6   *Reg1,immed-16*

`CSRSET    csr,0x1234`	Reg1 encodes CSR; Set selected bits in CSR
`CSRCLR    csr,0x1234`	Reg1 encodes CSR; Clear selected bits in CSR

Format C-1   *immed-16(Reg1),Reg2*

```
STORE.B   offset(r1),r2     Upper bits in reg are ignored
STORE.H   offset(r1),r2     . May cause unaligned exception
STORE.W   offset(r1),r2     . No overflow check on addr calculation
STORE.D   offset(r1),r2
```

Format C-2   *Reg1,Reg2,immed-16*

```
B.EQ      r1,r2,MyLabel     Branch if Reg1=Reg2; Offset is PC-relative
B.NE      r1,r2,MyLabel     Branch if Reg1≠Reg2; Offset is PC-relative
B.LT      r1,r2,MyLabel     Branch if Reg1<Reg2; Offset is PC-relative
B.LE      r1,r2,MyLabel     Branch if Reg1≤Reg2; Offset is PC-relative
```

Format D-1   *RegD,immed-20*

```
UPPER20   r7,MyLabel     RegD ← (immed<<16)
AUIPC     r7,MyLabel     RegD ← (immed<<16) + PC
ADDPC     r7,MyLabel     RegD ← immed+PC
JAL       lr,MyLabel     RegD ← return addr ; Target ← PC+immed
```

Appendix 2: Command Line Tools

Quick Summary

• The following tools are discussed:

asm	Assembler
link	Linker
createlib	Tool to create library files
dumpobj	Tool to display object files
hexdump	Tool to display contents of binary files

• For each tool, the command line options are described.

The Assember Tool

The assembler tool is a program named "**asm**". A typical use is:

```
asm hello.s
```

A particularly useful option is "**-l**", which will produce a listing. This is useful in seeing exactly what machine codes are being produced by the assembler.

```
asm hello.s –l
```

The following command line options may be given in any order:

filename

> The input source will come from this file. (Normally this file will end with "**.s**".) If an input file is not given on the command line, the assembly source code program will come from stdin. Only one input source file is allowed.

–o *filename*

(oh) If there are no errors, an object file will be created. The **-o** option can be used to give the object file a specific name. If this option is not used, then the input source file must be named on the command line (the source must not come from stdin). If **-o** is not used, the name of the object file will be computed from the name of the input file by removing the ".**s**" extension, if any, and appending ".**o**". For example:

```
test.s   →   test.o
foo      →   foo.o
```

–h

Print information describing the command line options, which is roughly identical to the information in this section. All other options are ignored and the tool terminates immediately.

–l

(el) Print a listing on stdout. The listing shows the entire source file and, for every line, indicates what bytes have been produced. The listing is best viewed in a fixed-width font.

–g

Produce output in the "**load-and-go**" format, instead of the normal object format. A load-and-go program must consist of a single segment and not import any symbols. The output will not be linked and must not contain anything requiring the linker.

Instead, the output file is a special executable format called "load-and-go". The output file can be loaded directly into memory and executed. Typically, the output file will be given a name that does not have the ".**o**" extension:

```
asm hello.s –g –o hello
```

A load-and-go file will not contain any symbol table information, which will make debugging more challenging.

–hex

Produce output in hex format, instead of the normal object format. The program must consist of a single segment and not import any symbols. The output will not be linked and must not contain anything requiring the linker.

The output file will be a series of lines of ASCII hex numbers.

–w

This option will suppress all warning messages.

–z

Wait for the linker. Defer the translation of some synthetics instructions to the linker, which may find slightly shorter translations in a few rare cases.

This option will force the assembler to defer to the linker all synthetic translations that are not guaranteed to be optimal.

This primarily concerns a JUMP/CALL to an absolute value that the assembler determines can be done in two instructions. However, if the linker happens to place the segment containing the JUMP/CALL close to the segment containing the target address, it might be possible for the linker to translate the JUMP/CALL using a single PC-relative JAL instruction. This option forces the assembler to only translate JUMP/CALL instructions when it can be done in one instruction, or when the target address is not a valid memory address.

A similar situation occurs with a MOVI that is moving an absolute value in the range 0x0_0000_8000 ... 0xF_FFFF_FFFF into a register. Such an instruction is likely to be loading the address of a JUMP/CALL target and will require two instructions if done by the assembler. The linker may be able to translate the MOVI with a single ADDPC instruction. This option will prevent the assembler from translating the MOVI using two instructions.

This situation can also be triggered for a segment which is not assigned a value for "**gp=**". Since the assembler doesn't know whether this segment will be in kernel space or in user space, it cannot assigned the default. It is possible that the linker will assign a default value that will make shorter

instruction sequences for MOVI, JUMP, CALL, Bxx, LOADx, and STOREx instructions usable.

–zw

This option is related to the **-z** option. This option will cause warnings to be generated whenever the assembler is synthesizing an instruction in a way that might not be optimal.

–s

Print the symbol table on stdout. This listing lists each symbol in the source file and, for each, shows its attributes, including its value (if known), whether the symbol is imported or exported, and which line the symbol was defined on. The output should be viewed in a fixed-width font.

–nodebug

By default, the assembler adds debugging info to the **.o** output file. This option suppresses this; if present no debugging information will be put into the output file. This option causes the assembler to ignore the debugging pseudo-ops, namely:

> **.sourcefile**
> **.function**
> **.endfunction**
> **.global**
> **.local**
> **.regparm**
> **.stmt**
> **.comment**

–d

Print internal assembler info (for debugging asm.c). This option may become disabled in the future. Generally speaking, this option will cause the "instruction list" to be printed. This is the internal representation of all instructions after the source file has been read in and parsed.

This option will also cause .skip instructions with extremely large values to be treated differently. Such instructions occur in the test files; with this option long runs of 0x00 will not be written out to the object file.

–d2

Print internal assembler info (for debugging asm.c). This option may become disabled in the future. Generally speaking, this option will print info tracing the **ProcessSynthetics** algorithm.

The Linker Tool

The linker tool is a program named "**link**". A simple use is:

```
link hello.o -o hello
```

At least one object file (such as "hello.o") is required.

The executable file that is to be produced (e.g., "hello") must also be specified. The "**-o**" option must be followed by the filename of the executable.

A more typical example includes several object files and libraries:

```
link  hello.o  fred.o  myLib.lib  math.lib  -o hello
```

The following command line options may be given in any order:

–h

Print information describing the command line options, which is roughly identical to the information in this section. All other options are ignored and the tool terminates immediately.

filename

> One or more input files must be specified on the command line. Each is assumed to be either a "**.o**" object file or a "**.lib**" library file. They may be given in any order. There must be at least one object file specified.

–o *filename*

> The name of the file to be created is required. If the file already exists, it will be overwritten.

–k

> If this option is present, all code and data segments will be placed in the kernel address space. Otherwise, they will be placed in the user address space.

	From	To
Kernel Address Region:	0x0_0000_0000	0x3_FFFF_FFFF
User Address Region:	0x8_0000_0000	0xF_FFFF_FFFF

–s

> This option causes the linker to print out the internal symbol table and other information about the linking process. The output should be viewed in a fixed-width font.

–s1

> This option causes the linker to print out an overview of memory usage for the resulting executable file.

–s2

> The linker will add information to the executable file that is intended only to be used by a debugger tool. This option will print out this information in human-readable form. This option is independent of option **-s**; they each print different information. The output should be viewed in a fixed-width font.

–w1

When synthesizing some instructions (e.g., JUMP, LOADx, STOREx, Bxx), the linker will compute the target address. If the value is not within the legal 36 bit range, (i.e., not within 0x0 ... 0xF_FFFF_FFFF) the linker will print a warning and ignore the upper 28 bits.

The **-w1** option causes the linker to suppress this warning.

–w2

When synthesizing some instructions, the linker may occasionally insert a NOP instruction after the machine code translation. If this occurs, a warning will be printed.

The **-w2** option causes the linker to suppress this warning.

The insertion of a NOP is a side-effect of the algorithm and does not indicate an error. It can occur when a forward JUMP initially required two machine code instructions; later, the translation of other instructions can move the JUMP forward, suddenly making a single machine code instruction adequate. To ensure algorithm termination, the translations can only grow, never shrink. The NOP should be harmless, aside from a small impact on execution speed.

–w3

Normally variables should be placed in a segment marked "writable, but not executable". Code should be placed in a segment marked "executable, but not writable". Read-only constants can go into either a code segment marked "executable, but not writable" or a segment marked "not executable and not writable."

A segment marked "executable and writable" is unusual and is not recommended. Programs that are able to modify themselves make life much easier for malware. Such segments are discouraged and a warning will be generated if the linker encounters such a segment.

The **-w3** option causes the linker to suppress this warning.

–w

> This option will suppress all warning messages. It is equivalent to "**-w1 -w2** …". If **-w** is used, the others (**-w1**, **-w2**, …) must not be used.

–d*XXX*

> Options of this form (such as **–d** and **–d4**) were used during debugging of the linker tool itself. They cause the printing of various internal data structures. These options are not useful to users and may be disabled in the future. For details, consult the source code of the linker tool.

The "createlib" Tool

To create a new library file, a tool named "**createlib**" is used. For example:

```
createlib sin.o cos2.o sqrt.o log.o –o math.lib
```

At least one object file is required and there are typically many.

The "**-o**" option must be followed by the filename of the output file. It is required.

The following command line options may be given in any order:

–h

> Print information describing the command line options, which is roughly identical to the information in this section. All other options are ignored and the tool terminates immediately.

filename

> One or more input files must be specified on the command line. Each is assumed to be a "**.o**" object file. There must be at least one object file specified, and their order is irrelevant.

–o *filename*

> The name of the file to be created is required. If the file already exists, it will be overwritten. It will typically end with ".lib" but this is not required.

–s

> Print the symbol table on stdout. A listing of each exported symbol and the module that exported it is printed.

The "dumpobj" Tool

The "**dumpobj**" tool will read a file and print its contents in a human-readable form on stdout. It can handle the following types of files:

> Object (**.o**) files
> Library (**.lib**) files
> Executable (**a.out**) files
> Load-and-go files

The dumpobj tool understands the formats used in these files. It will read a file and display the information in a form that is appropriate for the file type. This tool will also do some error checking on the file and, if problems in the file are encountered, the tool will print an error message and terminate. This tool will not modify any files.

The following command line options may be given in any order:

filename

> The input source will come from this file. If an input file is not given on the command line, the input will come from stdin. Only one input file is allowed.

–h

> Print information describing the command line options, which is roughly identical to the information in this section. All other options are ignored and the tool terminates immediately.

-v

>The "**v**" stands for "verbose". Header information, symbol information, and patch information is always printed. This option controls whether the data in the segments is printed. If present, the instructions and data are also printed.

The "hexdump" Tool

The "**hexdump**" tool will read a file and print its contents on stdout. It can handle any kind of file. The file contents will be printed both in hex and interpreted as ASCII.

For example, the following command:

```
% hexdump hexdump.c
```

will produce this output:

```
000000000:  2F2F 2054  6865 2042  6C69 747A  2D36 3420     // The Blitz-64
000000010:  2268 6578  6475 6D70  2220 546F  6F6C 0A2F     "hexdump" Tool./
000000020:  2F0A 2F2F  2062 7920  4861 7272  7920 482E     /.// by Harry H.
000000030:  2050 6F72  7465 7220  4949 490A  2F2F 2043      Porter III.// C
000000040:  6F70 7972  6967 6874  2032 3031  380A 2F2F     opyright 2018.//
000000050:  0A2F 2F20  5468 6973  2070 726F  6772 616D     .// This program
000000060:  2072 6561  6473 2061  2066 696C  6520 616E      reads a file an
    ...
```

This tool will not modify any files.

If the file happens to be a properly formatted UTF-8 file, then all ASCII characters will be displayed, but all remaining Unicode characters will be replaced with dots on the righthand side. This tool's output is purely ASCII.

For example, a file containing these characters:

```
café, naïve, x←(2÷3)
```

will be displayed as:

```
000000000:  6361 66C3  A92C 206E  61C3 AF76  652C 2078    caf.., na..ve, x
000000010:  E286 9028  32C3 B733  29                      ...(2..3)
```

The following command line options may be given in any order:

filename

> The input will come from this file. If a file is not given on the command line, the input will come from stdin. Only one input file is allowed.

–h

> Print information describing the command line options, which is roughly identical to the information in this section. All other options are ignored and the tool terminates immediately.

Appendix 3: The Assembler Algorithm

Introduction

We next describe the assembler algorithm that translates the remaining synthetic instructions into machine code instructions. Some synthetic instructions cannot be translated until link time and these will remain untranslated. Those that can be translated will be replaced with the correct machine instruction sequences.

ProcessSynthetics

The function in the assembler (i.e., in asm.c) which uses this algorithm is called "**ProcessSynthetics**".

Before this function is called, the simpler synthetic instructions will have been dealt with. Each remaining synthetic instructions will be one of

    Format-S1
    Format-S2
    Format-S3
    Format-S4
    Format-S5
    Format-S6
    Format-S7

For each of these, the translation has a variable length. This means the synthetic instruction may be expanded into several machine instructions. Possible translations are:

4 bytes	1 machine instruction
8 bytes	2 machine instructions
12 bytes	3 machine instructions
16 bytes	4 machine instructions

The function that performs an individual translation is called "**SynthesizeInstruction**". It takes two arguments:

- Instruction Pointer
- wantAction

Each instruction is represented with an instance of "**struct Instruction**". All instructions in the source code file are kept in a single linked list of these **Instruction** objects.

The "**instruction pointer**" points to an Instruction object in the linked list of instructions. If translation is possible, this function will replace a single synthetic instruction by one or more machine instructions.

The "**wantAction**" parameter is a boolean. If TRUE, the translation will take place and the instruction list will be modified. If FALSE, then no modifications will occur; This happens when the function is being called to determine if synthesis could take place, given the current conditions, and if so, how big the translation would be.

The **SynthesizeInstruction** function will return an integer indicating success or failure, and the size of the translation.

-1	FAILURE:	There was a problem and the translation could not be done.
4, 8, 12, 16	SUCCESS:	The size of the translation, in bytes.

Each instruction is represented with one **Instruction** object. The following fields on Instruction are used by this algorithm:

actualSize   The number of bytes required for this synthetic instruction Could be 4,8,12,16. -1 means variable/unknown/linker required.

maximumSize   The maximum number of bytes required for this synthetic instruction. Could be 4,8,12,16. Set once and then used once we determine we can't do anything with this instruction.

myLC   The offset of this instruction from the beginning of this domain.

myDomain   Which domain this instruction is in.

A "**domain**" is a sequence of instructions. All instructions in the sequence have an exact, known size, except possibly the last instruction. Relative offsets within a single domain can be computed with certainty.

In general, the assembler does not know where in memory the linker will place each segment.

The **.align** instruction presents a unique challenge. Since the assembler doesn't know exactly where in memory the segment will be placed, it cannot determine how many bytes will be inserted by the linker for each **.align** instruction. Thus, **.align** instructions are like synthetic instructions that must be handled by the linker.

[Prior to this algorithm, all "**.align 2**" and "**.align 4**" instructions were replaced with "**.skip 1/2/3**" instructions, so they are gone. All remaining **.align** instructions — that is, **8**, **16**, **32**, or **page** — are treated as unknowable by this algorithm. Even if a "startaddr=" is given for the segment, it will not be used for **.align** instructions, even though we could, in theory, determine exactly how many bytes some **.align** instructions would insert.]

The last instruction in a domain will be either a synthetic instruction whose size we cannot determine, an **.align** instruction, or the last instruction in a segment. Every **.begin** instruction will cause a new domain to be started. Likewise, a new domain will be started directly after a synthetic instruction whose size we cannot determine, and after every **.align** instruction.

Consider a synthetic instruction within some domain. The assembler can compute the exact offset from that synthetic instruction to another location, as long as that

location is in that domain. If the target location is in another domain, then the assembler cannot determine the relative distance between them. (This is because they are either in different segments or are separated by an **.align** or synthetic instruction whose size we cannot determine.)

Domains are identified by numbers and numbers are assigned sequentially so it is easy to determine whether two domains are equal.

If the exact starting locations of segments are given in the **.begin** instructions (using "startaddr="), then it might be possible to deduce the relative offset between two locations in different segments. However, this algorithm will not handle relative offsets between segments, even if they could, in theory, sometimes be inferred.

First Phase

In the first phase of the algorithm, we make the assumption that every synthetic instruction will be translated. The best case assumption is that each segment will be reduced to a single domain.

In this case, some synthetics will simply be impossible to translate, because they rely on imported symbols. In the first phase, we will identify these synthetic instructions and immediately give up on them. We will assume these will take the maximum size, and we will use a negative number (-4, -8, -12, or -16) to indicate that they cannot be translated.

However, for the remainder of the synthetic instructions, there is some hope that we will ultimately be able to translate them. So we will begin by assuming those synthetic instructions can be translated with only one (4 byte) instruction.

*// Initialize **domain** and **myLC**...*
LOOP thru the instruction list...
    Place each segment into a single domain
    Set "**actualSize**"
        For machine instructions, use the exact size (i.e., 4 bytes)
        For synthetics and **.align**, use the *maximum* possible sizes
    Also set "**myDomain**" and "**myLC**" for each instruction.
    For symbols used as labels, set their "**domain**" and "**offset**" fields.

// Determine which synthetics are simply not translatable...
LOOP thru the instruction list; look only at Format S instructions.
    Call "SynthesizeInstruction" — with arg "wantAction" = NO
        Get a size for this instruction, or -1 if not synthesizable.
        If we get a number, save it in "**maximumSize**", for later.
        Otherwise if we get -1, set "**actualSize**" to -(maxSize for this
            type of instruction)
        If size is a number, set "**actualSize**" to 4, the minimum.

The reason we must do it this way is shown by the following example:

```
1               .import    Undef
2       L3:
3               jump       Undef     # Unknown size - Can't synthesize
4       L4:
5               jump       L3        # Size 4, but can't synthesize
6       L5:
7               jump       L4        # Size 4, can synthesize
```

The jump on line 3 cannot be synthesized. However, it can be 8 bytes at most, which is the maximum size for any JUMP instruction. Since the distance from the JUMP on line 5 to "L3" is small, the assembler can determine that the JUMP on line 5 will require exactly 4 bytes. But the assembler can't know exactly what that distance is, so it can't synthesize the JUMP on line 5. Since the assembler knows the size of the JUMP on line 5, if not the exact value, it can synthesize the JUMP on line 7.

The information we pass to the linker is:

    The linker must synthesize the JUMPS on lines 3 and 5.
        The jump on line 3 can be any size.
        The jump on line 5 will take exactly 4 bytes.
    The jump on line 7 has already been synthesized; the linker will ignore it.

Second Phase: Relaxation

The second phase of the algorithm is essentially a "relaxation algorithm". We have previously set the size of every synthetic instruction that might be synthesizable to 4 bytes. Each "slot" is set to the minimum size and will gradually be enlarged until every slot is large enough to accommodate the translation.

First, we go through and assign addresses to all instructions and labels. We also re-assign domains.

Then, given the assignment of addresses and domains, we determine which synthetic instructions can actually be synthesized in the amount of space we have set aside for them. In some cases, the 4 bytes will be enough. However, for some, we may need more than 4 bytes. If so, we increase the number of bytes to accommodate the translation.

Then, if any synthetic instruction required more bytes than we had initially counted on, we need to repeat. We keep repeating until nothing further changes.

```
        somethingChanged = TRUE
        LOOP WHILE (somethingChanged)
            somethingChanged = FALSE

            // Re-assign LCs...
            LOOP thru instruction list
                Set "myLC" based on "actualSize"
                Set "myDomain"
                    For .begin and .align, start a new domain
                    Otherwise, create one domain per segment
                For symbols used as labels, set their "domain" and "offset"

            // Check that "ActualSize" is adequate and enlarge as necessary...
            LOOP thru instruction list; look only at format S instructions.
                If "actualSize" > 0
                    Call SynthesizeInstruction() — with arg "wantAction" = NO
                    If returned value  ==  -1
                        It was synthesizable before, but now it can't be.
                        Set "actualSize" = saved "maximumSize".
                If newSize > "actualSize"
                    "actualSize" = newSize
                    somethingChanged = TRUE
        END WHILE LOOP
```

[Since we are enlarging the slot sizes on each iteration and there is a maximum possible slot size (12 bytes), this repeat-until-no-changes loop will terminate. Most likely, the first iteration will determine the sizes we need and a few synthetic instruction slots will be enlarged to whatever is actually needed. In the second iteration, there will likely be no changes and the looping will be done. However, it is possible that the growth of one slot will have the consequence of moving two other things a little farther apart, requiring some other instruction that previously required 4 bytes to suddenly pass a threshold and require 8 bytes. In some pathological case, there might several iterations.]

In **the fourth phase**, the algorithm will again run through the instructions and assign locations to everything. Then it will loop through the instructions and actually perform the translations. We have already determined how many bytes are required, so we know when the translation can be done and how big it will be.

> *// Assign accurate LCs...*
> LOOP thru instruction list
>   Set "**myLC**" based on "**actualSize**"
>   Set "**myDomain**"
>   For **.begin** and **.align**, start a new domain
>   If "**actualSize**" < 0, then start a new domain
>   For symbols used as labels, set their "**domain**" and "**offset**"
>
> *// Perform the transformations...*
> LOOP thru instruction list; look only at format S.
>   If "**actualSize**" > 0
>   Call **SynthesizeInstruction**() — with arg "wantAction" = YES
>   If returned size == -1
>   Ignore; the target moved to a different domain
>   If returned size ≠ "**actualSize**"
>   ProgramLogicError

In **the final step**, we may make one last pass through the instructions to set the addresses and sizes so everything is consistent. In particular, we allocate zero bytes for all **.align** instructions (which the linker may increase) and 4 bytes for every remaining untranslated synthetic instruction (which the linker may increase).

> *// Finalize the "actualSize" and "LC" values...*
> LOOP thru the instruction list
>   For remaining synthetics, set "**actualSize**" to 4 bytes.
>   For **.align**, set "**actualSize**" to 0 bytes.
>   Set "**myLC**" based on "**actualSize**"

BUGS AND PROBLEMS: We still have some issues that need attention.

After this algorithm, **actualSize** will be...
    Negative (-4, -8, -12, -16)... indicates a mandatory size
    4 = no assumptions about size were made.

In the code, we consider expanding a synthetic instruction into a larger sequence. Right now, the code in **SynthesizeInstruction** always assumes the slot size is 4. It adjusts the offset if the target is BEFORE the synthetic but not AFTER. This is because it will be inserting an instruction.

Furthermore, it determines whether the adjustment is needed by looking to see if the target is in the current or following domain. We have changed things so that the target is always in the same domain.

It's possible that the slot size is 8 and is being enlarged to 12 (or from 12 to 16). Furthermore, the test about whether the adjustment is needed is wrong.

Appendix 4: The Linker Algorithm

Quick Summary

- A specific implementation of the linker tool is described.
- The code in the C program "**link.c**" is documented.
- This appendix can safely be ignored unless there is a bug in the linker.
- This appendix may be separated out into a separate document in the future.

Introduction

The linker tool is a C program named "**link.c**" and the executable is named "**link**". The C code includes some standard C libraries and some additional C code from **BlitzSupport.c**. The command line parameters are documented elsewhere. The program terminates with a standard Unix/Linux error code (EXIT_FAILURE, EXIT_SUCCESS).

Error messages and warnings go to the **stderr** output. Additional information may be printed for error and warning messages and this goes to **stdout**. Several command line options (such as **-s** and the various **-d** debugging options) print output which goes to **stdout**.

The following files contain all the linker code:

> **link.c**
> **BlitzSupport.c**
> **CheckHostCompatibility.c**

There are no **.h** header files, which is somewhat atypical of Linux/Unix coding style.

The file **BlitzSupport** contains a number of functions that are used by the linker, as well as other tools in the Blitz project, such as the assembler and the emulator.

The **CheckHostCompatibility.c** file contains a function named **CheckHostCompatibility** which tries to ensure that all assumptions about the host (e.g., byte-order, word size, and C "implementation dependencies") are as expected. This function is called once at startup and any problems cause an immediate halt.

In addition, the following well-known Linux/Unix "includes" are used:

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>

The linker primarily relies on the following "C" types, as well as pointers, arrays, and structs.

int   32 signed integers
int64_t   64 bit signed integers
char   bytes: 8 bit quantities
FILE *   For file I/O

For boolean values, we use type **int** and use **0** and **1** for FALSE and TRUE.

All sizes and lengths are in terms of bytes, and never in terms of words or doublewords.

I have a tendency to avoid defining constants with **#define** and tend to specify the value directly. I do this because I have lost too many debugging hours because I made incorrect assumptions about the value of a "constant".

Linux/Unix system functions that are most heavily used are:

calloc
free
fopen
fclose
fread
fwrite
fseek

feof
perror
errno (a variable)
exit
printf
fprintf

The following functions are also used in other Blitz tools:

strlen
strcmp
fscanf
putchar
fread
fwrite

The most common formatting codes used in **printf** are:

%d
%lld
%x
%llx
%s
%c

The program is compiled with a **make** file named "**makefile**", which contains roughly these lines:

```
CheckHostCompatibility1.s: CheckHostCompatibility.c
     gcc –g –std=c99 –Wall –DBLITZ_HOST_IS_LITTLE_ENDIAN \
                         –DWithoutOpt CheckHostCompatibility.c \
                         –S –o CheckHostCompatibility1.s

CheckHostCompatibility2.s: CheckHostCompatibility.c
     gcc –g –std=c99 –Wall –O2 –DBLITZ_HOST_IS_LITTLE_ENDIAN \
                         –DWithOpt CheckHostCompatibility.c –S \
                         –o CheckHostCompatibility2.s

link: link.c BlitzSupport.c checkHostCompatibility1.s \
                         checkHostCompatibility2.s
     gcc –g –std=c99 –Wall –O2 –DBLITZ_HOST_IS_LITTLE_ENDIAN \
                         –lm link.c checkHostCompatibility1.s \
```

```
checkHostCompatibility2.s –o link
```

<u>Error Handling</u>

The program often performs internal consistency checks and calls function **ProgramLogicError** if anything is wrong. The program also performs checks to make sure the input is well formed and error-free. If anything is amiss, it calls one of the functions: **FatalError**, **FatalErrorInFile**, or **FatalErrorInModule**. All of these functions print a message and terminate the program immediately.

Other types of errors are not fatal and the linker will keep going. In these cases, it prints an error message to **stderr**. There is a counter named **errorCount** which is incremented. Later, this counter is used to determine whether the program should return EXIT_FAILURE or EXIT_SUCCESS. The program also prints warning messages and there is a counter named **warningCount** which is incremented every time a warning is printed.

At certain moments, the program will call a function named **CheckForAbort**, which will take a look at **errorCount** and immediately terminate the program if any errors have been encountered. This prevents earlier errors from possibly leading to inconsistent data structures that might cause serious confusion or program logic errors in later stages of processing.

If the program terminates due to errors, it will remove the output file it created, if it was created.

Pointers and Objects

There are a number of types of objects created by the linker:

InFile	one per input file
Module	one per **.o** file; one per library module
Segment	one per **.begin** statement
Symbol	one per symbol exported or imported
Patch	one per patch entry in an input module
TableEntry	one per exported symbol in a library
Region	one per chunk of memory (containing one or more segments)

For each of these, there is a "C struct" with a number of fields.

Sometimes, we refer to structs as "objects". (Of course since the linker is a C program, there is no subclassing relationship involved.)

Many objects contain fields pointing to another object. For example, each **Segment** object contains a field named **myModule**, which points to the object representing the module which contains this segment. Likewise, each **Symbol** contains a field named **usedInModule**, which contains a pointer to the module that defined that symbol. And each **Patch** object contains a field named **segment**, which contains a pointer to the segment where that patch is to be applied.

This input files read by the linker (the object files and library files) identify things by number. For example, every segment in a module is numbered. Likewise, every symbol is numbered.

Initially, the linker will enter segments and symbols into arrays and use the array indices to locate the objects. But later, once the linker has identified the object by number, it will refer to in with a pointer.

There are a number of linked lists. Mostly, the linked lists are singly linked, with a "next" pointer. An exception is the list of **Region** objects, since it is necessary to insert objects into the middle of the list. The list of **Regions** is doubly linked, with fields named **next** and **prev**.

Most linked list are headed by a pointer to the first element. An example is the global list of all segments in the executable, which is pointed to by a variable named **segmentList**.

However, some linked lists have to be constructed in order, so the new elements have to be added at the tail end. Examples are

The list of input files:
> **firstInfile**
> **lastInfile**

The list of modules:
> **firstModule**
> **lastModule**

The list of symbols:
> **symbolList**
> **symbolListLast**

The list of patches:
> **patchList**
> **patchListLast**

The list of **Region** objects is handled differently. This doubly-linked list is maintained as a circular list. In other words, the are no NULL pointers among the **next** and **prev** pointers. Instead, there is a special dummy "header" **Region** object, which does not represent a valid region. The **next** pointer of the header points to the first real region. The **prev** pointer of the header points to the last real region. The global variable **regionHeader** points to the special dummy region. **Region** objects also contain a field (**regionStatus**) to tell what sort of region it is; a special value (-1) is used to identify the dummy header object.

Print Routines

There are a number of functions which will send characters to **stdout**. These functions are useful in debugging **link.c** and for printing information during normal operation, e.g., for the "-s" option.

PrintLExportedIndex ()
PrintLibraryIndex ()

PrintSymbolList ()
PrintSymbolHeader ()
PrintSymbol (Symbol * sym)

PrintSegmentList ()
PrintSegmentSublist ()
PrintSegment ()

PrintModuleList ()

PrintPatchList ()
PrintPatch (Patch * pat)
PrintPatch2 (Patch * pat)
PrintPatch3 (Patch * pat)

PrintRegionList ()
PrintRegion (Region * reg)

DumpAllDataStructures ()

The print functions always leave the data structures unchanged. In some cases, the functions check for errors in the data structures and abort the linker if any errors are detected.

The source code for **link.c** contains a lot of print statements that have been commented out. These were used during debugging and they have been left in to aid future debugging. These print statements may help the reader, since some of them effectively serve as comments.

Additionally, for some error conditions, the code may call a print functions, which will additional useful information to be printed, before producing the error messages itself.

The function **DumpAllDataStructures** is invoked by the **-s** command line option, as well as some of the debugging options.

DumpAllDataStructures ()
    This function begins by renumbering the symbols, segments, and regions. Initially symbol and segment numbers are local to the input **.o** modules; after renumbering, every symbol and every segment will have a unique number, making the numbers meaningful to humans.

    This function then prints:
        A table with one line per module
        A table with one line per symbol
        A table with one line per symbol (grouped by segment)
        A table with one line per patch

> A table with one line per region
> A table with one line per segment

Before we print things, we renumber the everything, which is useful in the debugging printouts.

RenumberSymbolsSegmentsAndRegions ()
  Run through all symbols, segments, and regions. Re-assign identification numbers.

  Segment numbers will start at 1. Any and all dummy zero-filled segments will be numbered -1.

Recall that each input file numbered the symbols 1, 2, 3, …, the numbers were not unique; each file will have a symbol #1, etc. We abandon the numbers that were used in the original files. After reading in the input and creating the data structures, we identify **Symbols** by objects and pointers.

However, the numbers are very needed in the debugging printout.

This function also assigns a number to each segment (1, 2, 3, …) and a number to each region (1, 2, 3, …).

Initialization

Upon startup, the program calls a function named **CheckHostCompatibility** to make sure some basic assumptions (word size, byte-ordering, etc.) are met.

Next, some internal data structures are initialized.

The "**library index**" is a hash table that will map the exported symbols in a library to the modules in that library that exported them. It is initialized.

The "**export index**" is a hash table that will map the exported symbols from any module included in the output program to the internal representation for that symbol. It is initialized.

Next, the command line is processed by a function named **ProcessCommandLine**. If the "-h" (help) option is present, this function prints the help info and terminates the program. Some options are flags (either present or absent). For such options, we set the following variables to TRUE or FALSE:

commandOptionS	**-s**
commandOptionK	**-k**
commandOptionD1	**-d1**
commandOptionD2	**-d2**
commandOptionD3	**-d3**
commandOptionD4	**-d4**
commandOptionD5	**-d5**
commandOptionD6	**-d6**
commandOptionD7	**-d7**
commandOptionDsmall	**-small**
commandOptionW1	**-w1**
commandOptionW2	**-w2**
commandOptionW3	**-w3**
commandOptionW	**-w**

The following options are used only for debugging the linker. They result in printing additional information during the linking:

-d1	Print all data after files read in, before the main algorithm
-d2	Print all data after algorithm finishes placement and patches
-d3	Print a trace during segment placement (implies -d1 & -d2)
-d4	Print a trace during equate processing
-d5	Print a trace during the synthesizing of patches
-d6	Print a trace during region rounding
-d7	Print a trace during output file creation
-dsmall	Set memory size to 0x1,0000 = 4 pages

There must be exactly one output filename following "-o". This file is opened for writing as the variable **outputFile** (of type FILE *). There will be a number of input filenames. For each, we create an **InFile** data structure. Each **Infile** contains a "FILE *" and we open each input file and determine whether it is a library or an ordinary object file by reading its magic number.

The InFile Data Structure

There is a linked list of **InFile** structures, with one per input file. There is one **InFile** struct for every **.o** file and one for every **.lib** file.

The variable **firstInfile** points to the first and **lastInfile** points to the tail of the list. The name of the original file is retained for use in error messages. The filenames are also placed into the output file.

[qqqq Verify that the previous sentence is true. This code is not yet written. ????]

```
struct InFile {
  char *      filename;       // The name of an input file
  FILE *      filePtr;        // The input file
  int         isLibrary;      // 1 = this is a .lib file; 0 = .o file
  InFile *    next;           // Next pointer in linked list
};
```

Functions for Reading and Writing

There are a number of support functions used to read data from files. These functions are located in **BlitzSupport.c:**

ReadByte (FILE *) —> int
ReadInteger16 (FILE *) —> int
ReadInteger32 (FILE *) —> int
ReadInteger64 (FILE *) —> int65_t

We use this notation as shorthand to describe functions, along with their arguments and return values.

There are a number of functions used to write to the output file. These functions are located in **link.c**:

WriteInteger8 (int)
WriteInteger16 (int)
WriteInteger32 (int)
WriteInteger64 (int64_t)

Reading the Input Files

In the next step (in the **main** function), we run through the linker list of input files and read each file. The file is either a normal (simple) object file containing a single module, or it is a library file.

If the file is a simple object file, we create a **Module** structure and add it to the list of **Module** objects.

If the input file is a library, we create a single **Module** object for each module in the library. However, we do not add it to the linked list of **Modules**. The linked list is for modules that will definitely be included in the output file; at this stage we can not assume that any library module will be added to the output file.

Instead, we read through all the exported symbol names for a module. For each symbol, we add the name to the **Library Index**. The Library Index maps symbol names to **Module** objects. We call a function named **AddToLibraryIndex** to do this.

The Module Structure

One **Module** object is created for every input file. A library will contain one or more modules and one **Module** object will also be created for each module in the library.

```
struct Module {
  char *        moduleName;          // The name of the original .o file
  int           moduleNumber;        // Sequential number (assigned
                                     //        when created)
  char *        filename;            // The name of the input file
  FILE *        filePtr;             // The input file containing this module
  int64_t       startingLoc;         // Where in the file this module begins
                                     //        (after magic number)
  Module *      next;                // Next pointer in linked list
  char *        sourceFilename;      // The name of the original .s file
  int           numberOfSegments;    // Number of segments in this module
  int           numberOfSymbols;     // Number of symbols in this module
  Symbol * *    symbolArray;         // Ptr to an array of ptrs to symbol
                                     //                      objects
  Segment * *   segmentArray;        // Ptr to an array of ptrs to segment
                                     //                      objects
};
```

Each module has a name (such as "**Hello.o**") and this name is stored in the object. Each module come from a file: either a simple object file or from a library file. The **Module** object contains information (**filePtr**, **startingLoc**) about where the module can be found.

Module objects are kept in a linked list. The variable **firstModule** points to the head of this list and **lastModule** points to the tail. The field **next** is used for this linked list. This linked list is a list of all modules that will go into the output file.

Each module originated in an assembly language program "**.s**" file. The name of this file is retained as **sourceFilename**. The filename is used in printing error messages.

A module consists of a number of "segments". Recall that each segment was introduced in the assembly file with a "**.begin**" pseudo-op. The linker must process each segment (e.g., finding a place for it in memory) and, for each segment in the module, a **Segment** object will be created. The module's segments are pointed to by an array named **segmentArray**. We also maintain a field named **numberOfSegments** so we can run through the array in order.

The segments are numbered in order (1, 2, 3, …) starting with 1. To make the **segmentArray** indices match the segment numbers, the array will contain an extra unused entry for index 0. As a result, the size of the array is **numberOfSegments**+1.

Each module will define a number of symbols and for each one we will create a **Symbol** object. The module's symbols are pointed to by an array named **symbolArray**. We also maintain a field named **numberOfSymbols** so we can run through the array in order.

Just like the segment, the symbols are numbered in order (1, 2, 3, …) starting with 1. To make the **symbolArray** indices match the symbol numbers, the array will contain an extra unused entry for index 0. As a result, the size of the array is **numberOfSymbols** +1.

Hash Tables: Library Index and Exported Index

There are two dictionaries mapping string names to objects. One is called the "**Library Index**" and the other is called the "**Exported Index**".

Both mappings are implemented as hash tables that map string names into objects. The Library Index maps string names into **TableEntry** objects. The Exported Index maps string names into **Symbol** objects.

Both mappings are organized identically. Here we will discuss the organization of the Exported Index, but the Library Index is the same.

Each **Symbol** object contains a variable length string. The fields of relevance from the **Symbol** object are **stringLength** and **stringChars**. The **Symbol** object is described elsewhere and we will ignore the remaining fields in our description of the hash table.

We assume that string names may contain an arbitrary sequence of characters, possibly including embedded NULL \0 bytes, so we use a string length for the number of bytes in the name, rather than use the NULL-terminated scheme typically used in Unix/Linux.

The key functions are

```
void          AddToExportedIndex (Symbol * sym)
Symbol *      SearchExportedIndex (Symbol * sym)
```

To add an element to the mapping, we first create a new **Symbol** object and then call **AddToExportedIndex**. If there is already an entry in the mapping with the same symbol name, this function will print an error message.

Not all **Symbol** objects will be added to the mapping. In particular, we will only add symbols that have been exported to the mapping. In the case of imported symbols, we will have a **Symbol** object and we need to search the mapping to see if it contains another **Symbol** with the same name. This test is done with the **SearchExportedIndex** function.

To search the mapping, we take the name and compute a hash value from the bytes. This computation is performed by a function named **ComputeHash**. The **ComputeHash** function lives in **BlitzSupport.c** since it is used in other Blitz-64 programs.

The mapping is implemented as an array of pointers. Each pointer points to a linked list of **Symbol** objects. Each **Symbol** object contains a field named **exportedIndexNext**, which is used for this linked list.

To find an element, we compute the hash value and then use it (mod array size) as an index into the array. This gives us a pointer to a linked list. Then we perform a linear search on the linked list.

The array size is defined by this constant, such as:

```
#define HASH_TABLE_SIZE 4999
```

This number can safely be enlarged, but you should always use a prime number.

Assuming that a typical program uses 2,000 exported symbols, most linked lists will not be longer than one element. Thus, the first object we test is highly likely to be the match we are looking for. To handle larges programs with good performance, this constant has been increased to an even larger number.

The Library Index is similar, except that it maps string names into **TableEntry** objects.

```
struct TableEntry {
  TableEntry * next;                  // Linked list for each hash value
  Module *     exportedFromModule;    // The module that exported this symbol
  int          stringLength;          // Number of characters
  char         stringChars[0];        // The characters
};
```

Recall that a library file contains a number of modules and each module exports a number of symbols. Each library life begins with an index telling which symbols are exported and which module exported them. First, the linker must first read in all the library files and enter each exported symbol into the Library Index. Later, as the linker is building the output file, it may encounter an imported symbol. The linker will then search the library index to find the symbol. After retrieving a **TableEntry** object, the linker can determine (using the **exportedFromModule** field) which module from the library to add to the growing output file.

When adding symbols to the Library Index (in **AddToLibraryIndex**), we check to make sure that there is not already an entry there and print an error message if necessary.

The Library Index is built first, as the input files are processed and library files are encountered.

The Module List

At this point (within function **main**) we have already run though the input files. We have already built the **ModuleList**, adding one **Module** for each **.o** file and we built the Library Index as we encountered **.lib** files.

In the next step, we will enlarge the Module List so that it will contain all the modules that need to go into the output file.

Initially, the list contains only modules that came from **.o** files, but we may need to bring in additional modules from library files. Whenever a module imports a symbol that is otherwise undefined, we will search the Library Index looking for a module

that exported that symbol. If one is found, the corresponding module will be added to the Module List.

Otherwise if there is no entry in the Library Index, an error will be generated.

Reading the Modules: AddNewModule

A function named **AddNewModule** is called once for each module that will go into the output file. The **AddNewModule** function will go the file that contains the module (either a **.o** object file or a **.lib** library file) and will read in the header information describing that module. The function will add information to the growing data structures.

```
void AddNewModule (Module * mod)
```

The **Module** object contains information about which file contains the module and where in the file the module begins. This function begins by reading the header information (number of segments, number of symbols, source file name).

Each module will define a number of symbols. The **AddNewModule** function will allocate an array (**symbolArray**, in the **Module** object) with one element per symbol. For each symbol in the module, we will create and initialize a **Symbol** object. Furthermore, if the symbol is exported, this function will add the symbol to the Exported Index.

Each module will also contain a number of segments. The **AddNewModule** function will allocate an array (**segmentArray**, in the **Module** object) with one element per segment. For each segment in the module, this function will create and initialize a **Segment** object.

Each module will contain a number of patches. For each patch in the module, the **AddNewModule** function will create and initialize a **Patch** object. Every **Patch** object will be on exactly two linked lists. There is one linked list for each module and there is a global linked list of all patches.

The **AddNewModule** function will not read in the actual data bytes for the segment, since that information will not be needed until later, when we are ready to build the output file.

Before we continue describing the initialization algorithm in function **main**, we will describe the primary data structures used in the linker.

The Segment, Symbol, and Patch Objects

Next, we discuss the data structures that are used to represent the information contained in the modules that are to be linked together. Generally speaking, these data structures are allocated, set up, and initialized by the function **AddNewModule**.

```
struct Segment {
  Segment *      next;                // For the linked list of all segments
                                      //      in executable
  Segment *      nextForRegionList;   // For the linked list of all segments
                                      //      in a region
  Segment *      subListNext;         // For segmentList0, ...4, ...5, ...6, ...7
  Module *       myModule;            // The module from which this segment came
  int64_t        locationInFile;      // Location in file where segment data
                                      //    bytes are located
  int            segNumber;
  int            lineNumber;
  int64_t        initialLength;       // Size in bytes (as given in .o module)
  int            isKernel;
  int            isExecutable;
  int            isWritable;
  int            isZerofilled;
  int64_t        startAddr;
  int64_t        gpValue;
  Patch *        patchList;           // The patches that apply to this segment
  Patch *        patchListLast;       // .
  Symbol *       labelList;           // The labels that are in this segment
  int64_t        currentAddr;         // Where the segment is placed in memory
  int64_t        currentLength;       // How big is this segment, in bytes
                                      //      (may not be a multiple of 8)
  int64_t        paddingAdded;        // Number of bytes (0..7) added to bring
                                      //       segment size up to multiple of 8
};
```

```
struct Symbol {
  Module *      usedInModule;       // The module from whence this symbol came
  int           symbolNumber;       // The number of the symbol (1, 2, ...)
  int           lineNumber;         // Source file line number
  int           symbolType;         // 1=IMPORTED, 2=LABEL, 3= EQUATE
  Segment *     segment;            // Only for type 2 (LABEL)
  int64_t       offset;             // LABEL: offset from segment start;
                                    // EQUATE: offset from relativeTo
                                    //      symbol, or absolute value
                                    // IMPORT: unused (zero)
  int           relativeTo;         // Only for type 3 (EQUATE);
                                    //      0 means "absolute" &
                                    //      offset is the value
                                    // IMPORT, LABEL: unused (zero)
  int           exported;           // Only for type 2 (LABEL) and
                                    //      type 3 (EQUATE)
  Symbol *      target;             // Type 1/IMPORTED: ptr to exported symbol;
                                    //      Type 3/EQUATE: ptr to relativeTo
                                    //      or NULL.
  Symbol *      listNext;           // For the linked list of all
                                    //      symbols in executable
  Symbol *      exportedIndexNext;  // Linked list for each hash value
                                    //      in ExportedIndex
  Symbol *      nextForSegmentList; // There is also one linked list per
                                    //      segment (labels only)
  int64_t       currentValue;       // For LABELs: the address; for EQUATEs:
                                    //      the computed value
  int           markFlag;           // EQUATES only: 0 = not done yet;
                                    //      1=in progress;
                                    //      2=currentValue determined
  int           stringLength;       // Number of characters
  char          stringChars[0];     // The characters
};
```

```
struct Patch {
  Patch *      next;                // For the linked list of all patches
  Patch *      nextForSegmentList;  // There is also one linked list per segment
  int          patchType;          // 1,2,3, ...
  int          lineNumber;         // Source file line number
  Segment *    segment;            // Segment where this patch must be made
  int64_t      initialOffsetToPatch;// Offset into segment where patch
                                   //      must be made
  int          initialSize;        // Number of bytes present in .o file
                                   //      (0,4,8,12, or 16)
  Symbol *     targetSymbol;       // Target symbol (NULL = absolute)
  int64_t      offsetFromTarget;   // Offset from target symbol (often zero)
                                   // .  For patch type = ALIGN, offset
                                   //      will be 8,16,32,or 16384
  int          exactSize;          // Exact size of result in bytes
                                   //      (4, 8, 12, 16) or -1 if don't care
                                   // .  Only for Format S1,S2, ... S7
  int          sizeIncrement;      // The number of bytes to be inserted
                                   //      by the linker
  int64_t      currentOffsetToPatch;// Offset into segment where the patch
                                   //      will actually occur
};
```

Segment Objects

There is a linked list containing all the segments that will be placed in the output file. This list is pointed to by the global variable **segmentList**. The **next** field in a **Segment** object is used for this list.

Later, we will describe memory "regions". The region concept is used when placing segments in memory, i.e., when assigning addresses to segments. Main memory will be divided into a sequence of regions. Each region will have a single set of attributes (writable, executable). Each **Region** object will have a linked list of all the segments in it. The **nextForRegionList** field in **Segment** objects is used for this linked list. For now, this field is just initialized to NULL.

Later, we'll look at the attributes of a segment and add it to exactly one of the following linker lists:

> **segmentList0** Linked list of all fixed segments
> **segmentList4** Linked list of segments that are not Executable, not Writable
> **segmentList5** Linked list of segments that are not Executable,   Writable
> **segmentList6** Linked list of segments that are   Executable, not Writable
> **segmentList7** Linked list of segments that are   Executable,   Writable

Each segment came from a module. The **myModule** field points to this **Module** object.

Segments are numbered within a module. The **segNumber** field contains this number. Each segment is placed in the module's **segmentArray**; the array index and this field match. Patches refer to segments by number.

The object file contains the line number on which the segment began. The **lineNumber** field saves this information so it can be used in error reporting.

The bytes for the segment (the data and machine code bytes) are in the file and are not read at this time. The **initialLength** field tells how many bytes are in the file. The linker may increase the size of segments (as a result of inserting bytes when translating synthetic instructions or .align directives), so the segment size may grow. However, the **initialLength** field remains unchanged.

Each segment has these attributes: **isKernel**, **isExecutable**, **isWritiable**, **isZerofilled**, **startAddr**, and **gpValue**. These are read in and stored in the **Segment** object for later use.

Each module will contain a number of patches. Each patch applies to one segment, namely the segment containing the synthetic instruction or the .align pseudo-op. Each **Segment** contains a list of **Patch** objects. This list is pointed to by the fields **patchList** and **patchListLast**. As the patches are read in, they are added to the list for whichever segment they apply to (in addition to the global patch list). The patches are in order of increasing address, so the new **Patch** objects are added to the tails of the lists.

Of the symbols in a module, some are "labels", which identify locations within a particular segment. (Other symbols are "equates" and "imports".) The labels for a segment are kept in a linked list, and each **Segment** has a field named **labelList**

which will point to a linked list of **Symbol** objects. Each **Symbol** on this list will be a label in this segment. Within the Symbol objects, there is a field named **nextForSegmentList** which is used for this linked list.

Later in the linking algorithm, each segment will be assigned an address in memory. (Actually, the algorithm may try different addresses until it can fit everything in, so the segment may be moved around.) The **currentAddr** tells where this segment will be placed. At this stage, this field is merely initialized.

The linker may grow a segment, and the **currentLength** field tells the current size of the segment. At this stage, this field is merely initialized.

Symbol Objects

Each module contains a bunch of symbols. For each symbol, a **Symbol** object will be created. The **usedInModule** field (in the **Symbol** objects) points to the **Module** that contained this symbol.

Modules identify symbols by number. Each symbol in a given module is numbered (1, 2, 3, …). This number is used by patches and other symbols. The symbol number is kept in the field **symbolNumber**.

The **lineNumber** field tells where in the .s source code file the symbol was defined. The source code line number is used in sorting the labels within each segment, in addition to error reporting.

There are three different kinds of symbol: "Imported", "Label", and "Equate". The symbol type is identified but the field **symbolType** and corresponds to the way in which the symbol was defined in the .s source code file. A number is used for **symbolType**:

> 1 = imported
> 2 = label
> 3 = equate

Depending on what type of symbol it is, the following fields are used a bit differently.

For imported symbols…

symbolType	1 = "imported"
segment	not used
relativeTo	not used
offset	not used
exported	not used
target	ptr to matching symbol, which was exported

For label symbols…

symbolType	2 = "label"
segment	ptr to **Segment** in which this label occurs
relativeTo	not used
offset	offset into segment, in bytes
exported	1=exported; 0=not exported
target	not used

Equate symbols can either be "**absolute**" or "**relativeTo**". Initially, they can be distinguished by the **relativeTo** field. Subsequently, they are distinguished by the "**target**" field.

An "absolute" symbol looks like this…

symbolType	3 = "equate"
segment	not used
relativeTo	not used (zero)
offset	The value
exported	1=exported; 0=not exported
target	NULL

An "relativeTo" symbol looks like this…

symbolType	3 = "equate"
segment	not used
relativeTo	not used after initialization (a symbol number)
offset	An offset to be added in (often zero)
exported	1=exported; 0=not exported
target	A pointer to another symbol

There is a global list of all symbols, which is headed by **symbolList** and **symbolListLast**. The **next** field is used for this linked list.

A symbol can be exported. If the symbol is exported, then it will be added to the Exported List, so that it can be located. (We'll need to look symbols up in the index whenever we have an imported symbol, so we can link an imported symbol to its matching exported target.) The **exportedIndexNext** field is used in the hash table linked lists for the Exported Index. If the symbol is not exported, then this field will never get used.

There is a linked list of all labels that appear in a segment. This list is pointed to by the field **labelList** in the **Segment** object. If a **Symbol** is a label, then it will get added to the linked list for the segment in which it occurs. The field named **nextForSegmentList** is used for this purpose. If the **Symbol** is not a label, the **nextForSegmentList** field will remain unused.

We know the value of absolute symbols as soon as the segment is read in from a file, but the value of labels will only be known later in the linking, after the segment has been assigned an address in memory. And if one placement doesn't work, the segment will get moved to another memory address. Thus, the value of the symbol may change. The field **currentValue** is only used for labels and will be changed during the linking algorithm.

With equated symbols, the symbol is defined in terms of some other symbol, called the "relativeTo" symbol. The equate symbol can be defined as equal to the relativeTo symbol, in which case the **offset** will be zero. Or the **offset** can be non-zero, in which case we will need to add the **offset** to the value of the relativeTo symbol, once it is known.

At one point, we must determine the value of all equates. It is always possible that equates can be circularly defined. Cyclic definition is an error; we must be able to process the equates and assign a value to each. The field **markFlag** is used when assigning values to the equates.

Patch Objects

A "patch" indicates that the linker will need to modify the code generated by the assembler in some specific location in a segment. There are two reasons that the linker will need to perform patching.

The first type of patch is for a synthetic instruction which could not be translated by the assembler. This could happen when the assembler was unable to determine the target address for the instruction.

The second reason the linker must take action is for **.align** pseudo-ops. Since the assembler doesn't know where exactly the segments will be placed in memory, it is unable to know how many bytes to insert to achieve the required alignment. In addition, there may also be uncompleted synthetic instructions preceding the **.align** again making it impossible for the assembler to know how many bytes to insert to achieve the required alignment.

The will be one **Patch** object for each required patch and all **Patch** objects will be allocated in the **AddNewModule** function. Associated with each module is a list of patches that must be made to the segments in that module. For every module that will be included in the output file (i.e., for every module in a **.o** input object file and for every module pulled in from a library file), there will be a separate list of **Patches**. These lists will be built by **AddNewModule**.

Each **Patch** object will actually sit on two linked lists. First, there is a global linked list of all **Patch** objects. This list is pointed to by the global variables **patchList** and **patchListLast**, and each **Patch** object contains a **next** field for this global linked list.

In addition, each **Patch** applies to a particular location within some segment. Each segment contains its own list of **Patch** objects. Each **Segment** object contains fields called **patchList** and **patchListLast** which point to the head and tail of the segment's private list. Every **Patch** object will be on exactly one of these private lists.

These private per-segment patch lists are in non-decreasing order, by the offset that needs to be patched. Fortunately, the assembler will add the patches to the object files in order, so all the linker does is verify the correct ordering is followed.

Each **Patch** contains a field named **segment** which points to the **Segment** object for the segment within which the patch is to be made.

(Any **Patch** "p" will be on the linked list "p->segment->patchList" and any **Patch** on this list will point back to that same **Segment**.)

Each **Patch** contains a field named **patchType** which tells what sort of patch operation the linker is required to perform. One type is "alignment" and the remaining types are for the different types of synthetic instructions. The patch type is given by an integer in the range 1 ... 25. Patch type 24 is for "alignment patch".

Each **Patch** contains a field **lineNumber**, referring to the original **.s** assembly language file containing the synthetic instruction or **.align** causing the patch. The **lineNumber** is only used to print error messages.

Each **Patch** contains a field **initialOffsetToPatch**. This gives an offset in bytes from the beginning of the segment of the address that needs to be patched. This offset value comes from the **.o** object module and is not changed by the linker. However, the linker will be inserting bytes into segments as a result of other patches. Thus the actual offset may increase during the linker algorithm.

Each **Patch** contains a field **initialSize**. This contains the number of bytes already present in the segment prior to linking. There is also a field named **exactSize** which indicates whether the assembler has already determined the number of bytes required for the patch.

For alignment patches, there will be zero bytes initially present in the segment. The **initialSize** will be 0. The **exactSize** field will be set to -1, indicating that the linker is unconstrained and can insert as many bytes as it needs to.

For most synthetic instructions, the assembler will make no assumptions about how the linker will translate it to machine code. In these cases, the **initialSize** will be 4 to indicate that 4 bytes are initially present in the segment. The **exactSize** field will be set to -1, indicating that the linker is unconstrained and may insert additional bytes as necessary in translating the synthetic instruction.

However, for a few synthetic instructions, the assembler will have determined that there must be a certain number bytes in the translation. Although the assembler was unable to perform the translation itself, it may have relied on that being the size of the translation. In this case, both **initialSize** and **exactSize** will be equal and set to 4, 8, 12, or 16. There will be exactly that many bytes initially present in the segment. This exact size is not really necessary, but is included as a safety check (a program

logic check) to make sure that the linker does exactly what the assembler expected it to, and relied on.

For synthetic instructions, there will always be at least 4 bytes initially present in the segment. These 4 bytes will contain 4 register fields, in the normal bits for machine instructions for **Reg3**, **Reg2**, **Reg1**, and **RegD**. Some synthetic instructions have register fields and these will be included in the obvious way in thee 4-bit fields. The opcode bits (OP1 and OP2) will always be zero; the type of instruction can be determined from **patchType**.

If the initial version of the segment contains additional words (i.e., when **exactSize** is 8, 12, or 16), these additional words will be zeros. In other words, the second, third, and fourth words (if present) will be set to zero in the initial version of the segment.

Each **Patch** contains two fields named **targetSymbol** and **offsetFromTarget**. These specify the operand that requires linker intervention. The **targetSymbol** will be set to point to a **Symbol** object. The **offsetFromTarget** will be an integer and will often be zero. Later, when we determine the actual value of the target symbol, the offset will be added to give the final, effective value to be used in creating the machine code.

During the linking algorithm, the size allocated for a patch may be increased. For example, in an alignment patch, the linker may determine that 300 bytes must be inserted. As another example, in the case of an unconstrained synthetic instruction (i.e., where **initialSize** = 4 and **exactSize** = -1), the linker may determine that an additional word of machine code is necessary. The linker may grow an unconstrained patch by adding up to 3 words (to make the total 16 bytes).

As a result of growing patches and inserting bytes, the offsets (from the beginning of the segment) to everything that follows the patch will be shifted.

The **sizeIncrement** and **currentOffsetToPatch** fields will be used by the linker algorithm, but will be set to zero initially. As the algorithm progresses, **sizeIncrement** and **currentOffsetToPatch** will change. Both are in terms of bytes.

Processing Imported Symbols

Now, let's continue describing the algorithm in function **main** that initializes these data structures.

Prior to processing the imported symbols step, we ran through all the input **.o** files by going through the Module List and calling function **AddNewModule** for each module. This allocated and initialized the **Segment**, **Symbol**, and **Patch** object for each module that was explicitly named on the command line.

Whenever an imported symbol is not otherwise defined, but is defined by some module in some library, that library module must be added to the output file. The added module itself may import more symbols, which may themselves be undefined, causing additional modules to be pulled in from the library files.

We've already gone through the Module List in order to call AddNewModule for each explicitly mentioned Module.

Now, in order to pull in the necessary library modules, we go through the Module List a second time, from beginning to end, looking at all imported symbols. During this process, we may add additional modules to the end of the Module List. In particular, whenever we determine that another module from a library is needed, we'll add a new **Module** to the tail of the Module List.

Whenever we add a new library module to the Module List, the function **AddNewModule** must be called to allocate additional **Segment**, **Symbol**, and **Patch** objects. Since newly added modules are placed at the tail end of the Module List, every newly added module will get processed later on as we encounter it when going through the Module List. Thus, its imported symbols will eventually be examined, perhaps pulling in yet more modules. (Obviously, this process will terminate since we only have a finite number of modules that can be added to the list.)

For each module, we'll run through all the symbols in that module, looking only at symbols of type "imported". For each imported symbol, we'll locate a matching symbol (i.e., same spelling) that is exported. First, we check the Exported Index to see if there is a matching symbol that has already been exported. If found, then we can move on. Otherwise if there is no matching entry, we must search the Library Index. If we find a match there, then we will pull in the module; otherwise we print

an error ("Undefined symbol"). As each module is processed by **AddNewModule**, the symbols it exports will be added to the Exported Index.

After pulling in all the modules, we make a second pass through the global list of **Symbols** and link every imported symbol with the corresponding exported symbol. In particular, we make the imported symbol's **target** field point to the exported symbol.

Next, we run through the global symbol list a second time. This time, we look at the **relativeTo** field. If a **Symbol's relativeTo** field points to an imported symbol, we will modify the symbol to point directly to the exported symbol.

Next, we run through the global list of **Patch** objects. If the **Patch** object's **targetSymbol** points to an imported symbol, then we modify the patch to point directly to the exported symbol.

Sorting the Label and Segment Lists

Every symbol that is a label belongs to exactly one segment. In other words, each label is intended to identify an address within some segment.

Each **Segment** object contains a linked list of all the labels that occur within it. Each **Segment** object has a field named **labelList** which points to a linked list of **Symbols**, which are linked using a field named **nextForSegmentList**.

The assembler places all symbols in the **.o** file in a random order. (The symbols come out of a hash table and the order is a byproduct of the hashing, so... it's effectively random.)

In the next step in function **main**, we sort each segment's label list to get them into the order they appeared in the original source code. This is necessary because, as we go through a segment and process the patches, we will be inserting bytes here and there. As we pass by labels, we will need to update them as well, to reflect the new addresses they will represent.

You might guess that we sort the labels on initial offset into the segment. However, it is possible that there can be more than one label for a single offset.

Consider this example:

```
label1:
     .align   16
label2:
```

We perform the sort before we invoke **PlaceSegment**, which means that the ALIGN patches all have zero length. Thus, the offsets for **label1**, **label2**, and the ALIGN will all be identical. But we need to process them in the correct order, since the ALIGN will expand to several bytes, making the resulting offsets for **label1** and **label2** different.

We know that only one label can occur per source code line so, instead of sorting on offset, we sort on source code line number.

SortLabelLists ()
> This function looks at the label list for each segment and sorts it. The sort is based line number into the segment. Actually, because ALIGNs have length zero, it is better to sort on line number. This keeps things in the proper order.

There are a couple of additional helper functions that do the actual sorting:

quicksortLabelList (int m, int n)
partitionLabelList (int left, int right) —> int

Segment Ordering

Generally speaking, a good way to pack "things" into an available space, is to try to fit the largest things in first, and proceed in order from largest to smallest. (Imagine packing several suitcases into the trunk of a car or furniture into a moving truck. You want to put the largest items in first.)

For floating segments (where the programmer has not said explicitly where to place the segment), the linker takes this approach when placing segments in memory: It looks at the floating segments in order, from largest to smallest.

The problem of packing segments into memory in an optimal way must, I think, be NP-complete; but trying to place larger segments before we try to place the smaller segments should yield acceptable results.

Before now, we have a single, global list of segments. In this step, we will partition the set of segments into 5 lists, which we call:

> **segmentList0** All fixed segments
> **segmentList4** Floating segments that are <u>not</u> Executable & <u>not</u> Writable
> **segmentList5** Floating segments that are <u>not</u> Executable & Writable
> **segmentList6** Floating segments that are Executable & <u>not</u> Writable
> **segmentList7** Floating segments that are Executable & Writable

The segments on all lists will be ordered from largest to smallest.

We do this by first sorting the list of all segments. Then, we run through it an place each segment on exactly one of the sublists.

The actual lengths of the segments will change over the course of the linker algorithm. (The segment's **currentLength** will change, but **initialLength** will remain unchanged). Since the lists are not ordered by the actual length but by initial length, the order may not be exactly perfect, but since the order of the segments is unlikely to change significantly as segment sizes are adjusted, this approach should lead to fairly good packing of segments into the available spaces.

OrganizeSegmentLists ()
  This function sorts the global list of all segments from smallest to largest, based on **initialLength**. Then, it builds all the individual segment lists, ordered from largest to smallest.

There are a couple of additional helper functions that do the actual sorting: **QuicksortSegmentArray** and **PartitionSegmentList**.

[We don't actually care about the order of the fixed segments. Since their locations are determined by the programmer, it really doesn't matter what order we look at each one. And you might have noticed that it is inefficient to sort one big list. We don't need to sort the fixed segments at all and it would be more efficient to sort the four small lists separately. Technically, this is accurate, but… (1) We do not expect to see many fixed segments; (2) We expect most segments to be either executable and not writable (for code and constants), or not executable and writable (for data), so we really have only two significant lists; and (3) We just don't expect to see a huge

number of segments. The time to quick-sort even a few hundred segments is still small.]

Regions and Placing Segments

All of main memory will be represented within the linker and **Region** objects will be used to represent memory regions.

```
struct Region {
  Region *      prev;           // Doubly linked list, ordered by address
  Region *      next;           // .
  Segment *     segmentList;    // List of segments in this region
  int64_t       address;        // Starting address of this region
  int64_t       length;         // Number of bytes (not necessarily a
                                //    multiple of anything)
  int           regionStatus;   // -1 = header; 0 = free; 4/5/6/7 = allocated
  int           regionNumber;   // For printing only
};
```

Each region has a starting address and a length in bytes. The fields named **address** and **length** describe the region's location and size.

Region Invariants

- Every byte of memory belongs to exactly one region.
- The regions are kept in an ordered list.
- All regions are contiguous.
- The address of the first byte of a region directly follows the address of the last byte of the previous region.
- Each region is either free or allocated.
- A free region contains no segments.
- An allocated region contains one or more segments.
- The segments in a region occupy exactly the bytes within that region.

Region objects are organized in a doubly linked list. The fields **next** and **prev** are used for this purpose.

The list is organized as a circular ring. There is a dummy header object that is inserted into the ring. Unlike all other **Region** objects, the dummy header object

does not represent a range of memory addresses. The dummy header is inserted after the last **Region** and before the first **Region**.

Each region has a status given by the field named **regionStatus**. These codes are used:

-1   Dummy header
0   Unallocated, i.e., free
4   Allocated, Not Executable, Not Writable
5   Allocated, Not Executable, Writable
6   Allocated, Executable, Not Writable
7   Allocated, Executable, Writable

Memory regions that are "free" are available for use. Initially, the region data structure contains only a single region which contains all memory bytes. (And the dummy header region exists, as well.)

The linker algorithm will place segments in memory. Whenever a segment is placed in memory, the region data structure will be modified. Bytes will be removed from a free region and added to the allocated region that will contain the segment.

Segments have memory attributes (executable, writable). When a segment is placed in memory, the pages in that region will need to be marked by the OS kernel with the correct (executable, writable) attributes. So when a region of memory is allocated, it will be allocated with some particular set of attributes.

Each region has a list of the segments that are in that region. When a segment is placed into a region, it will be added to that region's segment list. The field **segmentList** points to the linked list of **Segment** objects. Within **Segment** objects, the field **nextForRegionList** is used for this inked list. Unallocated regions will have **segmentList** == NULL.

Region objects are numbered with a field named **regionNumber**. This field is only used for printing to distinguish **Region** objects. There is a global variable named **nextRegionNumber** which is used to assign increasing numbers whenever a new region is created.

Initially, main memory will be divided into two regions:

- Dummy Header
- Free Region (covering all of usable memory)

The following constants are initialized during program startup based on the **-k** command line option:

	Kernel (-k)	**User Programs**
START_OF_MAIN_MEMORY	0x0_0000_0000	0x8_0000_0000
SIZE_OF_MAIN_MEMORY	0x8_0000_0000	0x8_0000_0000
HIGHEST_MAIN_MEMORY_ADDR	0x7_FFFF_FFFF	0xF_FFFF_FFFF

(There is also a **-dsmall** option which will reduce memory size to 4 pages, which is useful for debugging and testing boundary cases.)

During the linking algorithm as segments are placed in memory, a new region may be created and "carved out" of an existing free region.

The main linker algorithm repeatedly loops, looking for a solution to the segment placement problem. Whenever the algorithm iterates, it needs to start over. At the start of each new iteration, all memory regions will be freed and the **Region** data structure will be completely re-initialized.

MemoryReset ()
 This function creates the initial circular ring of two **Region** objects. Upon subsequent calls, it frees any previously allocated **Region** objects, as well.

As mentioned above, each region points to a linked list of segments that are in that region. During **MemoryReset**, we also go through the segments and re-initialize their "next" pointers, effectively removing them from the regions.

PrintRegionList ()
 This function prints a table showing all the regions. It also contains a call to **CheckRegionConsistency**.

PrintRegion (region)
> This function prints a single line describing a single region, including the
> numbers of the segments in that region.

CheckRegionConsistency ()
> This function runs through the region data structure, performing a number of
> consistency checks. It is only invoked from **PrintRegionList**.

In a normal use of the linker **CheckRegionConsistency** will not be invoked. If there
are problems or bugs, any run of the linker will almost certainly involve a command
line option that will print the region list and thus invoke **CheckRegionConsistency**.

The programmer can specify exactly where a segment is to be placed, using the
"**startaddr=**" on a "**.begin**" statement. A segment for which the programmer has
given a starting address is called a "fixed" segment and must be placed at the exact
address the programmer has specified. A segment without a starting address is
called a "floating" segment. The linker will determine the address of floating
segments and place them wherever it determines is a good place.

CreateNewRegion (freeRegion, address, segment) —> ptr to new region
> This function places a segment in memory. It creates a new **Region** object, places
> the segment in it, and returns a pointer to the new region. The function takes a
> free region as input, along with the segment that is being placed in memory and
> the address where the segment is to be placed. The free region is guaranteed to
> contain all the memory addresses that will be needed to place this segment at
> this address, but the free region may contain additional bytes as well.

This function will create a new **Region** object and place it in the ring data structure.
The new region may be identical in size and location to the given free region, in
which case the new **Region** will entirely replace the free **Region**. Or the new **Region**
may leave remnant free regions. There may be a shortened free **Region** before the
new **Region** and/or there may be shortened free **Region** after the new **Region**.

Note that after calling this function, it is possible that the original free region has
been entirely replaced by an allocated region and that this are region may have
exactly the same (executable/writable) attributes as the region before or after it.
These regions must be merged, but that is the responsibility of the caller; it is not
done by this function.

MergeWithNeighbors (region)

    This function is passed a newly allocated **Region**, previously created by function **CreateNewRegion**. It is possible that this newly allocated region has the exact same attributes (executable, writable) as the region directly before or after it. In such cases, the two regions are merged into one larger region. This function uses a helper function called **MergeTwoRegions**, first to deal with the region before the candidate region and second to deal with the region after the candidate region.

MergeTwoRegions (region1, region2)

    This function will merge these two regions into one region if and only if they have the same **regionStatus** codes. Whenever two regions are merged, the first remains and the second disappears. All segments in the second region are moved to the **segmentList** of the first region and the second region object is freed.

RegionsShareAPage (firstRegion, secondRegion) —> bool

    This function tests to see if two regions happen to share a page. The regions may not be adjacent, but first region is assumed to come before the second region. This function determines the page number of the last byte of the first region and the first byte of the second region and asks whether they are on the same page.

If two segments with different (executable, writable) attributes are placed in adjacent regions that happen to share a page, then an error must be reported.

If we are linking a user program, then we need to enforce the rule that two segments may not share a page unless that have the same (executable/writable) attributes. The next function does this.

RegionsInConflict (region, otherRegion) —> bool

    This function determines whether these two regions have conflicting attributes. For example, if one region is "executable, writable" and the other region is "executable, not writable", there is a conflict. Free regions never conflict, since they can take on any attributes.

The above function is only used by the next function.

CheckAndMergeNewRegion (region)
  The function is called directly after a new region has been created to contain a some segment. This function calls **RegionsShareAPage** to see if this new region shares pages with any other nearby regions and **RegionsInConflict** to check if are conflicts. If there is a conflict, then it prints an error unconditionally. Then this function calls **MergeWithNeighbors**.

This function calls a helper function named **FixedSegmentAttributeConflict** to print an error.

FixedSegmentAttributeConflict (region, region2)
  This function unconditionally prints "***** ERROR: These segments have different (executable, writable) attributes but try to occupy the same page. *****". It also prints additional information to augment the error message.

SegmentSharesPageWithRegion (segment, region) —> bool
  This function is passed a segment and a region. If this segment has any pages in common with the memory area in the region, this function returns true.

SegmentStatusConflictWithRegion (segment, region) —> bool
  This function is passed a segment and a region. Presumably they share a page, but this is not checked. Instead, it returns true iff they have (Executable/ Writable) attributes that are in conflict.

ThereIsAnAttributeConflict (segment, freeRegion) —> bool
  This function is passed a segment and a free region into which we are considering placing the segment. This function looks at the regions that precede the free region and the regions that follow the free region. It determines whether this segment shares a page with any allocated regions that have different (executable, writable) attributes. For user programs, every page will must have a unique set of attributes, so this is not acceptable placement of this segment.

If we are linking a kernel program, we don't care about conflicts and this function returns immediately. Otherwise, it begins by examining all regions that follow the free region, until it comes to a region that does not share a page with the segment. Then it examines all regions that precede the free region, again halting when it

comes to a region that does not share a page with the segment. This function calls **SegmentSharesPageWithRegion** and **SegmentStatusConflictWithRegion** to get the job done.

FindFreeRegionForFixedSegment (segment) —> region
   This function goes through the region list looking for the free region in which this fixed segment is to be placed. If none can be found, an error message is printed.

The above function can print the following errors, all of which are "fatal" and will immediately terminate the linker:

- The starting address of first segment overlaps some other fixed segment
- The ending address of first segment overlaps some other fixed segment or some unusable memory region
- The starting address of this segment is not within 0x0_0000_0000 ... 0x7_FFFF_FFFF, yet command option -k requires this
- The starting address of this segment is not within 0x8_0000_0000 ... 0xF_FFFF_FFFF. (For kernel code, use the -k option)

ComputeRegionStatus (segment) —> regionStatus
   A little helper function that returns the region status code number for this segment:
   4   Not Executable, Not Writable
   5   Not Executable, Writable
   6   Executable, Not Writable
   7   Executable, Writable

IsLegalAddress (integer) —> bool
   A little helper function that tests whether this integer is a legal address. By legal, we mean that it is any value within
   0_0000_0000 ... 7_FFFF_FFFF   if -k was used
   8_0000_0000 ... F_FFFF_FFFF   otherwise

PlaceSegment (segment, newAddress) —> size
   This function is passed a segment and the address where this segment is to be placed in memory. This function assumes that the patches have already been adjusted and it will not modify the sizes of the patches (except ALIGN patches).

This function will compute the size of this segment. It will also examine all the labels in the segment and (knowing where the segment is getting placed), it will set their values. This function will also determine the address that each patch for this segment is supposed to modify and will set that.

Now that we have addresses for the bytes within a segment, we can determine how many bytes to insert for an ALIGN patch. This function will determine the sizes of the ALIGN patches.

Given: An address (where to put the segment)
Returns: The new segment size in bytes

This function will take the patches, with their sizes as currently configured. It will not adjust patch sizes (other than from ALIGN patches).

For the segment...
    Set **currentAddr** and **currentLength**.
For every ALIGN patch...
    Set **currentOffsetToPatch**.
    Determine what size is needed & set **sizeIncrement**.
For all other patches...
    Use the current value of **sizeIncrement**.
    Set **currentOffsetToPatch**.
For every LABEL...
    Set its **currentValue** to an absolute address.

This function will go through the segment from top to bottom. It will only look at the segment's labels and patches, not the actual data.

The above function uses a fairly complex algorithm. Associated with each segment are two lists: **patchList** and **labelList**. These have previously been sorted. The function starts at the beginning go the segment and goes through it linearly. It doesn't actually look at the data bytes; these won't even be read in from the file until later when we are building the output file.

As this function goes through the segment, it is inserting bytes. Or, more precisely, it is figuring out how many bytes need to be inserted and computing how that shifts everything down in memory and makes the segment larger.

The main loop goes through the label list and the patch list simultaneously. For each iteration, it takes whatever comes next in the file. This is either a label or a patch.

The loop is keeping track of how many bytes have been inserted so far. When a label is encountered, it can use this information (**bytesAdded**) to determine the actual value of the label.

When it encounters an ALIGN patch, it can determine the current address and determine how many bytes to insert to give the proper alignment. And it also increments **bytesAdded** accordingly.

When it encounters any other kind of patch, it looks at the patch (in particular at the patch's **sizeIncrement** field) to determine how many bytes this patch has grown beyond what was originally in the segment. Again, it will increment **bytesAdded** accordingly.

Also, for all kinds of patches, it will make a note of exactly where in the segment this patch is now located, by setting the patch's **currentOffsetToPatch** field.

This function is "idempotent", which means that it can be called repeatedly with no adverse effects. If you don't like where the segment was placed, you can call this function again to put it somewhere else. As the main algorithm iterates, the segments will be moved around to different locations.

PlaceOneFloatingSegment (segment) —> freeRegion
   This function is passed a segment. It finds a location where this segment can be legally placed. It searches the region list and looks at all free regions. This function returns the free region that contains the segment's starting address.

   This function does not modify the Region data structure. However, this function calls **PlaceSegment** to place the segment at some address, which will modify the segment and set **segment->currentAddr**.

   If no location can be found to place this segment, this function causes a FatalError, which will abort the linker.

Since a floating segment can be placed anywhere and since memory is quite large, it is hard to envision a scenario where this function fails to find a place to put this segment. So the likelihood of getting this error message is small.

PlaceFloatingSegments (segmentList)
This function is passed a list of segments. It will go through the list and, for each segment, it will locate a place in memory where this segment can be placed. It will place the segment there and modify the region data structure.

This function runs though all the segments in the list. Some segments may have already been placed. For example, all fixed segments will have been placed previously. Also some segments may have size zero; these segments will not go into memory and we just ignore them.

For each segment, this function calls **PlaceOneFloatingSegment** to find a location for the segment. Then it calls **CreateNewRegion** to put the segment into a new region. Finally it calls **CheckAndMergeNewRegion** to merge the region with its neighbors. The function **CheckAndMergeNewRegion** will see if there are conflicts with nearby allocated regions, but this should never occur, since **PlaceOneFloatingSegment** will only find legal places to put a floating segment.

PlaceAllSegments ()
This function is called to assign a memory address to every segment and build the Region data structure, which will reflect how memory is used.

For each segment, this function will set the segment's...
    currentAddr
    currentLength

For every ALIGN patch, this function will...
    Set **currentOffsetToPatch**.
    Determine what size is needed and set **sizeIncrement**.

For all other patches, it will...
    Use the current value of **sizeIncrement**.
    Set **currentOffsetToPatch**.

For every LABEL, it will...
    Set its **currentValue** to an absolute address.

The above function will take the patches as they are currently configured. In other words, it will not evaluate the patches, modify them, or see if they are workable. After all, we can only compute or check the patches after the segments have been placed in memory, since we can't assign values to labels until after the segment placements have been made. We'll look at the patches later on.

The algorithm used to place the segments in memory is this:
- First, place all fixed segments at their locations.
- Then try to fill in gaps keeping segments with similar attributes together.
- Finally, place any remaining segments wherever we possibly can.

The **PlaceAllSegments** function begins by calling **MemoryReset** to allocate and initialize the Region data structure. Next, it marks all segments as "unplaced".

Then for each fixed segment, it calls…
> **PlaceSegment**
> **FindFreeRegionForFixedSegment**
> **CreateNewRegion**
> **CheckAndMergeNewRegion**

Next, **PlaceAllSegments** will look at each free region and try to fill it with floating segments that have the same (executable/writable) attributes as the previous region.

Previously, we have created separate lists. The lists are called:

segmentList0	All fixed segments
segmentList4	Floating segments that are <u>not</u> Executable & <u>not</u> Writable
segmentList5	Floating segments that are <u>not</u> Executable & Writable
segmentList6	Floating segments that are Executable & <u>not</u> Writable
segmentList7	Floating segments that are Executable & Writable

For example, imagine we have an "executable/not-writable" fixed segment followed by a free region. If there are other segments that are also "executable/not-writable", we'd like to place them in this free region. Perhaps by packing all the "executable/not-writable" segments close together, we can reduce the number of pages that must be marked "executable/not-writable".

In this step, the function searches for any free region preceded by an allocated region. For example, assume it finds a free region preceded by an "executable/not-writable" region. It chooses the correct segment list, e.g., **segmentList6**.

Then we run though that list, attempting to place those segments into this free region. To do that, this function calls a function named **TryToPlaceTheseSegmentsAfterThisRegion**.

Finally, we simply place the remaining floating segments anywhere we can fit them.

Do this this, PlaceAllSegments will call PlaceFloatingSegments four times, once for each list of floating segments. We will process the lists in this order:

> **segmentList5** Floating segments that are <u>not</u> Executable & Writable
> **segmentList4** Floating segments that are <u>not</u> Executable & <u>not</u> Writable
> **segmentList6** Floating segments that are Executable & <u>not</u> Writable
> **segmentList7** Floating segments that are Executable & Writable

The idea is that we are guessing that the segments that contain variables will be marked not executable and writable. We want this to go in low memory (0x0 for kernel or 0x8,0000,0000 for user programs, so that gp-relative addressing will work well. Then we follow it segments that are not executable and not writable; which we assume is read-only data; again we expect placement in low memory will tend to facilitate gp-relative addressing. Then we follow it with code, which is executable and not writable.

TryToPlaceTheseSegmentsAfterThisRegion (segmentList, region)
> This function is passed a list of segments, all of which have the same attributes (executable, writable). It is also passed a region, which is followed by a free region.
>
> We run through the segment list looking at each unplaced segment in turn. We attempt to place each such segment at the beginning of the free space.

The function **TryToPlaceTheseSegmentsAfterThisRegion** is passed a list of floating segments, all of whose attributes match the attributes of the region. The region is followed by a free region, at least when it is called.

The function runs through the list of segments and tries to pack them into the free region. The segments are sorted from largest to smallest, so it attempts to fill the free region with the largest first, followed by smaller segments.

Some segments may already have been placed; these are ignored. Otherwise, we call **PlaceSegment** to update the segment as if it has been placed. This will determine the segment's size. Then we check to see if it will actually fit in the space available. We also have to make sure that placing this segment here will not cause a conflict due to a shared page with a subsequent region. This is done by calling **ThereIsAnAttributeConflict**.

If everything looks good, this function creates a new region and places the segment into it, by calling **CreateNewRegion**. Then it calls **MergeWithNeighbors** to merge this region with the original region. It is also possible that the newly created region completely eliminated the free region and we can merge the new region with the following region.

On the other hand (if the free region was not large enough or there were attributes conflicts), the segment is not placed and we move on to the next segment (toward smaller segments) to see if it will fit.

Once all the segments have been placed in memory, every LABEL symbol will have been assigned an address. Now we can compute the value of all EQUATE symbols. Symbols of type EQUATE were defined with an **.equ** pseudo-op.

We no longer care about symbols of type IMPORT, since all references to a symbol defined with a **.import** pseudo-op have been replaced by references to an exported symbol, which necessarily must have been defined either as a LABEL or EQUATE symbol.

ResolveEquates ()
    This function runs through all symbols and, for every symbol of type EQUATE, computes and fills in its "**currentValue**" field.

    This function uses a marking algorithm, utilizing the "**markFlag**" field in symbols.
        0 = "*not done yet*"
        1 = "*in progress*"
        2 = "*done*" (**currentValue** has been determined)

First, the **ResolveEquates** function runs through all symbols and marks all EQUATE symbols as "*not done yet*". It marks all LABELS and IMPORT symbols as "*done*".

Then it runs through all symbols again, and for each EQUATE symbol, calls function **ResolveOneEquate**.

ResolveOneEquate (symbol)
   This is a recursive algorithm that computes the value of the given symbol.

The ResolveOneEquate function returns immediately if the symbol is marked as "*done*". If the symbol is already marked as "*in progress*", we have detected a cyclic definition, so we print an error and return.

If the symbol is an absolute value, then we can immediate set its value. We mark it "*done*" and return.

Otherwise, this symbol is defined as relative to some other symbol. We should take the value of the other symbol and add the given offset to it.

In order to get the value of the other symbol, we will call ourself recursively. So we set the **markFlag** to "*in progress*" and recursively call **ResolveOneEquate** on the relative-to symbol.

Upon return, we change the **markFlag** to "*done*", retrieve the value of the relative-to symbol, add the offset to it to determine this symbol's new **currentValue**, and return.

The Main Linker Algorithm

Now we have all the functions we need — the functions previously described. We are ready to give the algorithm.

```
   REPEAT until no more failures
      Place all fixed segments.
      Place all floating segments.
         (Placing segments will set "currentValue" for all labels)
      Resolve all equates.
         Recursive Algorithm: Set a flag to check for cycles.
            Initialize the flag to 0.
            0 = not done yet; 1=in progress; 2=final value determined
   Check all patches.
      Determine what machine code the patch translates to.
      If any patch is too big to fit its allocated space
         Increase the "sizeIncrement" of the patch
         FAILED = true
   END REPEAT
```

The placement of all fixed and floating segments is done within function **PlaceAllSegments**. Equates are resolved within function **ResolveEquates**. And the patches are checked within function **CheckAllPatches**.

So the above algorithm looks more-or-less like this in the code:

```
failureOccurred = 1;
while (failureOccurred) {
   failureOccurred = 0;
   PlaceAllSegments ();
   ResolveEquates ();
   CheckAllPatches ();
}
```

In order to understand this, think about the patches within segments. Each patch has an initial size. If, during the algorithm, that size is determined to be too small for the machine instructions that must be used, the patch size will be increased. This will constitute a "failure".

The placement of segments is done without modifying the patch sizes, with one exception exception: the ALIGN patches. The ALIGN patches are processed at the time a segment in placed at a specific address. (We can only perform the alignment after we know the actual addresses.)

Placing the segments has the side-effect of assigning an address to each label.

After the segments have been placed and the labels have been assigned addresses, we can process all the equates.

Once we have processed the equates, we have values of all symbols and we know where the patches are in memory.

Then, we can run through the patches. Each patch has a certain number of bytes allocated to it. The function **CheckAllPatches** makes sure that the machine instructions for each patch will fit into the bytes we have reserved for it. If there is a problem (i.e., the machine code for a synthetic instruction will not fit into the space we have reserved for the patch), then **CheckAllPatches** will determine how many bytes are needed to increase the reservation.

If **CheckAllPatches** ever determines that some patch would not fit into the space reserved for it, it will set the global variable **failureOccurred** and the algorithm will loop.

The size of a patch is given by two fields in the **Patch** object: **initialSize** and **sizeIncrement**. **CheckAllPatches** may increase the **sizeIncrement** and, if so, it will set **failureOccurred**.

If, however, there is adequate room reserved for every patch, then **CheckAllPatches** will complete and the repeat loop will terminate.

CheckAllPatches ()
   This function runs through all the patches and makes sure that there is adequate room in the segment for the patch.

   If we encounter a patch that will not fit in the allotted space, we set **failureOccurred** to TRUE and we increase **patch->sizeIncrement** to indicate how many bytes are required.

   For registers, we are using dummy values. The actual synthesized instructions are ignored.

The function **CheckAllPatches** simply runs through the global list of patches and, for each, calls **ProcessOnePatch**. Patches of type ALIGN are ignored, since they are processed in function **PlaceSegment**.

ProcessOnePatch (patch, finalRun)

    This function will process a single patch, creating the translation of a synthetic instruction. It will place the resulting machine code translation into these variables:

        word1 — 1st instruction word
        word2 — 2nd instruction word
        word3 — 3rd instruction word
        word4 — 4th instruction word

    It will use as many of these as necessary, placing NOPs in the remaining words.

    This function assumes that **word1** will initially contain the registers to use in fields Reg1, Reg2, Reg3, and RegD.

    During the main algorithm, the registers in **word1** will be zero and don't matter. During the final run when we are actually putting the bytes into the segment data, the registers in **word1** will be valid.

    This function will modify **patch->sizeIncrement**, increasing it as necessary.

    The number of bytes actually used is **initialSize + sizeIncrement**.

    If **sizeIncrement** was increased, this function will set **failureOccurred** to TRUE. Otherwise, **failureOccurred** will not be modified.

    If **finalRun** is true, this function will assume that **sizeIncrement** was correct and will produce a **ProgramLogicError** if not.

    Errors may be detected. They will be ignored, unless **finalRun** is TRUE, in which case they will be printed. The only user error detected is "offset out of range" for the LOADx-offset and STOREx-offset instructions.

The function **ProcessOnePatch** is lengthy.

We should make one note. Normally, the translation of a synthetic instruction does not depend on the values of Reg1, Reg2, Reg3, or RegD. There is one exception, namely the MOVI instruction. If the destination register in **gp (r13)** then the synthetic instruction may be translated differently.

To deal with this, there are actually two patch types for MOVI:

    patchType == 1:   MOVI (RegD ≠ gp)
    patchType == 25:   MOVI (RegD = gp)

For this reason, this function needs to know what is in the file. So, in this case, the function will read a word from the file at the site of the patch to get the register values to see if the destination register is, in fact, **gp** (**r13**).

Finalization

After the loop terminates, we call function **PerformRegionRounding**. This function only has any effect if we are linking a user program.

The executable will be organized into pages. The function **PerformRegionRounding** will enlarge each region to become an integral multiple of pages. It does this by creating some "dummy" zero-filled segments which it adds to regions as necessary.

PerformRegionRounding ()
    This function is called after all segments have been placed and the regions have been created. It rounds all regions to be an even multiple of pages and makes sure each region starts on a page boundary. It does this by taking bytes out of the free regions before and after a region.

    This function will also creates dummy "zero-filled" segments whenever the bytes in a page are not filled with a real segment. In other words, when bytes are move from a free region to an allocated region, a new zero-filled segment will be created and added to the allocated region. Later, when we are writing the allocated regions out to the executable file, these new zero-filled regions will be included, making sure that all bytes in the regions are either initialized with bytes or zero-filled.

    When linking a kernel program, this function does nothing.

CreateZerofilledSegment (region, startAddr, lengthInBytes)
> This function creates a dummy segment that is zero-filled and adds it to the given region.
>
> Such a segment is required when the linker places two or more segments in a single page but when there is a gap between them. These bytes must be zero-ed at load time. There is also a dummy module that will be created. This module will contain all the dummy segments.
>
> The module will NOT be placed on the module list, so it will not print out. However, if errors occur, the module will be needed for printing.
>
> The newly created dummy segment will be placed on the global segment list, but will not be placed on any of the segment sublists. The new segment will be placed on the region's segment list.

The program also checks to make sure there is a symbol named "_entry".

Finally, the program writes out the executable file. Given the data structures we have built up to this point, this part is straightforward.

Finally, we print out the data structures (by invoking **DumpAllDataStructures**) if the **-s** command line option was specified, then print counts of error messages and warnings and terminate.

Acronym List

CSR   Control and Status Register
EOL   End of line
ISA   Instruction Set Architecture (the core design)
KPL   Kernel Programming Language
LC   Location Counter
LSB   Least Significant Bit / Byte
MSB   Most Significant Bit / Byte
PC   Program Counter
UTF-8   An encoding for Unicode (Unicode Transformation Format)

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s, which has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. His technical focus and research interests have included AI and neural networks; parsing and natural language processing; logic, object-oriented, and functional programming; compilers, operating systems, interpreters, and system software; and discrete math and computational theory. He has programmed in many high-level languages and written assembly code for a variety of machines, dating back to the IBM 360/67 and Intel 8080.

Porter lives in Portland, Oregon. When not trying to figure out how his computer actually works, he skis, hikes, travels, and spends time with his children building things.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.