# CONVEX Compiler Utilities User's Guide

*First Edition*

CONVEX

CONVEX COMPUTER CORPORATION

**Convex Computer Corporation**
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

# CONVEX Compiler Utilities User's Guide

Order No. DSW-096

First Edition
October 1991

CONVEX Press
Richardson, Texas
United States of America

# CONVEX Compiler Utilities User's Guide

Order No. DSW-096

Copyright © 1991 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. All rights reserved. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions, or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE EQUIPMENT DESCRIBED HEREIN IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS EQUIPMENT. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
CONVEX C100 Series and C200 Series are trademarks of CONVEX Computer Corporation.
ConvexOS is a trademark of CONVEX Computer Corporation.
C1, C120, C210, C220, C230, and C240 are trademarks of CONVEX Computer Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Printed in the United States of America

## Revision Information for

# CONVEX Compiler Utilities User's Guide

| Edition | Document No. | Description |
|---|---|---|
| First | 720-005130-000 | This document combines the information that existed in two previous documents: *CONVEX Loader User's Guide* and *CONVEX Assembly-Language User's Guide*. New material in this document consists of:<br>• New loader options: `-A` and `-p`.<br>• New assembler options: `-ada`, `-cvtdc3`, `-n`, and `-tm`.<br>• Information on loading C programs.<br>• Information on loading FORTRAN programs.<br>• Descriptions of thread-private directives: `.tcbss` and `.tcdata`.<br>• Description of the `psum` directive. |

# Contents


## Using this book . . . . . . . . . . . . . . . . . . . xv

Purpose and audience . . . . . . . . . . . . . . . . . . xv
Organization . . . . . . . . . . . . . . . . . . . . . . xvi
Notational conventions . . . . . . . . . . . . . . . . . xvii
General conventions . . . . . . . . . . . . . . . . . . xvii
Associated documents . . . . . . . . . . . . . . . . . . xviii
Ordering documentation . . . . . . . . . . . . . . . . . xix
Technical assistance . . . . . . . . . . . . . . . . . . xix

## 1 Loader overview . . . . . . . . . . . . . . . . . . . 1

Loader input . . . . . . . . . . . . . . . . . . . . . . 2
Loader output . . . . . . . . . . . . . . . . . . . . . . 3
Loader functions . . . . . . . . . . . . . . . . . . . . 4
  Linking object files . . . . . . . . . . . . . . . . . 4
  Resolving internal references . . . . . . . . . . . . 4
  Resolving external references . . . . . . . . . . . . 5
  Iterative loading . . . . . . . . . . . . . . . . . . 6

## 2 `ld` command . . . . . . . . . . . . . . . . . . . . 9

Introduction . . . . . . . . . . . . . . . . . . . . . . 9
Command format . . . . . . . . . . . . . . . . . . . . . 10
Loader processing . . . . . . . . . . . . . . . . . . . 11
Loader options . . . . . . . . . . . . . . . . . . . . . 11
Reprocessing executable files . . . . . . . . . . . . . 16
Loading C programs . . . . . . . . . . . . . . . . . . . 17
Loading FORTRAN programs . . . . . . . . . . . . . . . . 18

## 3 Object file format . . . . . . . . . . . . . . . . . 21

Header format . . . . . . . . . . . . . . . . . . . . . 21
Text segment . . . . . . . . . . . . . . . . . . . . . . 23
Data segment . . . . . . . . . . . . . . . . . . . . . . 23
Relocation entries . . . . . . . . . . . . . . . . . . . 24
Symbol table . . . . . . . . . . . . . . . . . . . . . . 25
String table . . . . . . . . . . . . . . . . . . . . . . 28

## 4 Standard object file format (SOFF) . . . . . . . . . . 29

File header . . . . . . . . . . . . . . . . . . . . . 29
Optional header . . . . . . . . . . . . . . . . . . . 32
Segment headers . . . . . . . . . . . . . . . . . . . 35
Predefined segments . . . . . . . . . . . . . . . . 36
Initialized common blocks . . . . . . . . . . . . . 36
Segment header layout . . . . . . . . . . . . . . . 37
Segment data . . . . . . . . . . . . . . . . . . . . 40
Relocation information . . . . . . . . . . . . . . . 41
Symbol table . . . . . . . . . . . . . . . . . . . . 43
`n_value` field . . . . . . . . . . . . . . . . . . . 46
Handling initialized common blocks (ICBs) . . . . . . . 47
Initialization data format . . . . . . . . . . . . . 48
String table . . . . . . . . . . . . . . . . . . . . 49

## 5 Object libraries . . . . . . . . . . . . . . . . . . 51

Standard system libraries . . . . . . . . . . . . . . 51
Auxiliary programs . . . . . . . . . . . . . . . . . 53
`ar` . . . . . . . . . . . . . . . . . . . . . . . . 53
Creating libraries . . . . . . . . . . . . . . . . 55
Adding new modules . . . . . . . . . . . . . . . . 56
Updating modules . . . . . . . . . . . . . . . . . 56
Deleting modules . . . . . . . . . . . . . . . . . 57
Extracting modules . . . . . . . . . . . . . . . . 57
Listing module names . . . . . . . . . . . . . . . 57
`ranlib` . . . . . . . . . . . . . . . . . . . . . . 59
`sod` . . . . . . . . . . . . . . . . . . . . . . . 59

## 6 Loader examples . . . . . . . . . . . . . . . . . . 63

## 7 Assembler introduction . . . . . . . . . . . . . . . 65

Utilities . . . . . . . . . . . . . . . . . . . . . . 65
Assembly-language debugger . . . . . . . . . . . . . 65
`make` utility . . . . . . . . . . . . . . . . . . . 65
`rcs` utility . . . . . . . . . . . . . . . . . . . . 66
`error` utility . . . . . . . . . . . . . . . . . . . 66
Compilers . . . . . . . . . . . . . . . . . . . . . . 66
Loader . . . . . . . . . . . . . . . . . . . . . . . . 66
Optimum use of the assembler . . . . . . . . . . . . . 66

## 8 Architecture overview . . . . . . . . . . . . . . . . . 67

CONVEX architecture . . . . . . . . . . . . . . . . . . . 67
Parallel processing . . . . . . . . . . . . . . . . . . . 67
CONVEX register sets . . . . . . . . . . . . . . . . . . . 68
  Address registers . . . . . . . . . . . . . . . . . . . 68
  Scalar registers . . . . . . . . . . . . . . . . . . . 69
  Vector registers . . . . . . . . . . . . . . . . . . . 69
  Communication registers . . . . . . . . . . . . . . . . 69
    Communication register addressing . . . . . . . . . 70
    Hardware lock bit . . . . . . . . . . . . . . . . . 70
  Processor status word (PSW) . . . . . . . . . . . . . . 70

## 9 Assembler instruction formats . . . . . . . . . . . . 73

Instruction format . . . . . . . . . . . . . . . . . . . 73
Label field . . . . . . . . . . . . . . . . . . . . . . 74
  Name labels . . . . . . . . . . . . . . . . . . . . . 74
  Temporary labels . . . . . . . . . . . . . . . . . . . 75
Operation field . . . . . . . . . . . . . . . . . . . . 76
  Instruction mnemonics . . . . . . . . . . . . . . . . 76
  Assembler directives . . . . . . . . . . . . . . . . . 76
Operand field . . . . . . . . . . . . . . . . . . . . . 76
Comment field . . . . . . . . . . . . . . . . . . . . . 77
Terminator field . . . . . . . . . . . . . . . . . . . . 77

## 10 Op codes . . . . . . . . . . . . . . . . . . . . . . . 79

Instruction typing . . . . . . . . . . . . . . . . . . . 80
Extended op codes . . . . . . . . . . . . . . . . . . . 80
Machine op codes . . . . . . . . . . . . . . . . . . . . 81
  Memory-reference instructions . . . . . . . . . . . . 81
  Effective address calculation . . . . . . . . . . . . 82
  Memory-reference instruction format . . . . . . . . . 83
  Arithmetic instructions . . . . . . . . . . . . . . . 85
  Logical instructions . . . . . . . . . . . . . . . . . 86
  Data-type specification . . . . . . . . . . . . . . . 87
  Data-conversion instructions . . . . . . . . . . . . . 88
  Vector-reduction instructions . . . . . . . . . . . . 89
  Operations under mask . . . . . . . . . . . . . . . . 89
  Program-control instructions . . . . . . . . . . . . . 90
  Span-independent program-control instructions . . . . 91
  Machine-control instructions . . . . . . . . . . . . . 92
  Synchronization instructions . . . . . . . . . . . . . 93
    `lck` and `ulk` . . . . . . . . . . . . . . . . . . 93
    `snd` and `rcv` . . . . . . . . . . . . . . . . . . 94

CPU control instructions . . . . . . . . . . . . . . . . . 95
`pfork`, `wfork`, and `cfork` . . . . . . . . . . . . . . 96
`spawn` and `join` . . . . . . . . . . . . . . . . . . . 96
Pseudo operations (directives) . . . . . . . . . . . . . . . 97
Program-segment directives . . . . . . . . . . . . . . . 98
Storage directives . . . . . . . . . . . . . . . . . . . .101
Uninitialized storage allocation . . . . . . . . . . . .101
Initialized storage allocation . . . . . . . . . . . . .101
Strings . . . . . . . . . . . . . . . . . . . . . . . .103
Alignment directives . . . . . . . . . . . . . . . . . . .104
Floating-point directives . . . . . . . . . . . . . . . . .105
External symbolic-name directives . . . . . . . . . . . .107
`.globl` directive . . . . . . . . . . . . . . . . . . .107
`.comm` directive . . . . . . . . . . . . . . . . . . .108
Thread-private directives . . . . . . . . . . . . . . . .108
Symbol-table directives . . . . . . . . . . . . . . . . .109
Procedure summary directives . . . . . . . . . . . . . .110

## 11 Operands . . . . . . . . . . . . . . . . . . . . . . 113

Assembler character set . . . . . . . . . . . . . . . . . .113
Operators . . . . . . . . . . . . . . . . . . . . . . . . .114
Terms . . . . . . . . . . . . . . . . . . . . . . . . . . .115
Numeric constants . . . . . . . . . . . . . . . . . . . .115
Character constants . . . . . . . . . . . . . . . . . . .116
Symbolic names . . . . . . . . . . . . . . . . . . . . . .116
Expressions . . . . . . . . . . . . . . . . . . . . . . . .117
Relocatable expressions . . . . . . . . . . . . . . . . .117
Absolute expressions . . . . . . . . . . . . . . . . . . .118
External expressions . . . . . . . . . . . . . . . . . . .118
Type propagation in expressions . . . . . . . . . . . . .119
Symbolic name assignment statements . . . . . . . . . . . .120
Location counter . . . . . . . . . . . . . . . . . . . . .120
Addressing modes . . . . . . . . . . . . . . . . . . . . . .122
Register mode . . . . . . . . . . . . . . . . . . . . . .122
Immediate addressing mode . . . . . . . . . . . . . . . .126
Memory-addressing modes . . . . . . . . . . . . . . . . .128
Absolute-addressing mode . . . . . . . . . . . . . . . .128
Register-deferred mode . . . . . . . . . . . . . . . . .129
Indexed mode . . . . . . . . . . . . . . . . . . . . . .129
Indirect-absolute mode . . . . . . . . . . . . . . . . .130
Indirect-deferred mode . . . . . . . . . . . . . . . . .130
Indirect-indexed mode . . . . . . . . . . . . . . . . . .131

## 12 Calling conventions . . . . . . . . . . . . . . . . . 133

Special address registers . . . . . . . . . . . . . . . . .133
  Stack pointer . . . . . . . . . . . . . . . . . . . . .133
  Argument pointer . . . . . . . . . . . . . . . . . . . .134
  Frame pointer . . . . . . . . . . . . . . . . . . . . .134
  Compiler-generated code . . . . . . . . . . . . . . . . .134
Linkages . . . . . . . . . . . . . . . . . . . . . . . . .135
  `callq` . . . . . . . . . . . . . . . . . . . . . . . .135
  `calls` . . . . . . . . . . . . . . . . . . . . . . . .135
  `call` . . . . . . . . . . . . . . . . . . . . . . . . .136
General calling conventions . . . . . . . . . . . . . . . .136
  Function or subroutine stack layout . . . . . . . . . . .136
  Function or subroutine calling sequences . . . . . . . .137
    Sequence 1 . . . . . . . . . . . . . . . . . . . . . .137
    Sequence 2 . . . . . . . . . . . . . . . . . . . . . .138
C calling conventions . . . . . . . . . . . . . . . . . . .140
  Code generated for function calls . . . . . . . . . . . .140
  Function names . . . . . . . . . . . . . . . . . . . . .141
  Function arguments and return values . . . . . . . . . .141
FORTRAN calling conventions . . . . . . . . . . . . . . . .142
  Code generated for function calls . . . . . . . . . . . .142
  Subprogram names . . . . . . . . . . . . . . . . . . . .142
  Function arguments and return values . . . . . . . . . .143
  Subroutine arguments and return values . . . . . . . . .144

## 13 Using the assembler . . . . . . . . . . . . . . . . . 145

Invoking the assembler . . . . . . . . . . . . . . . . . .145
Error messages . . . . . . . . . . . . . . . . . . . . . .148

## 14 Sample programs . . . . . . . . . . . . . . . . . . . 149

Code to copy block of memory . . . . . . . . . . . . . . .149
Vector routine . . . . . . . . . . . . . . . . . . . . . .154
Sample parallel program . . . . . . . . . . . . . . . . . .159

## 15 Advanced topics . . . . . . . . . . . . . . . . . . . 167

Coding techniques . . . . . . . . . . . . . . . . . . . . .167
  High-level language processors . . . . . . . . . . . . .168
  Coding hints . . . . . . . . . . . . . . . . . . . . . .168
Using macros and the preprocessor . . . . . . . . . . . . .169
Optimizing performance . . . . . . . . . . . . . . . . . .169
  Optimizing scalar code . . . . . . . . . . . . . . . . .170
  Optimizing vector code . . . . . . . . . . . . . . . . .170
Debugging with `adb`, `csd`, and `cxdb` . . . . . . . . . .171

Parallel programming in CONVEX assembly language . . 171
Processes and threads . . . . . . . . . . . . . . . . . 172
Creating threads . . . . . . . . . . . . . . . . . . . 172
Processor scheduling . . . . . . . . . . . . . . . . . 172
Special registers . . . . . . . . . . . . . . . . . . 173
Ending threads . . . . . . . . . . . . . . . . . . . . 173
Communicating between threads . . . . . . . . . . . 174
Thread memory management . . . . . . . . . . . . . . 174
Shared memory . . . . . . . . . . . . . . . . . . . . 174
Unshared memory . . . . . . . . . . . . . . . . . . . 175
Thread data segments and thread stacks . . . . . . . 175
Further reference . . . . . . . . . . . . . . . . . . . . 176

**A Loader error messages . . . . . . . . . . . . . . . 177**

**Glossary . . . . . . . . . . . . . . . . . . . . . . . 181**

# Figures


Figure 1 Loader functions ........ 2
Figure 2 Generating library files ........ 3
Figure 3 Resolving internal references ........ 5
Figure 4 Header section format (from <a.out.h>) ........ 22
Figure 5 Format of a relocation entry (from <a.out.h>) ........ 24
Figure 6 Symbol table entry format (from <nlist.h>) ........ 26
Figure 7 File header layout (from <convex/filehdr.h>) ... 30
Figure 8 Optional header layout (<convex/opthdr.h>) ........ 33
Figure 9 Segment header layout (<convex/scnhdr.h>) ........ 37
Figure 10 Relocation entry format (<convex/reloc.h>) ........ 41
Figure 11 Relocatable address references ........ 42
Figure 12 Symbol table entry format (from <nlist.h>) ........ 44
Figure 13 ICB symbol table interconnections ........ 48
Figure 14 ICB raw data layout ........ 49
Figure 15 Processor status word—CONVEX C1 Series ........ 71
Figure 16 Processor status word—CONVEX C2 and C3 Series 71
Figure 17 Instruction format example ........ 73
Figure 18 Name labels example ........ 75
Figure 19 Temporary labels example ........ 75
Figure 20 Memory longword structure ........ 83
Figure 21 Memory-reference instruction format ........ 83
Figure 22 Use of program segments ........ 100
Figure 23 Examples of the ds directive ........ 102
Figure 24 Example of the fpmode directive ........ 106
Figure 25 Top of the runtime stack ........ 137
Figure 26 Stack layout ........ 139
Figure 27 Calling a C function ........ 141
Figure 28 Calling a FORTRAN subroutine ........ 142
Figure 29 Example of bcopy ........ 150
Figure 30 Vector routine ........ 155
Figure 31 Parallel program ........ 160

# Tables


Table 1 Reprocessing file options ........ 17
Table 2 ntype flags (from <nlist.h>) ........ 27
Table 3 File header flags ........ 32
Table 4 Optional header flags ........ 34
Table 5 Predefined segments ........ 36
Table 6 Protection flags ........ 39
Table 7 Segment header flags ........ 40
Table 8 n_type flags ........ 45
Table 9 Memory-reference instructions ........ 82
Table 10 Arithmetic instructions ........ 85
Table 11 Logical instructions ........ 86
Table 12 Data-type specifiers ........ 87
Table 13 Data-conversion instructions ........ 88
Table 14 Vector-reduction instructions ........ 89
Table 15 Program-control instructions ........ 90
Table 16 Span-independent program-control instructions ........ 92
Table 17 Machine-control instructions ........ 92
Table 18 Synchronization instructions ........ 95
Table 19 CPU control instructions ........ 97
Table 20 Program-segment directives ........ 98
Table 21 Assembler escape sequences ........ 103
Table 22 Binary and unary operators ........ 114
Table 23 Assembler escape sequences ........ 116
Table 24 Location-counter-directive sequences ........ 121
Table 25 Register names ........ 123
Table 26 Machine-control registers ........ 125
Table 27 Immediate operands ........ 126
Table 28 Complex function and FORTRAN subroutine ........ 143
Table 29 Character-valued function equivalent ........ 143

# Using this book

## Purpose and audience

The *CONVEX Compiler Utilities User's Guide* provides general information about the loader and assembler utilities. This book discusses the following topics:

- Pertinent facts about the CONVEX loader, including the `ld` command and its options
- How to load C and FORTRAN programs manually
- Object file format used for versions of the ConvexOS operating system prior to V6.1
- Standard object file format (SOFF) used with V6.1 of the ConvexOS and later
- Object libraries
- Pertinent facts about the CONVEX hardware architecture
- Assembler instruction formats, op codes, and operations
- Conventions specific to writing assembly-language programs on CONVEX supercomputers
- General procedures for using the assembler
- Detailed examples of working assembly-language programs
- Information helpful to advanced programming

This book addresses experienced programmers who wish to use the loader or assembler for specific applications.

## Organization

The fundamental information for using the loader is in Chapters 1 through 6. Information that pertains to the use of the assembler is primarily in Chapters 7 through 15. This book is organized into the following chapters and appendixes:

- Chapter 1 provides a conceptual overview of loader operations and functions.
- Chapter 2 describes the format of the loader command `ld` and describes options and file arguments used with the command.
- Chapter 3 describes the object file format for systems running V6.0, or earlier, of the operating system.
- Chapter 4 describes the standard object file format (SOFF) for systems running V6.1, or later, of the operating system.
- Chapter 5 provides information on location and contents of standard object file libraries.
- Chapter 6 provides programming examples.
- Chapter 7 presents an overview of the CONVEX assembler, its operating environment, and support tools.
- Chapter 8 describes fundamental information about CONVEX supercomputer architecture as it relates to assembly-language programming. This chapter includes information on addressing modes, memory addressing, the register set, operands, and the processor status word.
- Chapter 9 describes the command-line format and details the specific options of an assembly statement.
- Chapter 10 discusses machine operation codes, presents an overview of the instruction set, and describes pseudo-operations and assembler directives.
- Chapter 11 discusses operands, including symbolic names, expressions, and addressing modes.
- Chapter 12 includes the calling conventions for the CONVEX assembly language, a brief explanation of linkages and normal register use, and special conventions for C, FORTRAN, and the math library.
- Chapter 13 tells how to invoke the assembler and describes error messages.

- Chapter 14 includes detailed examples of working assembly-language programs, including a block-copy, vector, and parallel program.
- Chapter 15 discusses optimizing performance, debugging, using macros and the preprocessor, and writing parallel programs.
- Appendix A provides descriptions of some ambiguous loader error messages.
- A Glossary is also provided

## Notational conventions

This section discusses notational conventions used in this book.

### Command syntax

Consider this example:

COMMAND *input_file* [...] {a | b} [*output_file*]

where:

- COMMAND must be typed as it appears.
- *input_file* indicates a file name that must be supplied by the user.
- The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
- Either a or b must be supplied.
- [*output_file*] indicates an optional file name.

## General conventions

In general, this book uses the following conventions:

- **`Bold constant-width font`** identifies user input in examples.
- *Italic* type face
  - Designates user-supplied variables in a command-line example
  - Introduces new and important terms
  - Indicates document titles

- `Constant-width font` designates input and output, including:
  - Command names and options
  - System calls
  - Directives, program statements, machine op codes, printout examples, file names, directories, and error messages returned
- Horizontal ellipsis (...) shows a repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.
- References to man pages appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

## Note

**A Note highlights supplemental information.**

## Caution

**A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.**

## Associated documents

Using this book successfully may require information not specific to the tasks described herein or not within the scope of this guide. The following documents may answer questions to help you solve problems encountered when working through this book. CONVEX Computer Corporation provides the following related documents:

- *ConvexOS Man Pages for Users* (DSW-331), *ConvexOS Man Pages for Programmers* (DSW-332), and *ConvexOS Man Pages for System Managers* (DSW-333). These guides document the ConvexOS operating system.
- *CONVEX C Series Architecture Reference Manual* (DHW-300). This manual describes the architecture and instruction set used by the CONVEX computer system.
- *CONVEX C Series Assembly Language Reference Manual* (DHW-301). This manual describes all the instructions in the CONVEX C1, C2, and C3 Series computers.
- *CONVEX C Guide* (DSW-086). This guide describes the CONVEX C compiler.

- *CONVEX FORTRAN Language Reference Manual* (DSW-037). This reference manual defines the CONVEX extensions to the FORTRAN-77 standard.
- *CONVEX FORTRAN User's Guide* (DSW-038). This guide describes how to compile, link, debug, and execute FORTRAN programs.
- *CONVEX CXdb Concepts* (DSW-471) explains debugging concepts that might be required to understand the CONVEX visual debugger.
- *CONVEX Application Compiler User's Guide* (DSW-401). This manual describes how to use the application compiler. One of its chapters describes the `psum` directives, which can be used with assembly language programs.

## Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

## Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC). Use the phone numbers in the following table.

| Location | Phone number |
|---|---|
| Within the continental U.S. | 1(800)952-0379 |
| Outside continental U.S. | Contact local CONVEX office |

# Loader overview

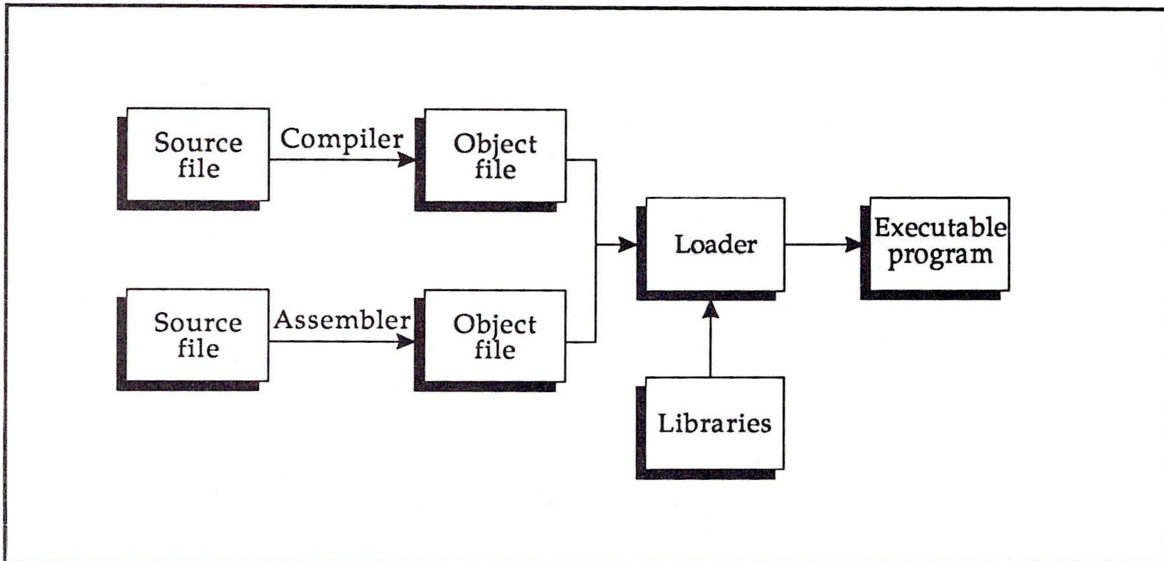
# 1

In general, a loader is a program that prepares the output of language processors for execution. The CONVEX loader, `ld`, performs this function by:

- Combining separately compiled object modules to produce executable programs
- Resolving symbolic references between modules
- Searching libraries of previously compiled object modules to resolve symbolic references

Loaders enable you to use separately compiled subroutines and object archives that house libraries of previously compiled subroutines. These capabilities support modular programming and enable you to use the machine more efficiently. Loaders are sometimes called *linkers* or *link editors* because they resolve external reference linkage between separately compiled modules. Figure 1 illustrates the basic functions of the loader.

**Figure 1**
Loader functions




The CONVEX loader does not load an executable program into main memory; programs are loaded into main memory when they are executed.

## Loader input

Only two types of files, object files and library files, are used as input to `ld`. An object file is a single object module that contains binary instructions and data. Library files contain frequently used object modules that the loader inserts into programs.

Each object file produced by the loader contains a symbol table. The `/usr/include/nlist.h` file defines the layout of the symbol table entries. Symbols and the symbol table for the object file format used by the operating systems through V6.0 are discussed in Chapter 3; Chapter 4 discusses the standard object file format (SOFF) produced by operating systems V6.1 and later.

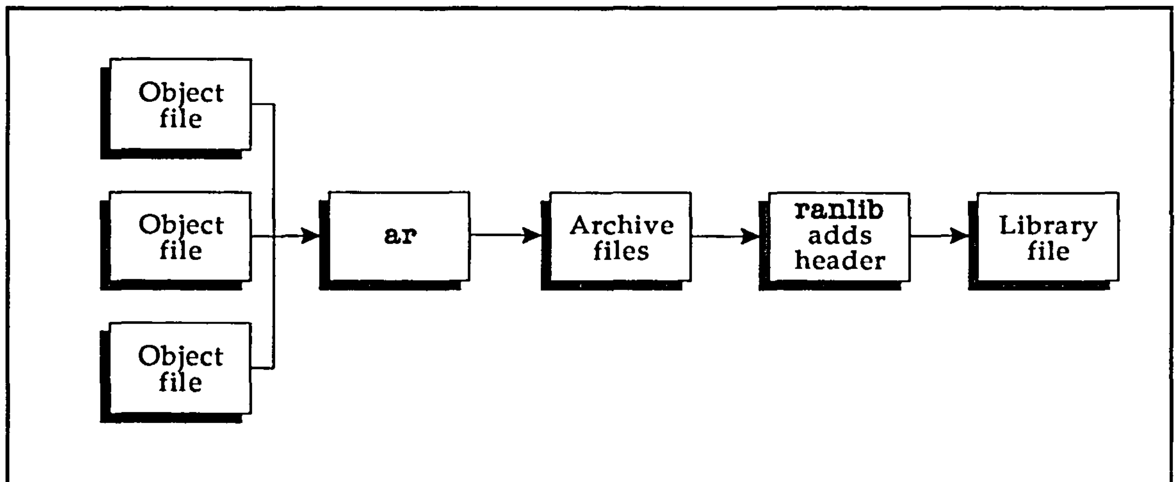
To produce library files, follow these steps:

1. Merge object files into archive files with the `ar` program.

2. Create a library header for the archive file using the `ranlib` program; `ranlib` adds header information to the beginning of the archive files, which enables the loader to quickly locate symbols in the file.

Figure 2 illustrates this procedure. Object-file libraries are also discussed in Chapter 5.

**Figure 2**
Generating library files




## Loader output

Object files that `ld` produces have the same internal format as loader input object files. For `ld`, the standard output is a file called `a.out`. The loader makes this file executable (by setting the file's execute-permission bits) if there are no errors or unresolved references in the input. Chapter 3 discusses the format of the object files produced by operating systems through V6.0. Chapter 4 discusses the standard object file format (SOFF) produced by operating systems V6.1 and later.

## Loader functions

This section provides an overview of the basic loader functions. These functions include:

- Linking object files
- Resolving internal references
- Resolving external references
- Iterative loading

### Linking object files

The four stages in the linking process are:

1. The loader resolves address references in the program.
2. The loader resolves any references made in the file to symbolic names in other object files or library files.
3. The loader creates an executable program file (unless you specify the `-r` option).
4. The loader produces an optional "load map" of symbols and references and reports unresolved external references and other errors.

Not all instructions in the CONVEX assembly-language instruction set run on every machine in the CONVEX family of supercomputers. Each assembly-language instruction is tagged within the assembler to be of a certain type. This *type* corresponds to the machine(s) that will accept the instruction. The types of all instructions in one assembled file are stored in the flags-word of the executable file's header. The loader accumulates all of the flags-words from each object file and stores them in the flags-word in the final executable file. No loader option to ignore these instruction types currently exists.

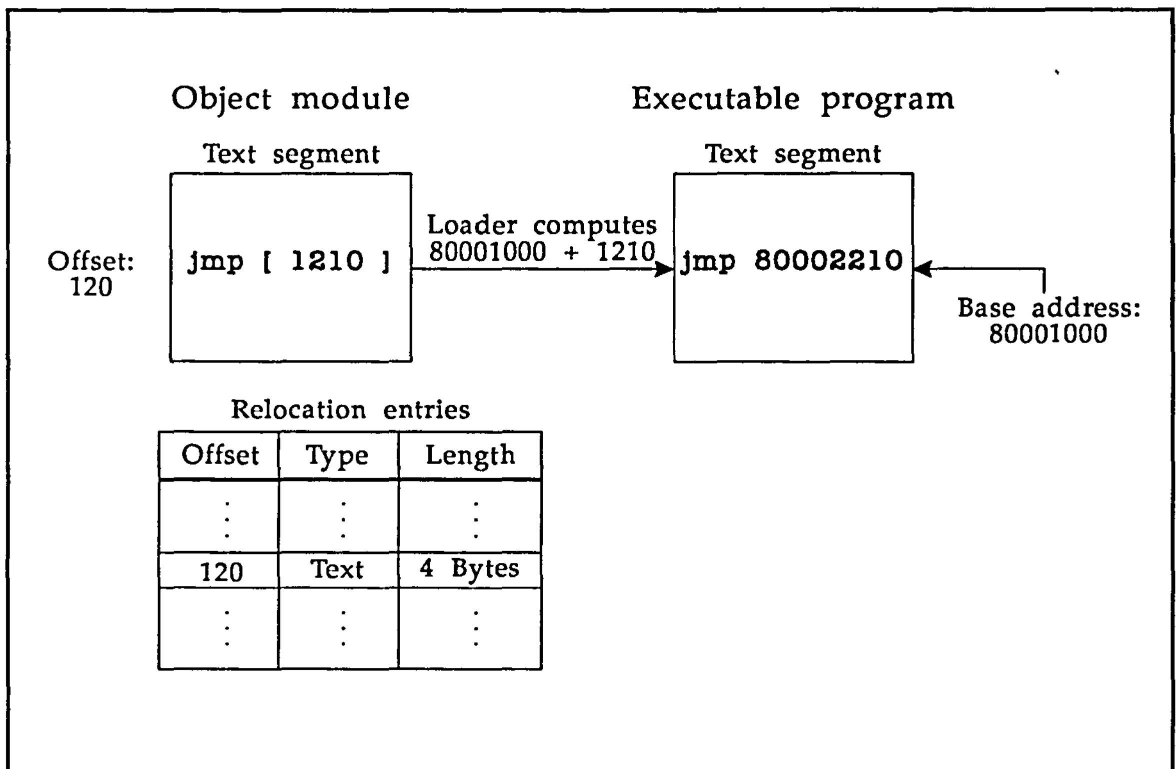
### Resolving internal references

The loader translates addresses of any relocatable data within object modules into absolute addresses in main memory for the generated executable program. Within each object module, the loader corrects machine instructions that reference main memory by adding a bias factor to account for the final location of the referenced data within main memory.

Within the object module, the assembler generates memory references as offsets within the .text, .data, .tdata, .bss, or .tbss segments of the program. The assembler also adds a special relocation record to the generated object file that instructs the loader to resolve that address by adding the base address of the appropriate segment.

The loader must fill in the actual address of the data. It does this by adding the base address of the appropriate segment (.text, .data, .tdata, .bss or .tbss) to the offset stored in the instruction and corrects the instruction in the generated executable program to reference the correct final address. Figure 3 illustrates this action.

**Figure 3**
Resolving internal references




| Offset | Type | Length |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 120 | Text | 4 Bytes |
| ⋮ | ⋮ | ⋮ |

## Resolving external references

The loader makes two passes through the object files and libraries specified on the ld command line to resolve external references and perform library searches. In the first pass, the loader looks at the symbol table of each object file to find

symbols that resolve references. If a symbol remains unresolved after this procedure, the loader looks in the header of subsequent libraries to find a symbol in that library that resolves the reference.

In the second pass, the loader "patches" locations containing symbol references with physical symbol addresses. As a second step, the loader copies from the library any object modules that were "marked" on the first pass as resolving references. The loader also generates an output file.

## Iterative loading

The loader can be used to combine small object files into a larger object file using a technique called iterative loading. To perform iterative loading, use the `-r` option on the command line.

Iterative loading minimizes overhead associated with reading and processing many smaller object modules. You can combine stable routines into a larger object file and leave undeveloped or changing routines accessible. Decreasing the number of object files can reduce the time required to link executable programs.

The following simple example of iterative loading uses the files `a.c`, `b.c`, and `c.c` as sources.

```
/* a.c */
   main()
   {
        f();
   }

/* b.c */
   f()
   {
        g();
   }

/* c.c */
   g()
   {
        printf("  hello\n");
   }
```

1. Compile all the sources to objects:

```
% cc -c {a,b,c}.c
  a.c:
  b.c:
  c.c:
```

2. Load the files `b.o` and `c.o` into a common object file:

```
% ld -r -o ofiles b.o c.o
```

3. Load `a.o` with the object file containing `b` and `c`:

```
% cc a.o ofiles
```

4. Execute your program:

```
% a.out
  hello
```

# ld command

# 2

## Introduction

This chapter describes the format of the ld command and provides information on the options and file arguments used by the command.

## Caution

**The loader should be invoked by a compiler (for example, fc or cc). Though the loader can be invoked directly in ld command lines to load previously compiled object files, this practice is not recommended. Safeguards have been built into the compilers to insulate you from the effects of changes that result from CONVEX software revisions. These safeguards are not present if you invoke the loader directly through an ld command line.**

**In particular, the names of libraries and the linking order are subject to change between releases of the ConvexOS operating system and of the compilers. The compilers are designed to preserve backward compatibility across such changes; the ld interface is not.**

**Loader command line examples given in this manual are intended to provide general guidance; the actual parameters required with such commands may differ from the examples, depending on which version of compiler and operating system software is used.**

## Command format

The `ld` command invokes the loader program. You can use this command either interactively or as part of a shell script. The format of the `ld` command is

`ld [`*options*`]` *objfiles* `[...]` `[`*libraries*`]`

The *options* you can specify on the command line are described in the section "Loader options," on page 11. By convention, names of *objfiles* end with `.o` and names of *libraries* end with the `.a` suffix.

Arguments follow `ld` on the command line. The loader processes the argument files in the order specified. Libraries should generally appear at the end of the command line in order to resolve external references.

Following are examples of typical `ld` commands. The command sequence

```
% ld -o mumble foo.o bar.o
```

combines the object files `foo.o` and `bar.o` into the executable program file, `mumble`.

The command sequence

```
% ld -o myprog /usr/lib/crt/crt0.o main.o \
  subr1.o subr2.o -lc
```

combines the C-generated object files `main.o`, `subr1.o`, and `subr2.o` with the runtime initialization routine `crt0.o` and the ANSI C runtime library. The executable file is named `myprog`.

The sequence

```
% ld -o myprog /usr/lib/crt/crt0.o main.o \
  subr1.o subr2.o mylib.a -lU77 -lF77 -lI77 \
  -lD77 -lmathC1 -lc
```

combines the FORTRAN-generated object files `main.o`, `subr1.o`, and `subr2.o` with the runtime initialization routine `crt0.o` and the FORTRAN libraries. The executable file is named `myprog`.

## Loader processing

Following is a general description of the processing carried out by the loader.

1. Each object file encountered on the command line is unconditionally loaded. As the object file is encountered, symbols are added to the global symbol table and segment sizes are accumulated.

2. If a library is encountered, it is scanned and objects from it are loaded *only* if they satisfy an unresolved external reference. If they do, they are processed just like object files encountered.

3. If unresolved symbols remain and the `-m` option was specified, the libraries are rescanned until the load is complete or no symbols are resolved during a pass through all the object files and libraries.

4. After the first pass through the input files, the common blocks are allocated space at the end of the `.bss` and `.data` sections. The special symbols (`_end`, `_etext`, `_edata`, `_etdata`, and so on,) are then created. `_end` is always created; the others are created if referenced.

5. Next, the values of all symbols in the symbol table are adjusted by the origins of their respective segments.

6. Each of the included object files and library members are now reprocessed in the order they appeared on the command line. For each object module, the symbol table is processed to check for multiply-defined symbols, and the symbols are written to the output file. The text and data for the object are then copied to the output file, with relocatable references properly updated.

## Loader options

Options available with the loader are given below. You can list options that have no following values on the command line either separately (for example, `-s -d`) or in a concatenated form (for example, `-sd`). Options are preceded by a hyphen. With the exception of `-fmode`, `-lname`, and `-Moption`, options and arguments can be separated by a space or not, as you prefer.

The order in which the options `-l`, `-t`, `-u`, `-y`, `-L`, and `-NL` are arranged on the command line is significant. This is because these options are interpreted when they are encountered as the command line is scanned on the first pass of the loader. Also,

ordering of flags `-s, -x`, and `-X` is extremely important. These options *must* precede any object files on the command line. If they are encountered after any objects or libraries are encountered, an error message is printed, and the load aborts. (Refer to Appendix A for descriptions of some ambiguous loader error messages.) If you use the `-s, -x`, or `-X` options, place them first on the command line. The order of the other loader options is not significant. The following options are available:

-A*alias=symbol*

Alias all occurrences of *alias* to *symbol*. This logically renames *alias* to *symbol*, causing all references of the symbol *alias* to resolve to the object pointed to by *symbol*.

-A*file*

Read an alias list from *file*. The alias list is in the form *alias =symbol*, listed one to each line.

-D *size*

Instructs the loader to add zeros to the data segment until it matches the specified *size* (a hexadecimal number). If the length indicated is not a multiple of 4096, the final size of the data segment will be rounded up to the nearest multiple of 4096.

`-d`

Instructs the loader to force a common storage definition even if the `-r` option is present.

-E *mode*

Changes an execution *mode* bit of the output file produced. This option can be used multiple times to change several *mode* bits. This option understands the following modes:

`-E demand`—sets the demand-paged option.
`-E nonswap`—sets the nonswapped option.
`-E noposix`—clears the POSIX executable bit.
`-E posix`—sets the POSIX executable bit.
`-E prepage`—sets the prepaged option.

You must have root priveleges to run a program in nonswapped execution mode. Once an executable has been built, you cannot change the posix bit by invoking `ld` with a different `-E` setting.

-e *name*
Instructs the loader to treat *name* as the symbol name that will be the entry point of the loaded program. Location `start` is the default. If `start` is not defined, the loader uses `_main`. If `_main` is undefined, the loader uses the first module loaded.

-F
Forces creation of an executable file even if unresolved externals still exist when the load operation finishes. The file produced does not have the executable-mode bit set; you must use the `chmod` utility to make the file executable. This option also causes `ranlib` to execute on any out-of-date libraries. Normally, the load simply terminates.

-f*mode*
Specifies the floating-point format expected of all object modules. Valid `mode` requests are `i` and `n`, for IEEE and native floating-point format, respectively. All modules will be checked for compatibility with the expected mode, and the link will not succeed if any module is in conflict. If the `-f` option is not included on the command line, the loader determines the floating-point format from the object files that it is processing.

**Note**

**Your system administrator may have chosen a default floating-point mode (either native or IEEE). Use the `getsysinfo` utility to see if a default floating-point mode exists and it is the one you want to use.**

-Hxxxx
Leaves empty space of size *xxxx* bytes at the end of the text segment; *xxxx* is specified in hexadecimal. The actual size of the empty space may be larger than the value specified. After adding the amount specified, `ld` rounds the segment size up to the nearest multiple of 4096.

-L *path*
Searches *path* for libraries (in addition to the regular search path). You can use this option more than once; each use adds a path to the search list. Paths specified by `-L` are searched before the standard library paths.

-l*name*
Is an abbreviation for the library name `/lib/lib`*name*`.a`, where *name* is a string. If that library does not exist, `ld` searches for `/usr/lib/lib`*name*`.a`. If that library in turn does not exist, `ld` searches for `/usr/local/lib/lib`*name*`.a`. The loader searches a library when it encounters the library's name, so the placement of the option is significant.

-M*option*

Instructs the loader to produce a load map. If no option is given, a primitive map consisting of a list of the object files included is produced. Three predefined maps are available. The option `S` produces a short map, `M` produces a medium map, and `L` produces a long map. You can also create your own map by specifying only the sections you want to be output. You can select from the following sections:

| | |
|---|---|
| `h` | Map header |
| `l` | Library modules included |
| `o` | Object modules included |
| `g` | Global symbols sorted by address |
| `s` | Global symbols sorted by name |
| `n` | Local symbols for each object file by name |

The sections `h`, `l`, and `o` make up the short map; sections `h`, `l`, `o`, `g`, and `s` make up the medium map; and all the sections together make up the long map.

You can also use option `P` with any of the predefined maps or when specifying individual sections. Option `P` removes most of the formatting from the map so that it can be more easily processed by other programs. You can use a load map in to determine the following:

– The location of objects are loaded.

– The location where a program aborts using `adb` .

– Whether a required program section exists.

– What options are used to invoke the loader.

`-m`

Performs multiple scans of libraries. If any unresolved externals remain at the end of the load operation, the libraries specified on the command line are rescanned. Libraries are scanned in the order specified on the command line and the scanning continues until one full pass is made and no object modules are extracted.

`-NL`

Ignores the standard search path for libraries. The option `-NL` in conjunction with the `-L` option enables you to scan only the directories you specify.

-o *file*
Instructs the loader to use *file*, rather than `a.out`, as the name of the `ld` output file.

-p
The `-p` option performs the following: outputs `o_entry` in the optional header and preserves relocation information in the presence of the `-r` option; allows warnings about undefined references in the presence of the `-F` option; and obeys the `-d` option.

-R *ring*
Relocates the resulting output file to the ring number specified in the *ring* argument. Typically, this option is used during the system-generation procedure to rebuild the operating system residing in ring 0. Because of the protection mechanisms in operation, it is impossible to execute a program that specifies a ring number less than 4 while the system is running. The default is ring 4. Refer to the *CONVEX C Series Architecture Reference Manual* for more information.

-r
Instructs the loader to generate relocation bits in the output file so that the file can be loaded again. This option also prevents final definitions from being given to common symbols and suppresses the "undefined symbol" diagnostics.

-s
Saves space by "stripping" the output of the symbol table and relocation bits. (One disadvantage to this process is that it impairs the usefulness of the debuggers.) This information can also be removed from the output file with the `strip` command.

-T *origin*
Signals the loader to use *origin* (a hexadecimal number) as the address of the text segment origin. The default origin is 1000 hex (4096 decimal). Any origin you specify must be a multiple of 4096 (decimal).

-t
Causes the loader to print the path name of each file as it is processed.

-u *sym*
: Causes the loader to create *sym* as a symbol and enter it as undefined in the symbol table. This capability is useful for loading routines exclusively from a library, because initially, the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. Refer to the section "Loading FORTRAN programs," on page 18, for an example.

-v *ver*
: Sets the version stamp in the file header. Same as executing the `vers` command with the `-v` option.

-w
: Suppresses warning messages from the loader.

-X
: Used by the C and FORTRAN compilers to discard internally generated labels (those that begin with `L`, `.L`, or `ABS` symbols), while retaining symbols local to routines.

-x
: Instructs the loader to omit local (nonglobal) symbols from the output symbol table and only enter external (global) symbols. This option reduces the size of the output file.

-y *sym*
: Displays each file in which the symbol *sym* appears. This option also displays the symbol type and indicates whether the file defines or references *sym*. For example, if the loader reports that a symbol (for example, `_ralph`) is unresolved, the load can be run again by specifying the option `-y _ralph`. The loader responds with a list of names of all object files that refer to `_ralph`. (It is usually necessary to begin *sym* with an underscore "_" because external C and FORTRAN variables begin with underscores.)

## Reprocessing executable files

If you give the loader a single executable file as input, it uses the options you have specified to set the flags in the file header and create a new executable file with the appropriate bits set. For example, a normal executable file can be converted to a prepaged executable without having to go through the entire load process a second time, or the floating-point format option for an executable file can be changed to indicate IEEE format.

## Note

**This type of conversion can be performed on only one executable file at a time; it is an error to specify more than one executable file as input to the loader (no object files can be specified).**

Table 1 indicates options that can be used for reprocessing executable files, along with the action performed when the option is used.

Table 1
Reprocessing file options

| Option | Action |
|---|---|
| `-fn` | Sets native floating-point mode. |
| `-fi` | Sets IEEE floating-point mode. |
| `-E demand` | Sets the demand-paged option. |
| `-E prepage` | Sets the prepaged option. |
| `-E nonswap` | Sets the nonswapped option. |

## Loading C programs

CONVEX does not recommend loading C programs manually because the order of command line options may potentially change with each release of the C compiler. The driver for the C compiler automatically invokes the loader with the correct options and libraries in the proper order.

For example, to load C object files `a.o` and `b.o`, use the following command line:

```
% cc a.o b.o
```

The resulting executable program is placed in `a.out`. Many of the C command line options affect options that are passed to the loader. Among these options are `-fi`, `-fn`, `-tm`, `-db`, and `-p`.

The driver for the C compiler passes the following options directly to the loader:

```
-A -D -d -E -e -F -L -l -M -m -r -s -T -t -u -X
-x -y
```

Additional arguments required by the loader can be specified on the C command line following the `-link` option. If the argument does not start with a dash (-) or starts with a `-l` (ell), the argument is added to the loader's file list.

For example, the following command line links the file `a.o` with the libraries `one.a` and `/usr/lib/libdbm.a`:

```
% cc a.o -link one -link -ldbm
```

Any arguments of the `cc` command line option `-link` that begin with a - (other than `-l`), are assumed to be loader options. Loader options that require a value, such as `-H`, must be written with no space between the option and its argument. For example, the following command line leaves an empty space of 4096 bytes at the end of the text segment:

```
% cc a.o -link -H4096
```

The following command lines load profiled C object files, C object files that will be used with `csd`, and C object files that call math functions, respectively:

```
% cc -p a.o
% cc -db a.o
% cc a.o
```

## Loading FORTRAN programs

CONVEX does not recommend loading FORTRAN programs manually. Instead, use the `fc` command. This ensures that the proper FORTRAN libraries are automatically loaded in the correct order. The compiler, in turn, passes any loader options on the `fc` command line to the loader. Options such as `-E`, which are loader options as well as `fc` options, are interpreted as `fc` options. For more information on the `fc` command, refer to the *CONVEX FORTRAN User's Guide* (DSW-038).

For example, to load FORTRAN object files `x.o` and `y.o`, use the following command line:

```
% fc x.o y.o
```

To load a FORTRAN program that uses the `prof` profiler, use the following command line:

```
% fc -p objfiles
```

To load FORTRAN programs that have been compiled with the VAX FORTRAN compatibility option, use the following command line:

```
% fc -vfc objfiles
```

To use the source-code debugger `csd`, the following command line should be used:

```
% fc -db objfiles
```

It is also possible to load a program from libraries only. In other words, you can perform a load without specifying any `.o` (or `.f`) files on the `fc` command line. Use the `-u` loader option to insert a symbol as undefined in the generated symbol table. This unresolved reference forces the first routine to be loaded. The following command line example loads a FORTRAN program that has been archived as a library:

```
% fc -u _MAIN__ proglib.a
```

`_MAIN__` is the name of the CONVEX FORTRAN internal routine thats gets control at runtime and eventually invokes the user's main program.

# Object file format

# 3

This chapter describes the 7 sections of the object file:

- header
- text segment
- data segment
- text-relocation information
- data-relocation information
- symbol table
- string table

This file format is used for object files produced with the operating systems through V6.0. Starting with V6.1 of the operating system, CONVEX uses a different file format, called standard object file format (SOFF), which is described in Chapter 4 of this manual.

## Header format

The object file header contains a number of entries that provide information on the file format and the size and location of various segments of the file. Figure 4 shows the format of the header.

**Figure 4**
Header section format (from <a.out.h>)

```
/*
 * Header prepended to each a.out file
 */
struct exec {
        long                    a_magic;        /* magic number */
        unsigned long           a_text;         /* text segment size */
        unsigned long           a_data;         /* initialized data size */
        unsigned long           a_bss;          /* uninitialized data size */
        unsigned long           a_syms;         /* size of symbol table */
        unsigned long           a_entry;        /* entry point */
        unsigned long           a_trsize;       /* size of text relocation */
        unsigned long           a_drsize;       /* size of data relocation */
        unsigned long           a_talign;       /* byte alignment of text */
        unsigned long           a_dalign;       /* byte alignment of data */
        unsigned long           a_balign;       /* byte alignment of bss */
        unsigned long           a_torigin;      /* starting address of text */
};

#define OMAGIC  0507    /* old impure */
#define NMAGIC  0510    /* compromise format */
#define ZMAGIC  0513    /* demand load format */
#define PMAGIC  0515    /* prepaged load format */
#define LMAGIC  0517    /* prepaged, nonswapped load format */
```

The fields of the exec structure are defined as::

a_magic
: First 32 bits of the header file, which specify the object file type. If the magic number is 0513 (octal), the data and text segments in the object file are aligned on 4096-byte boundaries, enabling demand-paging of the file. These files are called ZMAGIC-format files. If the magic number is 0507 (octal), the file is unaligned and unexecutable and is called an OMAGIC-format file.

a_text
: Size of the program text segment, in bytes.

a_data
: Size of the initialized portion of the data segment, in bytes.

a_bss
: Size of the uninitialized portion of the data segment, in bytes.

a_syms
: Size of the symbol table, in bytes.

a_entry
: Address at which program execution begins.

a_trsize
: Size of the text relocation commands, in bytes.

a_drsize
: Size of the data relocation commands, in bytes.

a_talign
: The byte alignment of the text segment.

a_dalign
: The byte alignment of the data segment.

a_balign
: The byte alignment of the bss segment.

a_torigin
: The starting address of the text segment.

## Text segment

The text segment of an object module contains the executable binary code and read-only constants. The text segment of object files with the ZMAGIC magic number begins at byte address 4096 in the object file. This 4096-byte alignment allows the operating system to demand-page the program directly from the executable file. ZMAGIC format files have their text and data segments padded to 4096-byte boundaries. OMAGIC format files begin the text segment immediately after the header. (The N_TXOFF macro in a.out.h returns the absolute position of text in OMAGIC format files when given the name of an exec structure as an argument.)

## Data segment

In ZMAGIC format files, initialized data starts at the next 4096-byte boundary after the end of the text segment. In OMAGIC format files, initialized data immediately follows the text segment.

## Relocation entries

Certain portions of any given object module may contain references to addresses whose values are not known when the object file is assembled. For example, instructions in the text segment of the object module may refer to addresses in the data segment. These data segment addresses must be relocated (biased) by the final memory base address of the data segment. In these cases, the assembler communicates these unresolved addresses to the loader via relocation entries (see Figure 5), and the loader generates the final addresses of all the unknown values.

There are two sets of relocation entries within each object module: text-relocation and data relocation. Text-relocation entries reference unknown values in the text segment the module. Data-relocation entries reference unknown values in the data segment of the module. Each entry contains the location of the unknown value and the ordinal of the symbol table entry whose address is unknown to the assembler. The loader processes the relocation entries and replaces the unknown values with fixedaddresses.

The text- and data-relocation entries immediately follow the data segment of an object module. If relocation information is present, it amounts to 8 bytes per relocatable datum, as Figure 5 indicates.

**Figure 5**
Format of a relocation entry (from `<a.out.h>`)

```
struct relocation_info {
        int             r_address;        /* address that is relocated */
        unsigned int    r_fill: 4;        /* nothing, yet */
        unsigned int    r_extern: 1;      /* does not include value of sym
                                             referenced */
        unsigned int    r_length: 2;      /* 0=byte, 1=halfword, 2=word */
        unsigned int    r_pcrel: 1;       /* no longer used */
        unsigned int    r_symbolnum: 24;  /* local symbol ordinal */
};
```

The fields of the `relocation_info` structure are defined as:

`r_address`
: Offset within the text or data section of a value that requires relocation.

`r_extern`
When the `r_extern` field contains a 1, the `r_symbolnum` field is the ordinal of a symbol whose value replaces the unknown value. If the relocation-entry field `r_extern` contains a value of 0, no external value is required for the unknown value. Instead, the value in `r_symbolnum` is a segment type (for example, `N_TEXT` for the text segment), and the starting address of that segment relocates the contents of the relocation address.

`r_length`
Size of the value to be relocated. If the value is 8 bits long, `r_length` = 0; 16 bits long, `r_length` = 1; and 32 bits long, `r_length` = 2.

`r_pcrel`
No longer used.

`r_symbolnum`
Ordinal symbol table entry number for a symbol whose value is used for relocation. Symbol table entries range in ordinal value from 0 to a number that is one less than the number of symbols in the symbol table.

The `a_trsize` and `a_drsize` fields in the object file header give the size (in bytes) of the relocation entries for the text and data segments. If either `a_trsize` or `a_drsize` contains a value of 0, no relocation entries are found in the file for the corresponding text or data segment.

## Symbol table

The symbol table follows the relocation information in the object module. (The location of the symbol table can be computed by the `N_SYMOFF` macro in `a.out.h`.)

In order to resolve external references between modules discussed above, the assembler creates a list of symbols, or a symbol table, in the object file. The symbol table entries contain all of the information that the assembler knows about the symbol, except its name. The `n_strx` field in the symbol table entry carries the pointer offset to the spelling of a symbol name in the string table. (Refer to the section "String table," on page 28, for more information.) The symbol table contains an entry for each symbol name found in the file, whether or not its location can be resolved.

The loader uses the symbol tables in object modules to resolve the references contained in the relocation entries. The `r_symbolnum` field of each relocation entry contains an ordinal

number that refers to one of the symbol table entries. The layout of a symbol table entry and the principal flag values that distinguish symbol types are shown in Figure 6.

**Figure 6**
Symbol table entry format (from `<nlist.h>`)

```
struct nlist {
        union {
                char            *n_name;   /* for use when in core */
                long            n_strx;    /* index into file string table */
        } n_un;
        unsigned char           n_type;         /* type flag */
        char                    n_other;        /* unused */
        short                   n_desc;         /* see <stab.h> */
        unsigned long           n_value;        /* value of symbol
                                                   (or sob offset) */
};
/*
 * Simple values for n_type
 */
#define  N_UNDF  0x0            /* undefined */
#define  N_ABS   0x2            /* absolute */
#define  N_TEXT  0x4            /* text */
#define  N_DATA  0x6            /* data */
#define  N_BSS   0x8            /* bss */
#define  N_COMM  0x12           /* uninitialized common */
#define  N_FN    0x1f           /* file name symbol */
#define  N_EXT   01             /* external bit, OR'ed in */
#define  N_TYPE  0x1e           /* mask for all type bits */
```

Descriptions of the obects in Figure 6 are:

`n_strx`
: Contains an offset into the string table, which is a pool of symbol names. This value is relative to the start of the string table.

`n_type`
: Contains a value that designates the symbol type. (Values for `n_type` are described in Table 2.)

**Table 2**
ntype flags (from <nlist.h>)

| Flag | Value | Description |
|---|---|---|
| N_ABS | 0x2 | The symbol is in the absolute segment. |
| N_BSS | 0x8 | The symbol is in the bss segment. |
| N_COMM | 0x12 | The symbol is an uninitialized common block. |
| N_DATA | 0x6 | The symbol is in the data segment. |
| N_EXT | 01 | External bit. |
| N_FORMAT | "%08x" | Format for name list values. |
| N_FN | 0x1f | File name symbol. |
| N_SCNHDR | 0x10 | The symbol is an initialized common block. |
| N_STAB | 0xe0 | The symbol is used in csd. |
| N_TBSS | 0xc | The symbol is for thread bss. |
| N_TCBSS | 0x16 | The symbol is an uninitialized thread common block. |
| N_TCDATA | 0x14 | The symbol is an initialized thread common block. |
| N_TDATA | 0xa | The symbol is for thread data. |
| N_TEXT | 0x4 | The symbol is in the text segment. |

n_other
Unused.

n_desc
Used only by symbol table (STAB) entries and serves as an index into the segment header table to the header that further defines the symbol. As an example, an initialized common block entered in the symbol table has the N_SCNHDR flag set and a segment header created for it, and the n_desc field is set to the index of the new segment header. In the segment header, the S_COMON flag is set, indicating that it is an initialized common block. For more information on the segment headers, refer to the section "Segment headers," on page 35.

`n_value`

Contains the value of this symbol. Generally, the field contains a resolved address. (Remember that even executable files with all of their references resolved can still contain symbol tables.)

Use the `nlist` function to examine the symbol table of an object module. Refer to the nlist(3) man page for more information.

## String table

The `n_strx` field of the symbol table entry contains an offset into the string table. The string table is a pool of null-terminated strings that are symbol name spellings. The string table immediately follows the symbol table in the object module.

# Standard object file format (SOFF)

# 4

Versions V6.1 and later of the CONVEX operating system produce object files in a format called the standard object file format (SOFF). This chapter describes the following features of SOFF files:

- File header
- Optional header
- Program segment headers
- Segment data
- Relocation information
- Symbol table
- String table

The SOFF format requires that the segments be arranged in the following order: the file header first, then the optional header. All the other segments can be arranged in any order.

**Caution**

**All pointer fields in the SOFF format are specified as `long long` words. Remember to cast them to an `off_t` before using them with routines such as `fseek` or `lseek`.**

## File header

The file header contains general information about the SOFF file, such as the magic number, version number, time and date stamp, number of segments defined, and so forth.

Figure 7 shows the layout of the file header.

**Figure 7**
File header layout (from `<convex/filehdr.h>`)

```
/*
 *  filehdr.h - Definition for the SOFF file header
 */
#ifndef _CONVEX_FILEHDR_H                  /* To allow multiple includes */
#define _CONVEX_FILEHDR_H

struct filehdr {
        unsigned long       h_magic;    /* Magic number                     */
        unsigned long       h_version;  /* Object file version number       */
        long                h_timdat;   /* Time stamp of file creation      */
        unsigned long       h_nscns;    /* Number of sections in file       */
        unsigned long long  h_scnptr;   /* Ptr to first section header      */
        unsigned long       h_opthdr;  /* Size, in bytes, of optional header*/
        unsigned long long  h_strptr;   /* Offset in file of string table  */
        unsigned long long  h_strsiz;   /* Size, in bytes, of string table */
        unsigned long long  h_flags;    /* Flag word                        */
        long                h_spares[3];/* Room for growth                 */
};

#define FILEHDR struct filehdr
#define FILEHSZ sizeof(struct filehdr)

/* The magic number in this header defines that this is a SOFF file.
 * The o_flags field in the optional header (see <convex/opthdr.h>) defines
 * what kind of object/executable file it is.
 */

/*  Magic number */

#define SOFF_MAGIC 0600
#define USER_SOFF_MAGIC 0603
#define CORE_SOFF_MAGIC 0605
#define CHKPNT_SOFF_MAGIC 0607
#define  IS_SOFF_MAGIC(X)  ((X) == SOFF_MAGIC)

/* FLAG definitions (h_flags) */

#define H_COMM     0x0000000000000001LL   /* File contains .comm sections */
#define H_RSVDBITS 0x000000ffffffffffLL   /* Reserved bits                */
#define H_USERBITS 0xffffff0000000000LL   /* Reserved for users           */

#endif /* _CONVEX_FILEHDR_H */
```

where:

h_magic
: SOFF magic number SOFF_MAGIC defined in the include file filehdr.h. This magic number indicates that this is a SOFF format file. The macro IS_SOFF_MAGIC returns a nonzero value if the magic number passed to it is a valid SOFF magic number.

h_version
: A version number can be placed in this field by the assembler. It is not a required field, but is set to 0 if it is not used. The loader selects the largest version number of all object files processed and uses this value as the version number of the executable file produced. If you specify a version number on the ld command line, this number is used regardless of any version numbers found in the object files. You can use the vers command to set the version number after the executable file has been produced.

h_timdat
: Current time and date (from time) are placed in this field during assembly.

h_nscns
: Number of segment headers written to the object file.

h_scnptr
: Offset, from the beginning of the file, to the start of the first segment header. This value can be used with the fseek function to locate segment headers.

h_opthdr
: Size, in bytes, of the optional header in the object file. Currently, this is set to the value OPTHSZ, defined in the include file opthdr.h.

h_strptr
: Offset, from the beginning of the file, to the start of the string table. This value can be used with the fseek function to locate the string table.

h_strsiz
: Number of bytes in the string table. It can be used with the fread function to read the string table.

h_flags
: Table 3 shows the flags defined for use in a SOFF file header.

h_spares
Reserved for future expansion; it is always set to 0.

**Table 3**
File header flags

| Flag | Value | Description |
|---|---|---|
| H_COMM | 0x0000000000000001LL | Set if the file has segment headers that describe initialized common blocks. The assembler sets this flag if it has output a segment of this type. |
| H_RSVDBITS | 0x000000ffffffffffLL | This mask covers all the bits that are reserved for use by CONVEX. |
| H_USERBITS | 0xffffff0000000000LL | This mask covers all the bits that are reserved for users. |

## Optional header

The optional header is used to store specific information about object and executable files. It contains information such as the location of the symbol tables and the entry point for executable files.

### Note

**The optional header differs from the file header in that the file header stores information specific to the SOFF format only, not its application to object files.**

Figure 8 shows the layout of the optional header.

Figure 8
Optional header layout (from <convex/opthdr.h>)

```
/*
 * opthdr.h - Definitions for SOFF optional header
 */
#ifndef _CONVEX_OPTHDR_H                      /* to avoid multiple inclusions */
#define _CONVEX_OPTHDR_H

  struct opthdr {
         unsigned long long o_symptr;   /* Offset in file of symbol table */
         unsigned long      o_nsyms;    /* Num of entries in symbol table */
         long               o_spare;    /* RESERVED - must be 0 */
         unsigned long      o_entry;    /* Entry point */
         unsigned long long o_flags;    /* defined below */
  };
#define OPTHDR struct opthdr
#define OPTHSZ sizeof(struct opthdr)
/*
 * Flag bits defined in o_flags...
 */
#define OF_DEMAND           0x0000000000000001LL  /* old ZMAGIC */
#define OF_PREPAGED         0x0000000000000002LL  /* old PMAGIC */
   .
   .
   .
#define OF_EXEC   0x8000000000000000LL
#endif           /* _CONVEX_OPTHDR_H */
```

The objects in Figure 8 are defined as:

`o_symptr`
: Offset, from the beginning of the file, to the start of the symbol table. This value can be used with the `fseek` function to locate symbol table entries.

`o_nsyms`
: Number of symbol table entries written to the symbol table. This value can be used with the `fread` function to read the symbol table.

`o_spare`
: Reserved for future expansion; this field is set to 0.

`o_entry`
: Stores the entry point of the executable image. This field is set by the loader. The assembler always sets this field to 0.

o_flags

Optional header flags are used to indicate a variety of information about the object and executable files contained in the SOFF file. These flags determine items such as execution modes, IEEE usage, whether the file is stripped, and so on.

Table 4 shows the flags defined for use in the optional header.

**Table 4**
Optional header flags

| Flag | Value | Description |
|---|---|---|
| OF_DEMAND | 0x0000000000000001LL | Set if the executable is to be demand paged. |
| OF_PREPAGED | 0x0000000000000002LL | Set if the executable is to be prepaged. |
| OF_NON_SWAP | 0x0000000000000004LL | Set if the executable is not to be swapped and should be locked in memory. |
| OF_POSIX | 0x0000000000000008LL | Set if the executable is to be POSIX conforming. |
| OF_RSVD_EXEC_BITS | 0x0000000000000ff0LL | Bits reserved for future expansion of the execution mode flags. |
| OF_EXEC_MASK | 0x0000000000000fffLL | This mask is used to set/extract the executable bits combined. |
| OF_INST_C1 | 0x0000000000000000LL | C1 Series instruction |
| OF_INST_C2 | 0x0000000000010000LL | C2 Series instruction |
| OF_INST_C2MP | 0x0000000000020000LL | C2 Series multiprocessor instruction |
| OF_INST_PARALLEL | 0x0000000000040000LL | Parallel instruction |
| OF_INST_INTRINSIC | 0x0000000000080000LL | Intrinsic instruction |
| OF_INST_MASK | 0x00000000000f0000LL | Instruction mask |
| OF_HDW_IGNORE | 0x0000000000100000LL | Ignore hardware features |
| OF_RESERVED0 | 0x0400000000000000LL | Reserved, 0. |

**Table 4 (continued)**
Optional header flags

| Flag | Value | Description |
|---|---|---|
| OF_NOT_VECTOR | 0x0800000000000000LL | Set if the code in this file does not use vector instructions. Not implemented at the present time. |
| OF_FP_MODE_NATIVE | 0x0000000000000000LL | Value if the code in this file uses only native-mode floating point arithmetic. |
| OF_FP_MODE_IEEE | 0x1000000000000000LL | Value if the code in this file uses only IEEE-mode floating point arithmetic. |
| OF_IEEE_MASK | 0x1800000000000000LL | This mask covers bits used to indicate the IEEE-mode of this object/executable. |
| OF_STRIPPED | 0x2000000000000000LL | Set if the file has been stripped using the `strip` utility. The assembler never sets this flag. The loader sets this flag when the `-s` command line flag is processed. |
| OF_OBJECT | 0x4000000000000000LL | Set if the file is an object file. The assembler sets this flag. |
| OF_EXEC | 0x8000000000000000LL | Set if the file is an executable image. The assembler never sets this flag; only the loader does. |

## Segment headers

Segment headers supply information about the various segments defined in the SOFF file. There are 5 predefined or standard segments—text, data, tdata, bss, and tbss—and 1 segment for each initialized common block.

A segment header is not written to the SOFF file if the segment it controls was not used. In other words, a segment with a size of 0 is not written to the output file.

## Predefined segments

The SOFF has 5 predefined segments, although none of these segments must exist in any particular object file. These segments handle the standard segments that occur in object files.

Table 5 shows the names of the 5 predefined segments and the type of data included in each.

**Table 5**
Predefined segments

| Segment name | Data type |
| --- | --- |
| `.text` | Instruction |
| `.data` | Initialized data |
| `.tdata` | Initialized thread data |
| `.bss` | Uninitialized data |
| `.tbss` | Uninitialized thread data |

These names are placed in the string table, although their location in the table is not important. The segments have names in the string table, but they do not have a symbol associated with them. The names `.text`, `.data`, `.tdata`, `.bss`, and `.tbss` are not reserved symbol names; they can be used as freely as any other symbol name.

For more information on program segments, refer to the section "Program-segment directives" on page 98. For information on multithreaded parallel programs, refer to the *CONVEX C Series Architecture Reference Manual,* section "Multithreaded execution (Forking/ASAP)." For information on debugging multithreaded parallel programs, refer to the *CONVEX adb (Assembly-Language Debugger) User's Guide,* Chapter 5, "Multithreaded Debugging."

## Initialized common blocks

The name of the segment that handles an initialized common block is the name of the common block. The symbol table entry and the segment header share an entry in the string table.

## Segment header layout

Figure 9 shows the layout of the segment headers.

**Figure 9**
Segment header layout (from <convex/scnhdr.h>)

```
/*
 * scnhdr.h - Definition for the SOFF section headers
 */

#ifndef _CONVEX_SCNHDR_H
#define _CONVEX_SCNHDR_H

#define  s_align s_vaddr   /* Also used for alignment data */

  struct scnhdr {
      unsigned long long s_strndx; /* Section name index in string table */
      unsigned long s_vaddr;       /* Virtual load address of section */
      unsigned long long s_size;   /* Size of section in bytes */
      unsigned long long s_scnptr; /* Offset in file to raw data */
      unsigned long long s_relptr; /* Offset in file to relocation data */
      unsigned long s_nrel;        /* Number of relocation entries */
      unsigned long s_prot;        /* Protection flags */
      unsigned long long s_flags;  /* Flags word */
  };

/*
 * s_vaddr and s_align are synonyms, s_align is used for alignment data in
 * object files since they can't have virtual addresses, and s_vaddr is used
 * in executable images, since they already have the alignment data figured
 * in to the load address.
 */

#define SCNHDR struct scnhdr
#define SCNHSZ sizeof(struct scnhdr)

   /* FLAG definitions */

   /* Section protection flags */

#define VM_PG_R   0x08     /* Section memory is readable */
#define VM_PG_W   0x04     /* Section memory is writable */
#define VM_PG_E   0x02     /* Section memory is executable */
#define VM_PG_N   0x01     /* Section memory is not cacheable */
```

**Figure 9 (continued)**
Segment header layout

```
    /* Section type flags */

#define S_XTYPE    0x00000000LL         /* extended type, not loaded */
#define S_TEXT     0x00000001LL         /* .text section */
#define S_DATA     0x00000002LL         /* .data section */
#define S_TDATA    0x00000003LL         /* unshared data section */
        .
        .
        .
#define S_USERBITS          0xffffff0000000000LL  /* Reserved for users */

    /* Protection flags (for s_prot, above) are defined in <convex/pte.h> */

#endif            /* _CONVEX_SCNHDR_H */
```

The objects in Figure 9 are defined as:

`s_strndx`
: Offset into the string table to the start of the segment name.

`s_align` or `s_vaddr`
: `s_align` and `s_vaddr` are synonyms. This field serves two purposes. For object files, it is the alignment requested for the segment, specified with the `.align` directive to the assembler. For executable files, it is the virtual address where the segment is loaded. All references to this field use the proper name (`#defines` are used to make the two names synonymous) for the value desired. That is, references to get or set the segment alignment use the `s_align` field name, whereas references to the virtual load address of an executable image use the `s_vaddr` field name.

  Default value for the alignment of any segment is 8. The assembler sets this field to 8 if no `.align` directive is encountered for a given segment. Otherwise, the field is set to the value specified in the `.align` directive.

`s_size`
: Number of bytes of data that the segment contains. For text segments, this is the size of the assembled code. For data segments, it is the size of the data defined. For bss segments, it is the size of bss to be allocated. Any segment with a size of 0 is not written to the object file.

s_scnptr
: Offset, from the beginning of the file, to the start of the raw data for the segment. This value can be used with the `fseek` function to locate the raw data for the segment. There is no restriction on placement of data in the file, except that it comes after the file and optional headers (the file and optional headers are the first sections in the file).

  Some segments do not have any data associated with them (for example, bss). For these segments, this field is set to 0. As a consequence, the setup for an object file that has bss is a segment header with the `S_BSS` flag set, a name of `.bss`, a size set to the needed bss area, and a `s_scnptr` field set to 0.

s_relptr
: Offset, from the start of the file, to the start of the relocation entries for the segment. This value can be used with the `fseek` function to locate the relocation entries. If the segment has no relocation information, this field is set to 0.

s_nrel
: Number of relocation entries written to the file for this segment. This value can be used with the `fread` function to read the relocation entries from the file. If the segment has no relocation information, this field is set to 0.

s_prot
: Protection flags for the segment when loaded into memory. The flags used are defined in the `<convex/pte.h>` include file. Table 6 describes these flags.

**Table 6**
Protection flags

| Flag | Value | Description |
|---|---|---|
| VM_PG_R | 0x00000008 | Marks pages that contain raw data for the segment to be readable by the running process. |
| VM_PG_W | 0x00000004 | Marks pages that contain raw data for the segment to be writable by the running process. |
| VM_PG_E | 0x00000002 | Marks pages that contain raw data for the segment to be executable. |
| VM_PG_N | 0x00000001 | Marks pages that contain raw data for the segment that cannot be put in the cache by the running process. |

s_flags
: Segment header flags, defined in Table 7.

**Table 7**
Segment header flags

| Flag | Value | Description |
| --- | --- | --- |
| S_XTYPE | 0x00000000LL | Marks the segment as an extended type that is not loaded. |
| S_TEXT | 0x00000001LL | Marks the segment as text. |
| S_DATA | 0x00000002LL | Marks the segment as data. |
| S_TDATA | 0x00000003LL | Marks the segment as unshared data. |
| S_BSS | 0x00000004LL | Marks the segment as bss. |
| S_TBSS | 0x00000005LL | Marks segments as unshared bss. |
| S_COMON | 0x00000006LL | Marks the segment as an initialized common block. |
| S_CONTEXT | 0x00000008LL | Marks the context segment. |
| S_TCONTEXT | 0x00000009LL | Marks the thread context segment. |
| S_TYPMASK | 0x000000ffLL | Mask to get section type . |
| S_FMTMASK | 0x0000ff00LL | Mask to get format flags. |
| S_RSVDBITS | 0x000000ffffffffffLL | This mask covers all the bits that are reserved for use by CONVEX. |
| S_USERBITS | 0xffffff0000000000LL | This mask covers all the bits that are reserved for users. |

## Segment data

Raw data for a segment depends on the type of segment. The format of the raw data is unchanged from that produced by the earlier version of the assembler. The difference, however, is where the data is written in the file. This difference, though, is of no consequence because the segment header field s_scnptr specifies the location, so no fixed position is required.

## Relocation information

Certain portions of any given object module can contain references to addresses whose values are not known when the object file is assembled. For example, instructions in the text segment of the object module can refer to addresses in the data segment. These data segment addresses are relocated (biased) by the final memory base address of the data segment. In cases such as these, the loader generates the final addresses of all the unknown values. The assembler communicates these unresolved addresses to the loader via relocation entries. Refer to Figure 10.

Each entry contains the location of the unknown value and the ordinal of the symbol table entry whose address is unknown to the assembler. The loader processes the relocation entries and replaces the unknown values with actual addresses.

If relocation information is present, it amounts to 8 bytes per relocatable datum, as Figure 10 indicates.

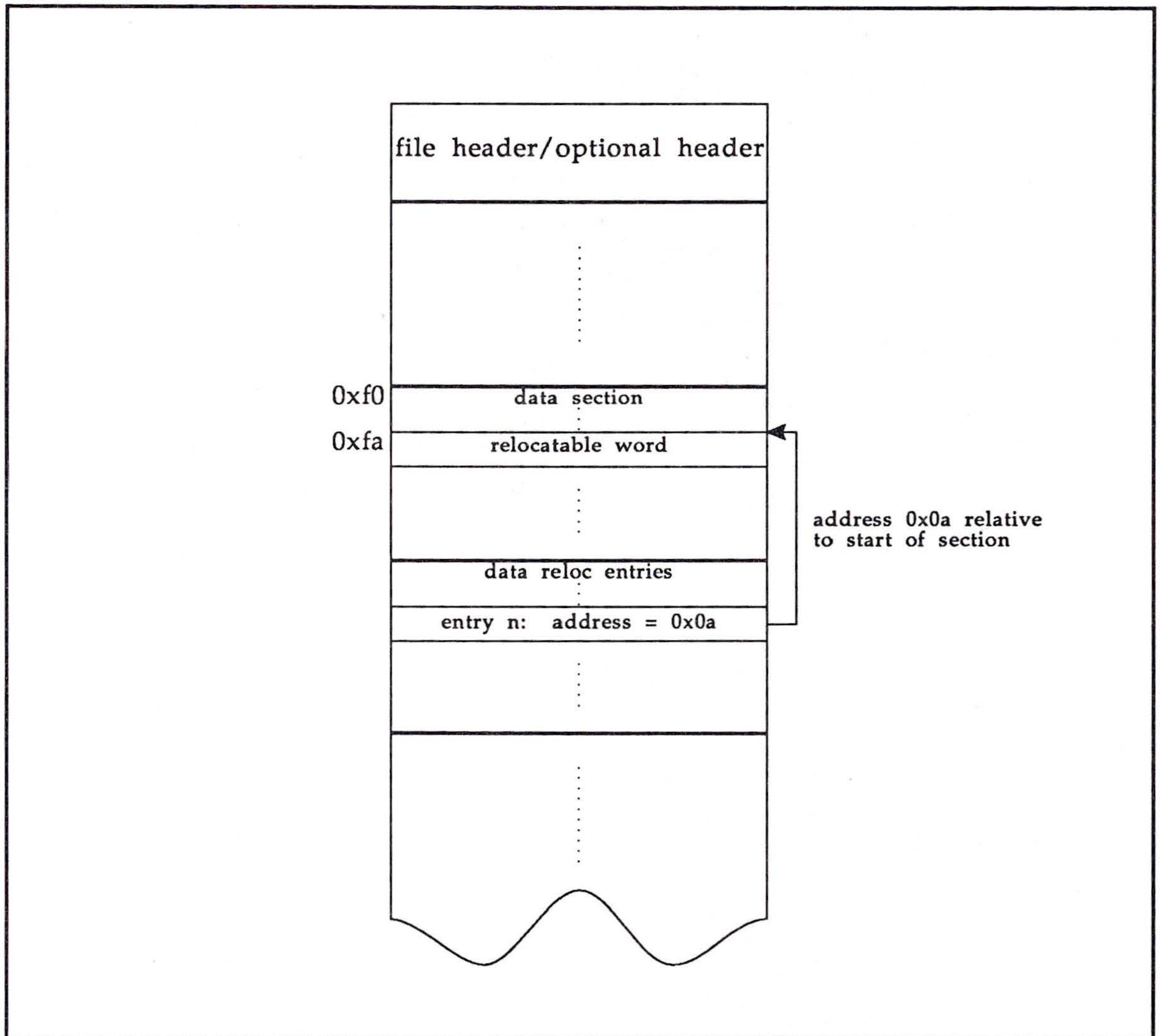
**Figure 10**
Relocation entry format (from `<convex/reloc.h>`)

```
/*
 * reloc.h - Definition of the SOFF relocation entry
 */

#ifndef _CONVEX_RELOC_H
#define _CONVEX_RELOC_H

/* Check if a.out.h has been included; skip struct declaration if so   */

#ifndef N_BADMAG
struct relocation_info {
        int             r_address;      /* Address that is relocated   */
        unsigned int    r_fill:4;       /* Nothing yet                 */
        unsigned int    r_extern:1;     /* Symbol is external          */
        unsigned int    r_length:2;     /* 0=byte,1=half,2=word,3=long */
        unsigned int    r_pcrel:1;      /* no longer used              */
        unsigned int    r_symbolnum:24; /* Local symbol ordinal        */
};
#endif /* N_BADMAG */

#define RELOC_INFO struct relocation_info
#define RELOC_INSZ sizeof(struct relocation_info)

#endif /* _CONVEX_RELOC_H */
```

The objects in Figure 10 are defined as:

`r_address`
: Offset from the beginning of the segment to the byte that starts the reference to the relocatable symbol. Figure 11 indicates how this address is used.

**Figure 11**
Relocatable address references



r_extern

When this field contains a value of 1, the `r_symbolnum` field is the ordinal of a symbol whose value replaces the unknown value. If the relocation entry field `r_extern` contains a value of 0, no external value is required for the unknown value. Instead, the value in `r_symbolnum` is a segment type (for example, `N_TEXT` for the text segment), and the starting address of that segment relocates the contents of the relocation address.

`r_length`

Size of the value to be relocated. If the value is 8 bits long, `r_length` = 0; 16 bits long, `r_length` = 1; and 32 bits long, `r_length` = 2.

`r_symbolnum`

Ordinal symbol table entry number for a symbol whose value is to be used for relocation. The symbol table entries range in ordinal value from 0 to a number that is 1 less than the number of symbols in the symbol table.

## Symbol table

The structure used for name list entries is unchanged in the SOFF implementation. There is still 1 entry for every symbol defined or referenced in any object file. Symbol tables are still used as described in Chapter 3. A flag value for the `n_type` field has been defined. This new flag is used in conjunction with handling common blocks.

Figure 12 shows the layout of the symbol table.

**Figure 12**
Symbol table entry format (from <nlist.h>)

```
/*      $CHeader: nlist.h 0.4 90/01/26 18:26:06 $*/
/*      Copyright 1988-1990 Convex Computer Corp.*/

#ifndef  __NLIST__
#define  __NLIST__        1

/* format of a symbol table entry */
struct nlist {
        union {
                char      *n_name;      /* for use when in core */
                long      n_strx;      /* index into string table */
        } n_un;                        /* union for name and string index */
        unsigned char     n_type;      /* type flag, see below */
        char              n_other;     /* unused */
        short             n_desc;      /* see <stab.h> */
        unsigned long     n_value;      /* value of symbol (or sdb offset)*/
};
#define  n_hash   n_desc                /* used internally by ld */

/* simple values for n_type */

#define  N_UNDF   0x0      /* undefined */
#define  N_ABS    0x2      /* absolute */
#define  N_TEXT   0x4      /* text */
#define  N_DATA   0x6      /* data */
#define  N_BSS    0x8      /* bss */
#define  N_TDATA  0xa      /* thread data */
#define  N_TBSS   0xc      /* thread bss */
#define  N_SCNHDR 0x10     /* extended symbol, defined by section hdr */
#define  N_COMM   0x12     /* uninitialized common */
#define  N_FN     0x1f     /* file name symbol */
#define  N_TCDATA 0x14     /* thread common initialized */
#define  N_TCBSS  0x16     /* thread common unintialized */
#define  N_EXT    01       /* external bit, or'ed in */
#define  N_TYPE   0x1e     /* mask for all the type bits */
/* csd entries have some of the N_STAB bits set. Given in <stab.h> */
#define  N_STAB   0xe0            /* any of these bits set -> csd entry */
/* format for namelist values */
#define  N_FORMAT "%08x"

/* FILE_MOD_FAIL is returned by libc routine nlist() to signify that  */
/* the file being read has been changed in the time it took to do a   */
/* symbol name lookup. */

#define FILE_MOD_FAIL      -2
#endif   /* __NLIST__ */
```

The objects in Figure 12 are defined as:

`n_strx`
: Contains an offset into the string table, which is a pool of symbol names. This value is relative to the start of the string table.

`n_type`
: Contains a string designating the type of symbol (described in Table 8).

**Table 8**
n_type flags

| Flag | Value | Description |
|---|---|---|
| N_ABS | 0x2 | The symbol is in the absolute segment. |
| N_BSS | 0x8 | The symbol is in the bss segment. |
| N_COMM | 0x12 | The symbol is an uninitialized common block. |
| N_DATA | 0x6 | The symbol is in the data segment. |
| N_EXT | 01 | External bit. |
| N_FORMAT | "%08x" | Format for name list values. |
| N_FN | 0x1f | File name symbol. |
| N_SCNHDR | 0x10 | The symbol is an initialized common block. |
| N_STAB | 0xe0 | The symbol is used in csd. |
| N_TBSS | 0xc | The symbol is for thread bss. |
| N_TCBSS | 0x16 | The symbol is an uninitialized thread common block. |
| N_TCDATA | 0x14 | The symbol is an initialized thread common block. |
| N_TDATA | 0xa | The symbol is for thread data. |
| N_TEXT | 0x4 | The symbol is in the text segment. |
| N_TYPE | 0x1e | Mask for all the type bits. |
| N_UNDF | 0x0 | The symbol is undefined. |

The flag N_SCNHDR indicates that the symbol is not fully described by the symbol table entry. It is further described in a segment header.

`n_other`

Unused.

`n_desc`

Used only by symbol table (STAB) entries and serves as an index into the segment header table to the header that further defines the symbol. As an example, an initialized common block entered in the symbol table has the `N_SCNHDR` flag set and a segment header created for it, and the `n_desc` field is set to the index of the new segment header. In the segment header, the `S_COMON` flag is set, indicating that it is an initialized common block.

`n_value`

Contains the value of this symbol. Generally, the contents of this field are a resolved address. (Remember that even executable files with all of their references resolved may still contain symbol tables.)

## `n_value` field

The use of the `n_value` field in the symbol table depends on the type of symbol referenced. The two symbol types are *relocatable* and *common*.

If the symbol is relocatable, the value stored is the offset from the start of the segment in which the word resides. With the old object file format, the value stored was its offset from the beginning of the file; now it is stored as the offset from the start of the segment. This is done to simplify the implementation of any utilities that generate SOFF files. Because the symbol values are not relative to their position in the output file, they can be created without first laying out the whole structure of the file. This allows segments to be created in temporary files, then combined into the final output file without having to alter all the symbol values.

The value of a common-block symbol, initialized or uninitialized, is its size in bytes. The assembler keeps track of the largest size specified for any given common block. Remember that an uninitialized common block is indicated by the `n_type` flags `N_UNDF` and `N_EXT` and a nonzero value along with an initialized common block indicated by the flag `N_SCNHDR`; it is further described in a segment header. Refer to the next section, "Handling initialized common blocks (ICBs)," for more details.

## Handling initialized common blocks (ICBs)

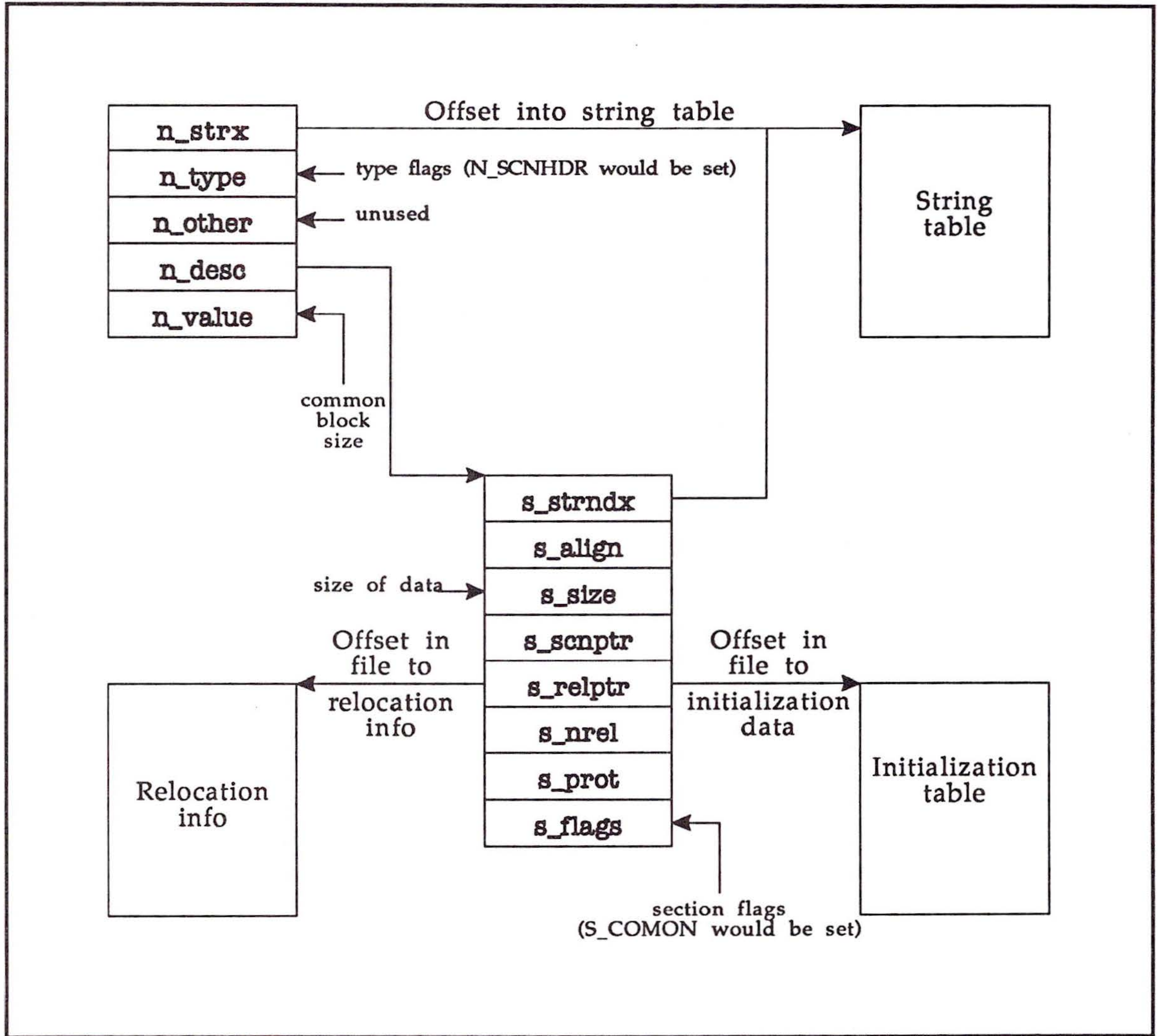
Initialized common blocks (ICBs) are handled differently from ordinary symbols. They are handled more like segments than anything else. There is a symbol in the name list for the common block, and it has an `n_type` flag `N_SCNHDR`. The `n_desc` field stores an index into the segment header table for the segment header that describes this common block. This index has its origin at 0 and is simply an index into the table of segment headers in the object file.

Each ICB is controlled with a segment header, as though it were a text or data segment. The size of the common block is kept in the `n_value` field of the symbol table entry. The `s_size` field in the segment header indicates the size of the raw data for the ICB. Using two fields to store size information for the ICB allows for alternate formats of raw data. If some form of compressed format is used, the size of the raw data may be different from the size of the common block.

The `s_scnptr` field in the header points to the initialization data for the ICB. In this way, the initialization data for a given ICB that is initialized in more than one object file can be properly sized and overlaid. ICBs can have relocation entries associated with them. This occurs if any of the initializers are based on relocatable values.

After object files have been loaded into an executable image, the ICBs no longer have segment headers. They have been merged with the data segment, just like any other data symbol. All uninitialized common blocks are rolled into the bss segment. Figure 13 illustrates interconnections that exist for an ICB between the symbol table and segment headers.

**Figure 13**
ICB symbol table interconnections



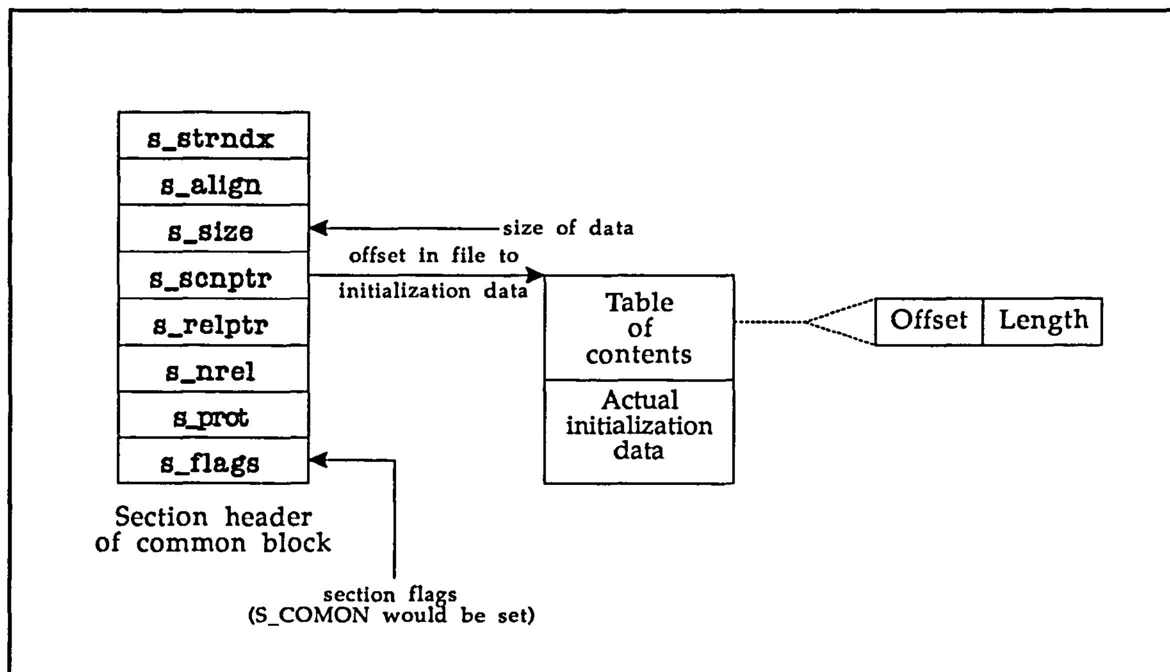

## Initialization data format

To properly keep track of the initializers applied to a particular common block and to check for multiply-initialized bytes, a "table of contents" of initializers is stored at the front of the raw data segment for an ICB. The first word in the raw data is the number of entries in the table of contents. This table of contents consists of a series of entries containing the offset into the common block and the length of the data initialized (refer to the

scnhdr(5) man page). The loader uses this data to determine any multiply-initialized bytes, as well as to control the process of overlaying initialization data.

Figure 14 illustrates how the raw ICB data is organized.

**Figure 14**
ICB raw data layout




## String table

Segment names and symbol names are stored contiguously in the string table as null-terminated character strings. The values stored in `s_strndx` and `n_strx`, for segment names and symbol names, are an offset from the start of the table to the first character in the name. These offsets start at 0 for the first name.

In the earlier object file format, the size of the string table was stored as the first word in the string table. This is no longer done. The total size, in bytes, of the string table is now stored in `h_strsiz` in the file header.

# Object libraries

# 5

This chapter discusses the location and contents of the standard system libraries and the auxiliary programs `ar`, `ranlib`, and `sod`. The `ar` and `ranlib` programs perform archiving and cataloging functions, including the addition and deletion of object modules within libraries. In particular, `ar` creates and updates library files used by the loader, whereas `ranlib` creates the library headers described later in this chapter. The `sod` program enables the user to dump a standard object file format (SOFF) file and examine its contents.

## Standard system libraries

System libraries available to CONVEX users are relocatable libraries. Relocatable libraries are files in which you can store, catalog, and reuse your routines. As the name implies, the final storage locations that the program uses when loaded into main memory are not fixed. Main-storage addresses are fixed when the loader extracts modules.

The contents of the libraries are object modules. A library header contains information enabling the loader to find object modules that contain a definition for an unresolved symbol.

The standard system library is located in the `/usr/lib` directory as follows:

/usr/lib/libansic.a
: Strictly conforming ANSI C mode runtime library.

/usr/lib/libc.a
: Extended mode C runtime library. This library contains a complete support package for C programming in the extended compatibility mode, including system calls, input/output, and math functions.

/usr/lib/libC1.a
: Machine dependent library for the CONVEX C1 Series computer.

/usr/lib/libC2.a
: Machine dependent library for the CONVEX C2 Series computer.

/usr/lib/libD77.a
: FORTRAN dummy VMS-to-UNIX file name translation routines.

/usr/lib/libF77.a
: FORTRAN intrinsic library; that is, math and string manipulation routines.

/usr/lib/libI77.a
: FORTRAN I/O library.

/usr/lib/libmathC1.a
: Math routines for the CONVEX C1 Series computer.

/usr/lib/libmathC2.a
: Math routines for the CONVEX C2 and C3 Series computers.

/usr/lib/libU77.a
: FORTRAN utility library; that is, `time`, `date`, `exit`, and so on.

/usr/lib/libV77.a
: FORTRAN constants for VAX FORTRAN compatibility.

Versions of these and other libraries can exist for other utilities or programs. These separate versions are indicated by several letters preceding the library extension. For example, the letters `_p` indicate that a profiled version of the library is suitable for use with `prof` or `grof`. Letters that indicate different versions of the libraries include:

_old
: These letters indicate a library that the backward compatible mode of the CONVEX C compiler uses. For example, `/usr/lib/libC1_old.a` is the CONVEX C1 Series machine dependent library used in the backward compatible mode of the CONVEX C compiler.

_p
: These letters indicate a library that has been profiled for use with the `prof` or `gprof` profilers. For example, `/usr/lib/libF77_p.a` is a profiled version of the CONVEX FORTRAN intrinsic library.

_pa
: These letters indicate a library that includes instrumentation for use with the CONVEX Performance Analyzer, CXpa. For example, `/usr/lib/libI77_pa.a` is an instrumented version of the CONVEX FORTRAN I/O library.

## Auxiliary programs

The following sections discuss

- ar
- ranlib
- sod

### ar

The `ar` program is a general-purpose utility that maintains groups of files combined into a single archive file. One use of `ar` is creating and updating library files used by the loader. The format of the `ar` command is

`ar` *key* [*posname*] *afile* `name`

where:

key
: Refers to the character that enables you to specify which tasks you want `ar` to perform. *keys* differ from switches in that switches must be preceded by a hyphen. *key* is one character from the set `drqtpmx`, optionally concatenated with one or more of the set `vuaibclo`. The values for *key* are defined below.

posname
: Refers to the name of the file used as a marker for positioning the new file in a library.

afile
: Is the archive file.

name

Refers to constituent files within the archive file.

Meanings of *key* characters are:

c

Creates the named file. Suppresses the message usually produced when *afile* is created. (The message can be very lengthy.)

d

Deletes the named files from the archive file.

l

Local. Normally, `ar` places its temporary files in the directory `/tmp`. This option places them in the local directory.

m

Moves the files named to the end of the archive. If a positioning character is present, the *posname* argument must be present.

p

Prints the named archive files.

q

Quickly appends the files named to the end of the archive file. Optional key positioning characters are invalid. The command does not check whether the added members are already in the archive. This key helps you avoid excessive processing time when creating a large archive piece by piece.

r

Replaces the named archive files. If the optional key character `u` is used with `r`, then only those files with "last-modified" dates later than the archive files are replaced. If `a`, `b`, or `i` is used, the *posname* argument must be present. New files are loaded before (use `b` as a key) or after (use `a`) the marker *posname*. If you do not use *posname*, the loader places new files at the end of the library.

t

Prints a table of contents of the archive file. If you do not specify names, all files in the archive are tabled. If you do specify names, only those files named are tabled.

v

Verbose. Gives a file-by-file description of the making of a new archive file from the old archive and the constituent file names. When used with t, this option gives a long listing of all information about the files. When used with p, this option precedes each file with a name.

x

Extracts the files named. If no names are given, all files in the archive are extracted. In neither case does x alter the archive file. Normally, the "last-modified" date of each extracted file is the date when it is extracted. If the o switch is used, however, the "last-modified" date is reset to the date recorded in the archive.

There are several reasons why a programmer may want to create a library. Two of the more common reasons are:

- To enhance the loading speed. When the loader links all the modules together, it is much faster for it to process one library than to process several .o files from the command line.
- To allow better control of site-specific routines. Several users can use one library file that contains site-specific routines. This avoids lengthy command lines and duplication of the routines among users; it also provides more consistency in the routines used.

The following subsections show how ar can be used to create and maintain libraries.

### Creating libraries

The ar utility can be used to create libraries.

**Example 1:**

```
% ar q sublib.a m10.o m20.o m30.o m40.o
ar: creating sublib.a
% ranlib sublib.a
```

**Example 2:**

```
% ar qc sublib.a m10.o m20.o m30.o m40.o
% ranlib sublib.a
```

In Example 1, you create a new library with the q (quick append) key. ar creates the library from the .o files. These files are placed in the library in the same order that they appear on

the command line. Example 2 uses the `c` key. This instructs `ar` not to print the usual creation message. (The `ranlib` utility must be run whenever a library is modified.)

### Adding new modules

When you add a new module to an existing library, you must use the `r` (replace) key. You may also use one of the optional keys `b` (before) or `a` (after). If the `a` or `b` key is used, the *posname* argument must also appear in the command. The new module is inserted after *posname* for `a` or before *posname* for `b`. Following are examples of adding new modules.

Example 3:

```
% ar rb m40.o sublib.a m35.o
% ranlib sublib.a
```

Example 4:

```
% ar ra m30.o sublib.a m35.o
% ranlib sublib.a
```

Example 5:

```
% ar r sublib.a m45.o
% ranlib sublib.a
```

Because the library `sublib.a` was created in Example 2, Example 3 and Example 4 produce identical results in that Example 3 adds new module `m35.o` before module `m40.o`, and Example 4 adds new module `m35.o` after module `m30.o`. Example 5 adds module `m45.o` to the end of the library `sublib.a`.

### Updating modules

To update a module in an existing library, use the `r` (replace) key and give the library name followed by the module name to be updated. If the module does not exist, the new module is appended to the end of the library file.

Example 6:

```
% ar r sublib.a m20.o
% ranlib sublib.a
```

### Deleting modules

To delete a module from an existing library, use the `d` (delete) key and simply give the library name followed by the module name you want to delete. The module names must be the same as they were when the modules were added to the library; that is, the `.o` extension is present.

Example 7:

```
% ar d sublib.a m40.o
% ranlib sublib.a
```

### Extracting modules

To extract a module from a library, use the `x` (extract) key. List the modules you want extracted after the library name. In the second example below, no module names are given, so `ar` assumes you want all modules within the given library extracted. This overwrites existing files with no warning. The module names must appear the same as when the modules were put in the library, and it is unnecessary to run `ranlib` after extracting modules from a library because this process does not modify the library.

Example 8:

```
% ar x sublib.a m10.o
```

Example 9:

```
% ar x sublib.a
```

### Listing module names

To list the names of all modules in a library file, use the `t` (table of contents) key. The next example illustrates use of the `t` key and lists sample output from a file named `sublib.a`. Command-line input is in bold, constant-width type.

```
% ar t sublib.a
m10.o
m20.o
m30.o
m40.o
```

To list the modules and the symbols contained in these modules, use the command `nm`. The next example illustrates use of the `nm` command and lists sample output for a file named `sublib.a`.

```
% nm sublib.a
allprint.o:
00000000   a .L1
00000000   a .L2
00000000   a .L3
00000132   t L10000
0000006c   t L13
000001b6   d L15
00000090   t L16
000001b9   d L17
000000b4   t L18
000001bc   d L19
000000d8   t L20
0000015a   t L2000001
000001bf   d L21
000000fc   t L24
000001b0   d L25
00000182   t L30
000001aa   t L9998
00000000   t LL0
           U __flsbuf
00000000   T _allprint
           U _fprintf
00000190   T _printable
00000158   T _sprint
           U _yyout

main.o:
00000000   a .L1
00000000   t LL0
           U _exit
00000000   T _main
           U _yylex
.
.
.
```

## ranlib

ranlib converts each archive file to a form that the loader can load more rapidly. It does this by adding a table-of-contents file named `__.SYMDEF` to the beginning of the archive. It uses `ar` to reconstruct the archive. The `ranlib` program must be run when the library is created and after each time the library file is moved or modified.

The format of the `ranlib` command is:

```
ranlib archivefile [...]
```

where:

archivefile
: Name of the library file to be converted.

## sod

sod is a utility that dumps the contents of a SOFF file and presents it in human-readable form.

The format of the `sod` command is

```
sod file [-l] [-f] [-n] [-t] \
    [-s[{h|d|r}] {name | all}]
```

where:

file
: Names the object file you want to examine.

-l
: Enables long-form printing. Flags in the file header, the optional header, and the section headers are expanded to names that show what is set. The default prints the flags as hexadecimal values only.

-f
: Prints the file header and the optional header.

-n
: Prints the symbol table (name list).

-t

Displays a table of contents of file sections. Each entry in the table gives the name of the section, its size, its flag word, the number of relocation entries for the section, and the alignment for the section. If the file has been stripped, the names of the sections are replaced with numeric identifiers that are used to display a particular section with the `-s` commands.

-s

Displays a file section. If *name* is given, only the named section is displayed. If `all` is used, all file sections are displayed. The modifiers `h`, `d`, and `r` select the format of the display. With no modifiers, `-s` displays only the section header. If `h` is specified, the section header is *not* displayed. If `d` is specified, the section's raw data (program text for the `text` section, for example) is printed. If `r` is specified, the section's relocation information is printed. The `-s` option can be repeated.

## Note

**If the symbol and string tables have been removed from *file* with the `strip` utility, `sod` does not display or recognize section names.**

If you specify no options on the command line, `sod` enters an interactive mode and prompts you for commands. All command-line arguments have the same use and meaning, though `-s` is not recognized.

The following additional options are available in interactive mode:

?
: Displays the help message.

c
: Prints the full path name of the current working file.

e
: Switches to a new `file`. `sod` displays the last component of the current working file path name in the prompt. When you change to a different file, the prompt also changes to show the new path name.

h
: Displays the help message, which lists valid options and explains each option.

l
: Switches the long form on and off.

q
: Quits `sod`.

# Loader examples

# 6

In the simplest case, the loader is used to create an executable program from separately compiled object modules. For example, the command sequence

```
% ld prog.o moda.o modb.o
```

causes the loader to combine modules `prog.o`, `moda.o`, and `modb.o`, and to resolve symbolic references between modules. The executable program file is named `a.out`.

To access the loader from the C compiler, type a command sequence similar to

```
% cc main.c sub1.o sub2.o -o main
```

This command sequence instructs `cc` to compile the C program `main.c` and to invoke `ld` to combine `main.o` with the object modules `sub1.o` and `sub2.o`. The `-o` option names the resulting executable program `main`.

Create user libraries with a command sequence similar to

```
% ar cv mylib.a foo.o bar.o mumble.o; ranlib \
  mylib.a
```

where the characters `cv` are *keys*, or options. In this case, `c` instructs `ar` to create the library `mylib.a` with constituent files `foo.o`, `bar.o`, and `mumble.o`. The `v` key instructs `ar` to give a file-by-file description of the making of the library from the constituent files. Invoke `ranlib` to make the library header after the archive is created.

Once the library is created, using it is simple. For example, to load the routine `main.o` with the routines in the library `mylib.a`, use the command sequence:

```
% cc main.o mylib.a
```

You can also access the loader using the FORTRAN compiler, `fc`. This is the preferred method of loading compiled FORTRAN object modules because it does not require enumerating the libraries. For example,

```
% fc -o noral main.o plot.o datain.o -lappl
```

creates the program file `noral` out of object files `main.o`, `plot.o`, and `datain.o`. This program also resolves references to a user library (`/usr/lib/libappl.a`).

# Assembler introduction 7

This chapter briefly introduces important background information about the CONVEX assembler. Specifically, this chapter describes the relationship of the assembler to other CONVEX utilities, compilers, and the CONVEX loader.

The CONVEX assembler operates with the ConvexOS operating system, compilers, macro processors, debuggers, loader, object module libraries, and other auxiliary programs. The operating system is documented in *ConvexOS Man Pages for Users* and *ConvexOS Man Pages for Programmers*.

## Utilities

Utilities typically used with the assembler include the `adb` debugger, the `make` utility, the `rcs` utility, and the `error` utility. The *ConvexOS Man Pages for Programmers* document describes these utilities. The *CONVEX adb Debugger User's Guide* further describes the use of `adb`.

### Assembly-language debugger

The assembly-language debugger, `adb`, traces the execution of assembly-language programs. You can run `adb` on any executable program without reassembly. This debugger is also a useful tool for examining variables, setting breakpoints, and examining core dumps from failed programs.

### make utility

The `make` utility assembles source files as required to create target programs.

### rcs utility

The revision control system, `rcs`, archives revisions of assembly-language source files.

### error utility

The `error` utility merges assembler error messages into the assembler source file, where you can see the errors on the lines at which they occur.

## Compilers

Once compiled, object files from the C compiler, FORTRAN compiler, and assembler can be loaded together for execution as long as you use compatible calling sequences. The FORTRAN compiler and the C compiler invoke the assembler when files that end in `.s` are specified on the compiler command line.

## Loader

You can also invoke the assembler directly, as described in Chapter 13. The assembler accepts an assembly-language source file as input and produces an object module as output. You can then invoke the loader to create an executable file.

## Optimum use of the assembler

To assist you in making optimum use of the assembler, Chapter 15 discusses scheduling instructions, using caches, debugging, using macros and the preprocessor, and writing parallel assembly-language programs.

# Architecture overview

# 8

This chapter describes general features of CONVEX supercomputer architecture that can help assembly-language programmers make informed programming decisions. Specifically, this chapter discusses the following topics:

- Overview of CONVEX supercomputer architecture
- CONVEX register sets
- Fields of the processor status word (PSW)

## CONVEX architecture

CONVEX supercomputers are uniprocessing and multiprocessing systems that incorporate vector processors within their CPUs. The system architecture combines 64-bit integrated scalar and vector processing with large real and virtual memory, high-performance I/O, ConvexOS-supported user applications, and flexible system software.

CONVEX supercomputers have several subsystems that can operate in parallel (including outboard I/0 processors), in addition to a set of 8 general-purpose registers, a set of 8 address registers, and a set of 8 vector registers (each vector register has a length of 128 elements). The CPU(s) can operate on integer data of lengths 8, 16, 32, and 64 bits as well as floating-point data of lengths 32 and 64 bits. A potentially large physical memory (1 Gbyte) with 4096-byte pages complements the large virtual memory (4 Gbytes).

## Parallel processing

The CONVEX C2 and C3 Series architectures provide the operating system and user with a flexible set of instructions for dynamic CPU allocation, deallocation, and communication. Each CPU in a CONVEX multiprocessor operates independently

as a standard CONVEX 64-bit supercomputer. CPUs within a system configured for multiple CPUs share the same physical address space.

The multiprocessor-management hardware tightly couples these CPUs to allow a user application to execute in parallel. Synchronization instructions and CPU control instructions support parallel execution of a process, but do not enforce it. Because these instructions do not enforce parallel execution, you can write code that is independent of the number of CPUs.

The fundamental principle of operation for multiprocessing is that each CPU within a system configuration is responsible for its own scheduling. Each process posts the need for another CPU to join in the computation. An idle CPU recognizes the request and responds to it without the operating system intervening.

## CONVEX register sets

The CONVEX C1 architecture supports the following 3 general register sets:

- Address registers A0 through A7 (8 registers x 32 bits)
- Scalar registers S0 through S7 (8 registers x 64 bits)
- Vector registers V0 through V7 (8 vectors, each 128 elements x 64 bits)

In addition to the register sets listed above, the CONVEX C2 and C3 Series architectures support a set of communication registers that allow for synchronization and communication between threads of a process.

These register sets and the processor status word (PSW) are described in the following sections.

### Address registers

The address registers consist of eight 32-bit registers. The address registers do not support floating-point operations or 64-bit integer arithmetic, whereas the scalar and vector registers support all arithmetic operations.

Address register A0 is the stack pointer (SP), A6 is the argument pointer (AP), and A7 is the frame pointer (FP). For more information on the argument pointer, stack pointer, and frame pointer, refer to Chapter 12, "Calling conventions."

## Scalar registers

The scalar registers consist of eight 64-bit registers. The scalar registers support all arithmetic operations.

## Vector registers

The vector registers consist of 8 vectors, each 128 elements by 64 bits. These registers support all arithmetic operations and speed up program execution by pipelining repetitive operations. To add 128 pairs of numbers, you must specify 6 operations:

- Set vector length register to 128
- Set vector stride
- Vector load—first of each pair
- Vector load—second of each pair
- Vector add
- Vector store—of result

The integrated vector processor processes the operation much more quickly than it processes a loop.

Three special hardware registers support vector processing:

- **Vector length**—A number from 0 to 128 that indicates how many elements of a vector register are to be used.
- **Vector stride**—The number of bytes between successive memory addresses whose contents are loaded or stored into the vector register.
- **Vector merge**—A 128-bit register that indicates which elements of a vector are to participate in some operations.

## Communication registers

The hardware communication registers are a set of high-speed registers used to exchange data between the threads of a process. All threads of a process share the same communication registers.

The CONVEX C2 and C3 Series hardware supports 8 sets of 128 communication registers. Each CPU is linked to a set of communication registers by means of the communication index register (CIR), which is assigned an appropriate value by the operating system.

### Communication register addressing

Each communication register is an addressable 64-bit register. Of the 128 communication registers indexed by the CIR, 64 are reserved for the hardware and operating system and 64 are available for user applications.

Communication register addressing is based on the CIR. That is, communication registers are not addressed as 1024 contiguous registers (8 x 128), but as 8 sets of 128. The 64 communication registers you can access are numbered from 8000 (hex) through 803F (hex).

**Note**

**The first 32 user-accessible communication registers are reserved for compilers and runtime libraries. Use communication registers 8020 (hex) through 803F (hex) for assembly-language applications.**

### Hardware lock bit

Each communication register is associated with a hardware-maintained lock (that is, semaphore) bit. This lock bit synchronizes threads of a process running in parallel in the following manner:

- It checks to see whether an event has completed.
- It checks to see whether a communication register has valid data.
- It controls critical regions of code.

You can manipulate this lock bit independently using synchronization instructions. For more information on how to do this, refer to Chapter 10, the section "Synchronization instructions," on page 93.

## Processor status word (PSW)

The processor status word (PSW) is a 32-bit register that contains coded bits indicating the current state of the processor. This register contains flags that enable or disable exception processing and indicate the results of numerical operations. The PSW contains no privileged mode bits.

Figure 15 illustrates how the bits are arranged in a CONVEX C1 Series PSW.

**Figure 15**
Processor status word—CONVEX C1 Series

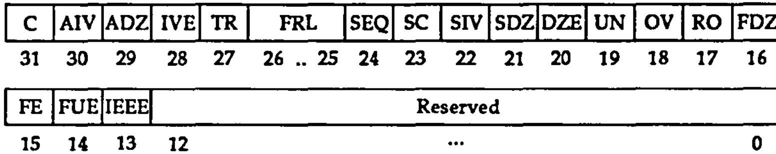
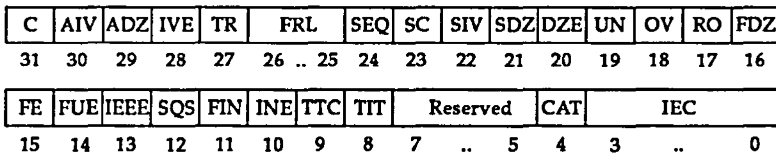



Figure 16 illustrates how the bits are arranged in a CONVEX C2 and C3 Series PSW.

**Figure 16**
Processor status word—CONVEX C2 and C3 Series



# Assembler instruction formats

# 9

This chapter illustrates and describes the format of an assembly-language instruction. Specifically, this chapter defines keywords and discusses the following topics:

- Assembly-language instruction format
- Label field in an assembler instruction
- Operation field in an assembler instruction
- Operand field in an assembler instruction
- Comment field in an assembler instruction
- Terminator field in an assembler instruction

## Instruction format

An assembly-language program is a sequence of instructions and assembler directives. The *CONVEX C Series Architecture Reference Manual* defines the CONVEX instruction set. Chapter 10 in this guide describes assembler directives.

An instruction consists of 5 fields: a label field, an operation field, an operand field, a comment field, and a terminator field. Only the terminator field is required. Figure 17 shows the instruction format and a sample instruction.

**Figure 17**
Instruction format example

```
;[label] [operation]  [operand]  [comment]                  terminator

looptop: ld.w         #1,s0      ;initialize loop counter   <NL>
```

The assembler imposes the following input-formatting restrictions:

- No method to continue a line exists.
- Any number of spaces and tabs can separate fields within the instruction.
- All fields in a single statement must appear on the same source line.

No column boundary restrictions exist.

## Label field

Labels allow instructions to refer to specific addresses within the program. The assembler supports 2 types of labels: name labels and temporary labels. A name label is a user-defined symbolic name. The name `looptop` in Figure 17 is a name label. A name label remains defined throughout the source file that contains the label definition. You cannot redefine a name label. Temporary labels, which function similarly to name labels, are typically used for local branches. You can redefine and reuse temporary labels many times within a source file.

### Name labels

Use name labels to define the targets of branch instructions and variable data locations whose addresses are needed throughout a program.

To define a name label, begin an instruction with a symbolic name followed immediately by a colon, as shown in the label field of Figure 18. When a name label is defined, it assumes the current value of the assembler location counter.

**Figure 18**
Name labels example

```
;Label     Operation     Operand     Comment

start:     st.w          #0,count   ;defined the label "start" and references
                                    ;the label "count"
           .
           .
           .
count:     bs.w          1          ;defines the label "count"
           .
           .
           .
           jmp           start      ;references the label "start"
```

## Temporary labels

Use temporary labels to define instructions that are targets of branch instructions generated by a program such as a compiler or macroprocessor. Temporary labels can be redefined and reused within a source file. These labels have the form *n*`$:`, where *n* is a 32-bit, positive decimal integer. Temporary labels remain defined until you define the next name label, or until you invoke an assembler directive that changes the value of the current location counter. Temporary labels eliminate the need for large numbers of unique name labels, as shown in the label field of Figure 19.

**Figure 19**
Temporary labels example

```
;Label   Operation    Operand    Comment

1$:      add.w        #1,a1      ;defines the temporary label "1$"
         st.w         a1,data
         jmp          1$         ;references the first label "1$"
next:                            ;next name label undefines "1$"
         .
         .
         .
1$:      eq.w         #0,a1      ;another (different) definition of "1$"
         .
         .
         .
         jmp          1$         ;references the second label "1$"
```

Similar to name labels, temporary labels assume the current value of the location counter when they are defined.

## Operation field

The second instruction field is the operation field. The operation field contains a mnemonic corresponding to the operation to be performed. Operations are of 2 types: instruction mnemonics and assembler directives. The following sections briefly describe instruction mnemonics and assembler directives.

### Instruction mnemonics

Instruction mnemonics are symbolic names for machine instructions. An instruction mnemonic in the operation field of a statement causes the assembler to generate the binary representation for that machine instruction with correct codes for the operand. For more information on instruction mnemonics, refer to Chapter 10, "Op codes."

### Assembler directives

Assembler directives are commands to the assembler that either alter its operation or cause it to reserve storage for data values. Assembler directives do not generate binary instructions. For more information on assembler directives, refer to Chapter 10, "Op codes."

## Operand field

If the operand field is present, it consists of one or more operands separated by commas. The number and type of operands present depend on the operation being performed. For example, the number and type of operands required for the `add` instruction mnemonic depend on the register set being used. To determine the number of operands for a particular instruction, refer to the instruction descriptions in the *CONVEX C SERIES Architecture Reference Manual*. You can use spaces or tabs between operands, but not within an individual operand.

The operand field generally contains 1 or more operands that specify a machine register or a memory location using one of the computer addressing modes. For further information, refer to Chapter 11, the section "Addressing modes," on page 122.

## Comment field

The comment field is optional and typically describes the operations being performed. The comment field begins with a semicolon at any location within the input line, and ends with a statement terminator (which also ends the statement). A comment can contain any sequence of characters as long as the sequence does not contain one of the termination characters. The assembler ignores the characters in the comment field.

## Terminator field

To end an assembly-language statement, use one of 3 termination characters: newline <nl>, form feed <ff>, or ! . Use the ! character to separate multiple statements entered on a single input line. The terminator field is the only required field in an assembly-language statement.

# Op codes 10

This chapter describes machine op codes and pseudo-operations. Specifically, this chapter discusses the following topics:

- Instruction typing
- Extended op codes
- Machine op codes, including
  - Memory-reference instructions
  - Calculation of effective addresses
  - Memory-reference instruction format
  - Arithmetic instructions
  - Logical instructions
  - Data-type specification
  - Data-conversion instructions
  - Vector-reduction instructions
  - Operations under mask
  - Program-control instructions
  - Span-independent program-control instructions
  - Machine-control instructions
  - Synchronization instructions
  - CPU control instructions
- Pseudo-operations, including descriptions of assembler directives

## Instruction typing

Not all instructions in the CONVEX assembly-language instruction set run on every machine in the CONVEX family of supercomputers. Each instruction is tagged within the assembler to be of a certain type. The *type* of an instruction specifies the machine that accepts the instruction. The types of all instructions in one assembled file are accumulated in the "flags word" of the executable file's header. The kernel examines these flags to determine whether the machine can or cannot accept the file.

## Extended op codes

An op code is the sequence of bits in an instruction that determines the operation to be performed. Extended op codes are 1- to 3-halfword CONVEX op codes prefixed with a 16-bit modifier. This modifier is called a *prefix op code*. The prefix op code has a bit (called the E bit) set to 1 (true) or 0 (false) that modifies the behavior of the instruction. For example, following are 3 instructions: a standard op code (prefixed ST) and 2 extended op codes (prefixed E1 and E0).

| Mnemonic | Hexadecimal | Binary |
|---|---|---|
| eq.b Vj,Vk | 0x6800 | ST 0110100000 |
| eq.b.t Vj,Vk | 0x7EF8 0x6800 | E1 0110100000 |
| eg.b.f Vj,Vk | 0x7EF0 0x6800 | E0 0110100000 |

The two prefix op codes are defined in hexadecimal as 0x7EF8 and 0x7EF0.

The extended op code format allows for the addition of many new instructions, a majority of which are vector operations. The CONVEX C200 Series architecture supports these vector instructions by recognizing *operations under mask*, which employ the extended op code format. For further information on these instructions, refer to the section "Operations under mask," on page 89 of this chapter.

For further information on extended op codes, refer to the *CONVEX C Series Architecture Reference Manual*.

# Machine op codes

When specified, the operation field of an instruction contains a mnemonic. This mnemonic is the symbolic name for a binary machine instruction. The assembler uses the instruction mnemonic, the operand types, and possibly a data-type specification to determine the appropriate bit pattern required for the instruction.

The CONVEX instruction set is divided into the following categories:

- Memory-reference instructions
- Arithmetic instructions
- Logical instructions
- Data-conversion instructions
- Vector-reduction instructions
- Program-control instructions
- Span-independent program-control instructions
- Machine-control instructions
- Synchronization instructions
- CPU control instructions

The following sections describe these instructions and requisite information.

## Memory-reference instructions

Use memory-reference instructions to load registers from memory, store the contents of registers in memory, and modify memory. Table 9 lists memory-reference instructions.

**Table 9**
Memory-reference instructions

| Instruction | Description |
| --- | --- |
| `ld` | Load register from memory |
| `ldea` | Load the effective address into an address register |
| `ldvi` | Load a vector register with a vector of indices |
| `pop` | Pop the top element on the stack into a register |
| `psh` | Push register onto memory stack |
| `pshea` | Push the effective address onto the stack |
| `stvi` | Store a vector register with a vector of indices |
| `tac` | Test and clear a memory location |
| `tas` | Test and set a memory location |

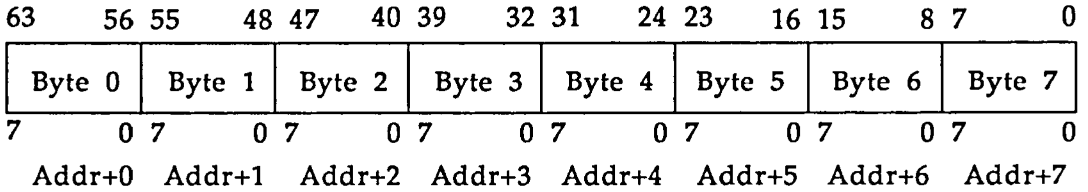
## Effective address calculation

Memory-reference instructions calculate an effective address by adding the value found in the displacement field of the instruction (see Figure 21) to the contents of the A register specified in the instruction. The A register is referenced by the A*j* field. When no register is specified or when A0 is specified, 0 (instead of A0) is added to the value of the instruction displacement field.

Memory-reference instructions may specify indirect addressing. When indirect addressing is used, the calculated effective address is not a final address, but a reference to a location in memory that contains the final effective address.

Instructions that reference memory are usually 32, 48, or 64 bits long. The difference in length is caused by a difference in the length of the displacement field and the presence or absence of the prefix halfword. Figure 20 illustrates the ordering of each addressable entity within a 64-bit longword.

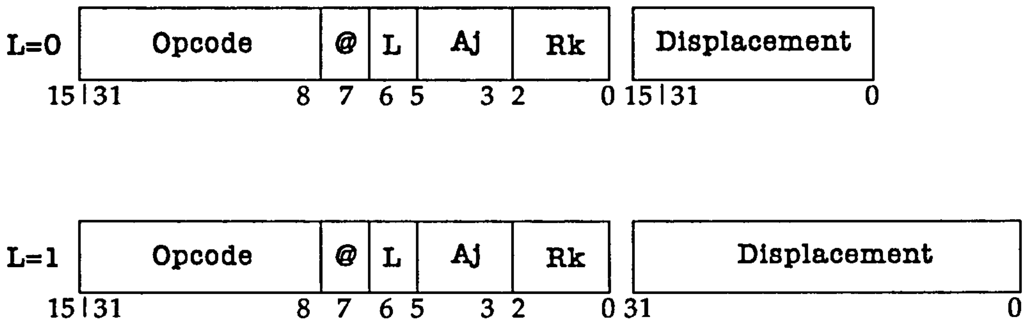
**Figure 20**
Memory longword structure




## Memory-reference instruction format

Figure 21 shows the structure of memory-reference instructions.

**Figure 21**
Memory-reference instruction format



The fields in Figure 21 are defined as follows:

Op code = bits<15..8>

Specifies which operation to perform on the referenced data, and selects the register class (A, S, or V) referred to by the R*k* field.

bit<7> - Indirection

Specifies the existence or absence of indirection. If @=0, no indirection is specified; the effective address is the sum of the contents of A*j*, plus the displacement. If @=1, indirection is specified; the effective address is the address stored at the location computed by adding the contents of A*j* to the displacement.

L = bit<6> - Length

Specifies the length of the displacement field. If L=0, the displacement field is a two's complement 16-bit integer. This field is sign-extended to 32 bits before it is added to the effective address. The assembler generates the instruction length automatically. The length is based on the displacement.

Aj = bits<5..3>

Specifies the `A` (address) register used to generate the logical address ($0 \leq j \leq 7$). If you specify `A0`, the value of 0 is used as the contents of `A0` for absolute addressing. The true contents of `A0` are unused. If you specify `A1`-`A7`, the contents (all 32 bits) of the specified address register are added to the displacement field to generate the final effective address of the target address. The first byte of the target operand is referenced. If indirection is specified, the effective target address references byte 0 of a 32-bit word.

Rk = bits<2..0>

Specifies the source or destination of the referenced operand ($0 \leq k \leq 7$). The op code specifies the precision, data type, and structure of the operand. `R`*k* can be either a scalar register (`S`*k*), an address register (`A`*k*), or a vector register (`V`*k*), depending on the instruction.

Displacement

This field contains a 16-bit or 32-bit value that is either added to the entire contents of an `A` register, or used directly as a byte address. A 16-bit displacement value is sign-extended to 32 bits before use.

## Arithmetic instructions

Use arithmetic instructions to perform arithmetic operations on the contents of machine registers and to store the results of an arithmetic operation in a machine register. Table 10 lists arithmetic instructions.

**Table 10**
Arithmetic instructions

| Instruction | Description |
|---|---|
| `add` | Add the operands |
| `div` | Divide the operand |
| `eq` | Compare equal operands |
| `le` | Compare less than or equal operands |
| `leu` | Compare less than or equal unsigned operands |
| `lt` | Compare less than operands |
| `ltu` | Compare less than unsigned operands |
| `mul` | Multiply the operands |
| `ne` | Compare not equal operands |
| `neg` | Negate the operand |
| `sub` | Subtract the operand |

## Logical instructions

Use logical instructions to perform logical operations on the specified register operands. Table 11 lists logical instructions.

**Table 11**
Logical instructions

| Instruction | Description |
|---|---|
| and | Perform the logical product on the operands |
| lop | Determine bit position of left-most 1 bit |
| mov | Copy source operand |
| not | Perform the one's complement on the operands |
| or | Perform the logical sum on the operands |
| plc | Count the population (number of 1s) in the operand |
| shf | Logically shift contents of the register |
| tzc | Count trailing 0 bits |
| xor | Perform the exclusive-OR operation on the operands |

## Data-type specification

Many instructions operate on data of a specific width or size. Most CONVEX instruction mnemonics include a data-type specifier along with the basic instruction name. The width specifier consists of a period followed by a single letter that specifies the desired operand size. For example, `ld.w` is a load (`ld`) instruction whose data size, `.w`, is a 32-bit machine word. (Several data-type specifiers also include a second letter that specifies either the upper or lower half of a 64-bit constant.) Table 12 lists data-type specifiers for CONVEX assembly language.

**Table 12**
Data-type specifiers

| Specifier | Description |
|---|---|
| .b | Byte data (8 bits) |
| .d | Double-precision floating-point data (64 bits) |
| .dl | Lower word of a 64-bit double-precision floating-point constant |
| .du | Upper word of a 64-bit double-precision floating-point constant |
| .h | Halfword data (16 bits) |
| .l | Longword data (64 bits) |
| .ll | Lower word of a 64-bit longword constant |
| .lu | Upper word of a 64-bit longword constant |
| .s | Single-precision floating-point data (32 bits) |
| .u | Upper word of longword (32 bits) |
| .w | Word data (32 bits) |
| .x | Extended data (128 bits) |

## Data-conversion instructions

Use data-conversion instructions to convert data from one format to another. The CONVEX instruction set does not provide a complete set of conversion combinations, so you may find it necessary to use data-conversion instructions in combinations. Table 13 lists data-conversion instructions.

**Table 13** Data-conversion instructions

| Instruction | Description |
|---|---|
| `cvtb.w` | Convert a byte to a word |
| `cvtd.l` | Convert a double-precision floating-point number to longword format |
| `cvtd.s` | Convert a double-precision floating-point number to single-precision format |
| `cvth.w` | Convert a halfword to a word |
| `cvtl.d` | Convert a longword to double-precision floating-point format |
| `cvtl.s` | Convert a longword to single-precision floating-point format |
| `cvtl.w` | Convert a longword to a word |
| `cvts.d` | Convert a single-precision floating-point number to double-precision format |
| `cvts.l` | Convert a single-precision floating-point number to longword format |
| `cvts.w` | Convert a single-precision floating-point number to a word |
| `cvtw.b` | Convert a word to a byte |
| `cvtw.h` | Convert a word to a halfword |
| `cvtw.l` | Convert a word to a longword |
| `cvtw.s` | Convert a word to single-precision floating-point format |

## Vector-reduction instructions

Use vector-reduction instructions to reduce vector operands to a scalar value. Table 14 lists these instructions.

**Table 14**
Vector-reduction instructions

| Instruction | Description |
|---|---|
| `all` | Perform the logical product (AND) operation on the elements of a vector |
| `any` | Perform the logical summation (OR) operation on the elements of a vector |
| `max` | Find the maximum element of a vector |
| `min` | Find the minimum element of a vector |
| `parity` | Perform the exclusive-OR operation on the elements of a vector |
| `prod` | Perform the product (Π) operation on a vector |
| `sum` | Perform the summation (Σ) operation on a vector |

## Operations under mask

Most vector instructions can operate under mask. These instructions use the extended op code format; that is, they are prefixed with a halfword modifier to operate under a true or false mask (refer to the section, "Extended op codes," on page 80).

Operations under mask work as follows: the vector-merge (VM) register has a bit associated with each vector register element. When an operation is performed under mask, each element is either included or excluded from the operation based on the state of its corresponding VM bit. Operations under mask assume two forms:

- True—Elements with the VM bit set to 1 are included in the instruction; these instructions have a `.t` suffix.
- False—Elements with the VM bit set to 0 are included in the instruction; these instructions have a `.f` suffix.

Following are 2 examples of operations under mask—`add.w.t` and `add.w.f`. In these examples, assume the following initial values:

```
V0=0 1 2 3 4 5   VL=6
V1=6 7 8 9 2 3   VM=0 1 1 0 0 1
V2=5 5 5 5 5
```

Performing an `add.w.t V0,V1,V2` produces the following:

```
V2=5 8 10 5 5 8
```

Performing an `add.w.f V0,V1,V2` produces the following:

```
V2=6 5 5 12 6 5
```

## Program-control instructions

Use program-control instructions to call and return from subroutines and to jump or branch to specific locations or labels. Table 15 lists these instructions. The term "PC" means program counter.

**Table 15** Program-control instructions

| Instruction | Description |
|---|---|
| `bkpt` | Perform a trap for the debugger |
| `br` | Perform a PC-relative branch |
| `bra.f` | Perform a PC-relative branch if address carry is false |
| `bra.t` | Perform a PC-relative branch if address carry is true |
| `bri.f` | Perform a PC-relative branch if interrupt-enable flag is false (off) |
| `bri.t` | Perform a PC-relative branch if interrupt-enable flag is true (on) |
| `brs.f` | Perform a PC-relative branch if scalar carry is false |
| `brs.t` | Perform a PC-relative branch if scalar carry is true |
| `call` | Call a subroutine in long format |
| `callq` | Call a subroutine in quick format |

**Table 15 (continued)** Program-control instructions

| Instruction | Description |
|---|---|
| calls | Call a subroutine in short format |
| jmp | Jump to the specified location |
| jmpi.f | Jump on interrupt-on false |
| jmpi.t | Jump on interrupt-on true |
| jmpa.f | Jump if address carry is false |
| jmpa.t | Jump if address carry is true |
| jmps.f | Jump if scalar carry is false |
| jmps.t | Jump if scalar carry is true |
| pbkpt | Force a process breakpoint |
| rtn | Return from a subroutine |
| rtnq | Return from a subroutine that was called with a `callq` instruction |
| sysc | Perform a system call |

## Span-independent program-control instructions

Use span-independent program-control instructions to access generic forms of the jump and branch instructions. The assembler resolves span-independent program-control instructions into either jump or branch instructions, depending on the value of the displacement field in the instruction.

Instructions beginning with `jmp` transfer program control to the absolute address specified in the address portion of the instruction. Instructions beginning with `br` determine the address to branch to by adding or subtracting from the current value of the program counter. The quantity added or subtracted is passed to the instruction as an argument.

Span-independent instructions begin with `jbr` and accept absolute addresses, relative addresses, or labels as arguments. If you use labels with a `jbr` instruction, the assembler calculates the relative distance to the location specified and the width of displacement to be used. The assembler also generates the appropriate form of either `jmp` or `br` when resolving the `jbr` instructions. The assembler changes displacement widths when you change locations or when you modify the program.

Table 16 lists span-independent program-control instructions.

**Table 16**
Span-independent program-control instructions

| Instruction | Description |
|---|---|
| `jbr` | Jump or branch to the specified location |
| `jbra.f` | Jump or branch if the address carry is false |
| `jbra.t` | Jump or branch if the address carry is true |
| `jbri.f` | Jump or branch if the interrupt-enable flag is false (off) |
| `jbri.t` | Jump or branch if the interrupt-enable flag is true (on) |
| `jbrs.f` | Jump or branch if the scalar carry is false |
| `jbrs.t` | Jump or branch if the scalar carry is true |

## Machine-control instructions

Use machine-control instructions to control the various units within the machine. These instructions are used principally by the operating system and are of little use in normal system and application programs. Table 17 lists these instructions.

**Table 17**
Machine-control instructions

| Instruction | Description |
|---|---|
| dsi | Disable interrupts |
| eni | Enable interrupts |
| exit | Error exit of a program |
| halt | Halt the processor |
| ldkdr | Load the kernel segment-descriptor register |
| ldsdr | Load a segment-descriptor register |
| mski | Mask interrupt |
| nop | Perform no operation |
| pate | Purge a single ATU entry |
| patu | Purge the ATU |
| pich | Purge the instruction cache |
| plch | Purge the logical cache |
| xmti | Transmit interrupt |

Some of these instructions are *privileged* instructions that cannot be used in user programs. Refer to the *CONVEX C Series Architecture Reference Manual* for detailed information on privileged instructions.

## Synchronization instructions

Recall that communication registers allow for communication between threads of a process (refer to the section "Communication registers," on page 69). These registers control the process flow after a process divides into threads. Synchronization instructions let you manipulate these communication registers to manage the execution of parallel programs.

In particular, two sets of synchronization instructions manage communication registers:

- lck (lock) and `ulk` (unlock) instructions
- snd (send) and `rcv` (receive) instructions

These synchronization instructions use the address carry bit (C) or the scalar carry bit (SC) in the PSW to indicate success or failure. The following sections discuss these instructions in more detail.

### lck and `ulk`

The `lck` and `ulk` instructions use the hardware `lock` bit as a semaphore (refer to the section "Communication registers," in Chapter 8). The `lck` instruction sets the lock bit. If the lock bit is clear, then the address carry bit (C) is set. If the lock bit is already set, then C is cleared to indicate failure.

The `ulk` instruction clears the lock bit, and C is set to the previous value of the lock bit.

The `lck` and `ulk` instructions can be used to:

- check to see whether or not an event has completed
- define critical regions of code

### snd and `rcv`

The `snd` instruction moves data from an address (A) or scalar (S) register to a communication register and sets C or SC if the communication register is unlocked. If the communication register is locked, `snd` leaves the communication register unchanged and clears C or SC, depending on whether an A or S register was specified.

The `rcv` instruction moves data from a communication register to an A or S register and sets C or SC if the register is unlocked. If the register is locked, then C or SC is cleared.

## Note

**The `snd` and `rcv` instructions automatically handle locking. You do, however, need to supply branch-back instructions to handle instances when `snd` and `rcv` temporarily fail because of the action of another thread.**

The `snd` and `rcv` instructions use the hardware lock bit to control critical regions of code and determine whether a communication register has valid data A communication register is locked when it contains valid data.

Table 18 lists these and other synchronization instructions used to manipulate communication registers. For further information on these instructions, refer to the *CONVEX C Series Architecture Reference Manual*.

**Table 18**
Synchronization instructions

| Instruction | Description |
|---|---|
| inc | Increment a communication register by another register and return the new value |
| lck | Lock a communication register |
| popr | Pop a word from a resource structure at the effective address into an address register |
| pshr | Push an address register onto the resource structure at effective address |
| rcv | Move the contents of a communication register into a register |
| rcvr | Move the contents of a synchronized resource structure in memory into a register |
| snd | Send the contents of a register to a communication register |
| sndr | Send the contents of a register to a synchronized resource structure |
| ulk | Unlock a communication register |

For an illustration of how these instructions can be used, refer to the section "Sample parallel program," on page 159, and the section "Parallel programming in CONVEX assembly language," on page 171.

## CPU control instructions

Allocating CPUs in the CONVEX multiprocessor architecture is defined in terms of a fork. A fork is a request to the CPU. CPU control instructions allow you to:

- Post a fork, which requests a CPU to share in the computation of a single process
- Clear a fork
- Force the current CPU back into an idle state
- Force a multithreaded process back into a single thread

CPU control instructions essentially offer two programming schemes. One scheme uses `pfork`, `wfork`, and `cfork` instructions. The other scheme uses `spawn` and `join` instructions. These programming schemes represent different ways of synchronizing resources and allocating available CPU time. For more information on these programming schemes, refer to the section "Parallel Programming in CONVEX

Assembly Language," on page 171, and the *CONVEX C Series Architecture Reference Manual*. The following sections briefly describe these instructions.

### pfork, `wfork`, and `cfork`

The `pfork` instruction requests that one additional thread start executing. The new thread:

- Begins at a PC of the effective address
- Uses the stack pointer specified in *Ak* field of the `pfork` instruction
- Inherits the PSW, FP, and AP of the parent thread

The `wfork` instruction causes the thread to terminate and immediately begins executing a *pending* thread. A pending thread is a thread request (in this case, a `pfork` request) that has been posted and is waiting to be accepted and processed by an available CPU. The `wfork` instruction does not, however, clear pending threads. The `cfork` instruction clears pending threads.

### spawn and `join`

The `spawn` instruction requests that as many threads start executing as there are processors available. The new threads:

- Begin at a PC of the effective address
- Use the stack pointer specified in *Ak* field of the `spawn` instruction
- Inherit the PSW, FP, and AP of the parent thread

The `join` instruction reduces the process to a single thread of execution. Threads in a process reach a `join` and terminate their work. Each CPU terminates its thread of execution and either returns to an idle state or continues execution after the `join` as a single-threaded process.

Table 19 lists these and other CPU control instructions used to request other CPUs to assist in the computation of a single process.

**Table 19**
CPU control instructions

| Instruction | Description |
|---|---|
| `cfork` | Clear a fork |
| `idle` | Attempt to accept a posted fork in the specified communication index register (CIR); if a fork is not posted, idle the current CPU without deallocating the current thread |
| `join` | Force all threads created by a process to join at a single execution point |
| `pfork` | Post the need for a CPU to assist in the computation of a process |
| `spawn` | Post the need for as many CPUs as possible to assist in the computation of a process |
| `wfork` | Terminate a thread, idle the current CPU, and look for any posted forks |

For further information on these instructions, refer to the *CONVEX C Series Architecture Reference Manual.*

## Pseudo operations (directives)

Pseudo-operations are assembler directives that either alter the assembler's operation or cause the assembler to reserve storage for data values. The syntax of these directives is similar to the instruction set syntax.

This section describes the following directives:

- Program-segment directives
- Storage-allocation directives
- Alignment directives
- Floating-point directives
- External symbolic-name directives
- Thread-private directives
- Symbol-table directives
- Procedure summary (psum) directives

## Program-segment directives

Address space in ConvexOS programs is divided into 6 sections, called *program segments*. A program segment is a block of code identified by user-defined program-segment directives. The directives in Table 20 identify the type of data included in each program segment.

**Table 20**
Program-segment directives

| Directive | Data type |
|---|---|
| `.bss` | Uninitialized data |
| `.cdata` | Initialized common block |
| `.data` | Initialized data |
| `.tbss` | Uninitialized thread data |
| `.tdata` | Initialized thread data |
| `.text` | Code |

The assembler begins generating instructions and data in the `.text` segment. The program cannot write into the `.text` segment (read-only data can be stored in this segment). The program may, however, write into the `.data`, `.tdata`, `.bss`, `.tbss`, and `.cdata` segments.

The `.tdata` and `.tbss` directives are specifically designed to create thread data segments with unshared memory. Use these directives when you need to write thread memory to the same virtual address, as in a parallelized loop. Unshared thread memory is managed by the thread identifier (TID) register, which makes threads unique in the translation from virtual to physical memory. Data within `.tdata` and `.tbss` directives are mapped to `.data` and `.bss` when run on a CONVEX C1 system.

Threads with unshared memory may require separate stack spaces. Programs that do not have separate stack spaces for different threads are difficult to debug. To create an individual thread stack for uninitialized thread data, use the `.tbss` directive to allocate data space and then assign a thread SP to that space. Refer to the *CONVEX CXdb User's Guide* for more information on how to debug these types of problems.

The .cdata segment differs from the other program segments in that the current program counter for each .cdata segment is set back to zero, while the program counter is always incremented for the other segments. You must precede each .cdata directive with a .comm directive, that allocates the space to be filled by the .cdata segment (.comm directives are discussed in detail in the section, "External symbolic-name directives," on page 107). You can use any number of .cdata segments in your program, as long as they each follow a .comm directive.

The .cdata segment generates a fatal warning if it exceeds the space allotted by the .comm directive. If the .cdata segment exceeds the space allotted by the .comm directive, adjust the .comm directive to the correct size and reassemble the program.

If multiple .cdata directives exist for the same location within one file, the assembler checks to make sure that initialized locations are not reinitialized or partially overwritten. Do not reinitialize variables within a common block, or a fatal assembler error will occur. If multiple .cdata directives exist among several files, the loader overlays multiple initialized common sections, then issues a warning.

Figure 22 shows the use of these program segments.

**Figure 22**
Use of program segments

```
            .text                  ;current section is .text
;
;  Assembly is taking place in the text
;  section.
start:      ld.w   #1000,a1
            ld.w   data,a2

            .data                  ;current section is .data
            .comm  page, 48        ;allocate space for .cdata
            .cdata page            ;current section is .cdata
;
;  Set up an initialized data table
;
data:       ds.w 1,2,3,4,5,6,7,8,9,10,11,12

            .bss                   ;current section is .bss
;
;  Set up uninitialized data table
;
udata:      bs.w    1000
;
;  Now back to the text section
;
            .text                  ;current section is .text
;
;  Continue with the program code
            mov     a1,a3
```

These program segments can be subdivided into 4 subsections (0, 1, 2, or 3). Use these subsections to specify code and data in a convenient order. The assembler collects code or data generated in each subsection in the order in which it appears. Format the program-segment directives to specify subsections as follows:

`.bss` *n*

`.cdata` *name*

`.data` *n*

`.tbss` *n*

`.tdata` *n*

`.text` *n*

where *n* is 0, 1, 2, or 3, or an absolute symbolic name whose value is 0, 1, 2, or 3. The subsection default is 0. Subsections are output in increasing numeric order. For the `.cdata` directive, *name* is the name of a common block, and you can use any number of these.

## Storage directives

To allocate storage, use `bs`, the block-storage directive, and `ds`, the define-storage directive. Note that storage is not automatically aligned. For information on storage alignment, refer to the section, "Alignment directives," on page 104.

The following sections describe the use of `bs` and `ds` directives.

### Uninitialized storage allocation

When you specify the `bs` directive, the assembler allocates a specified number of storage locations of a particular width. The format of the directive is as follows:

```
bs.x    n
```

where *x* is one of the data-type specifiers `b`, `h`, `l`, `w`, `s`, or `d` (see Table 12). The single operand *n* represents the number of units of the specified width *x* to be allocated. If the directive has a label, the value assigned to the label is the address of the leftmost byte of the first unit.

The following example illustrates the use of the `bs` directive.

```
abc:    bs.w    50      ;allocate 50 words
xyz:    bs.l    10      ;allocate 10 longwords
```

### Initialized storage allocation

When you specify the `ds` directive, the assembler allocates and initializes storage locations. Within the directive, you specify the number of locations to allocate and their initial values. The format of the directive is as follows:

```
ds.x    operand [,...]
```

where *x* is one of the data-type specifiers `b`, `h`, `w`, `l`, `s`, or `d` (see Table 12), and *operand* is a numeric constant, symbolic name, character constant, or string constant. Each operand is used, in turn, to initialize items of storage whose size is specified by *x*.

The operands following the ds directive define the initial values assigned to the storage locations. The directives that specify either single-precision or double-precision floating-point numbers should specify operands that are floating-point numbers. The halfword, word, and longword directives require integers as operands. The ds.b directive accepts either strings or integers as operands, which lets you store messages. The example in Figure 23 illustrates the correct use of the w, s, and b operands with the ds directive.

**Figure 23**
Examples of the ds directive

```
ABC:     ds.w     0x1000,200,5           ;define and initialize word storage

xyz:     ds.s     458.3,100.0            ;define and initialize 2 single
                                         ;precision storage locations
msg:     ds.b     "Message",012,015      ;define a message followed by
                                         ;a CR and LF
```

The ds directive does not override the characteristics of the program segment within which it is defined. Therefore, using this directive in the .text segment allocates and initializes the number of locations specified, while using the ds directive within the .bss or .tbss segment generates an error message.

The following examples illustrate the use of this directive.

```
var1:   ds.w  5000  ;allocate a word
                    ;containing the number 5000
var2:   ds.w 0,0,0  ;allocate 3 words, each 0
```

The initialization arguments may include a repeat factor that initializes blocks of storage with a single value. You can specify the repeat factor as either a numeric value or the value of an absolute local symbolic name. In the latter case, precede the data operand with the symbolic name specified as a repeat factor, and enclose the data operand in parentheses.

The following examples illustrate these two uses of the repeat factor.

```
table:    ds.w 1000(0)    ;allocate 1000 words
                          ;of zero

foo=100
tiny:     ds.b foo(0x20)  ;allocate 100 bytes
                          ;of 0x20
```

### Strings

A string is a sequence of characters that is delimited by quotation marks. Strings cannot span lines in the source program. Use strings only as arguments to a storage-definition directive. The following list contains examples of valid strings:

```
"CONVEX Assembler"

"A message."

"Error number 20"
```

The assembler allows you to specify escape sequences within strings. These are similar to certain C escape sequences. Table 21 lists valid assembler escape sequences.

**Table 21** Assembler escape sequences

| Description | Assembler notation |
|---|---|
| Backslash | \\ |
| Backspace | \b |
| Bit pattern | \ddd |
| Carriage return | \r |
| Double quote | \" |
| Form feed | \f |
| Horizontal tab | \t |
| Newline | \n |
| Single quote | \' |

The escape sequence `\ddd` consists of the backslash and 1 to 3 octal digits specifying the value of the desired character. For example, \0 indicates the character NUL. When the character following a backslash is not one of those specified in Table 21, the backslash is ignored.

Assembler strings, unlike C strings, are not automatically terminated with a null byte. You can use null bytes as terminators for string constants by including the \0 escape sequence as the final character in the string.

## Alignment directives

Code and data in a module may have to be aligned to a specific byte-address multiple within the program address space. For example, all instructions must begin on even address boundaries, and certain ConvexOS data structures must begin on 4096-byte boundaries. Also, many memory references operate most efficiently when the referenced data are aligned on 32-bit or 64-bit address boundaries. These alignments, as described in this section, can be specified explicitly in the assembly-language program. Both the assembler and the loader coordinate to maintain the program alignments that you specify.

The `.align` directive specifies the byte multiple to which any subsequent output in the current program section is aligned. For example:

```
.align 2
```

adjusts the location counter of the current program segment for halfword alignment so that the low-order bit of the location counter is zero. An alignment of 2 bytes at the beginning of the code portions of a module remains throughout the remainder of the module, because all instructions are multiples of 16 bits in length. Data segments within the program may need to be aligned so that program variables do not begin on inefficient address boundaries. In most cases, you achieve maximum efficiency when data are given an alignment equal to the size of the item in bytes. For example, 2-byte data should be aligned on 2-byte boundaries, and 4-byte data on 4-byte boundaries. However, 8-byte data on CONVEX C1 and C2 Series architectures need only be aligned on 4-byte boundaries, but on the CONVEX C3 Series architectures, 8-byte data should be aligned on 8-byte boundaries. Each segment is initially aligned on an 8-byte boundary.

The assembler handles alignment specifications of 8 bytes or fewer by padding the output for the current segment with bytes of zeros. The loader automatically preserves alignments of 8 bytes or less, and manages alignments greater than 8 bytes. The assembler passes these large alignments to the loader for processing. You may specify only one alignment of greater than 8 bytes for each segment of each module. This alignment must appear in the source file before any code or data to be generated in that segment. Once code or data has been generated in a segment, an alignment directive greater than 8 is meaningless.

The `.pad` directive moves the program counter to the next multiple of its argument (assuming an origin of zero). For instance:

```
.pad 1024
```

moves the program counter to the next multiple of 1024. If you pad to greater than a longword (8 bytes), you must use the `.align` directive in conjunction with the `.pad` directive to make certain the boundary is appropriate for that module. Using the `.align` *n* directive forces the loader to load that section on a multiple of *n*.

For further information on storage alignments of data types, refer to the *CONVEX C Guide*.

## Floating-point directives

The `.fpmode` directive controls the translation of floating-point constants within your assembly code. The directive takes one operand, either `ieee` or `native`. Any other operands are invalid and do not change the floating-point mode currently in effect. Figure 24 is an example of how to use the `.fpmode` directive:

**Figure 24**
Example of the `fpmode` directive

```
                         .
                         .
                         .
.fpmode ieee             ;translate following floating-point
                         ;constants to IEEE format

.data
ds.d 3.415               ;translated to IEEE format

.fpmode native           ;translate following floating-point
                         ;constants to native format

ld.s #1.0,s0             ;translated to native format
```

The initial floating-point mode of the assembler is determined by the floating-point-format option specified on the command line. The `-fi` option specifies IEEE mode, and the `-fn` option specifies native mode. If no `-f` option is specified, the site default is used.

The assembler, not the `.fpmode` directive, sets the execution mode of the object file produced from your source code. The system manager determines the site default when running the `sysgen` program to generate the system. (Refer to the chapter "Customizing kernel boot-time parameters" in *Managing ConvexOS: Configuration Guide* for details on system defaults.)

Single-mode programs (all IEEE or all native) never need a `.fpmode` directive; the mode specified by the floating-point-format option determines the translation mode of all floating-point constants in the file. You may use `.fpmode` directives in single-mode programs as long as the directive agrees with the mode specified on the command line. In single-mode programs, a fatal error occurs when the argument to `.fpmode` differs from that given with the floating-point-format option.

It is good practice to state explicitly the intended mode for the resulting object (and executable) files with the floating-point-format option (`fi` or `fn`) rather than to make assumptions about the machine on which the file is assembled. The default mode is machine-dependent.

## External symbolic-name directives

Complete programs often reside in separately assembled modules that the loader later combines into a single program. Two external symbolic name directives (`.globl` and `.comm`) let the assembled modules communicate. These directives define symbolic names that are declared and used in separate modules.

The assembler and loader support external and local symbolic-names. External symbolic names are global symbolic names visible to other modules, and are used to resolve external references in other modules. Local symbolic names are invisible to other modules; they are known only within the current assembly object file.

Source programs define the values for external symbolic names by declaring the symbolic name to be global in one object module. Other object modules simply refer to the symbolic name. Any symbolic name that the assembler cannot resolve within a source file is treated as an unresolved external symbolic name.

### .globl directive

The `.globl` directive is usually used for function or subroutine names. It notifies the assembler that the symbolic names following are declared to be visible in other assembly modules. The format of the `.globl` directive is as follows:

```
.globl name [ ,...]
```

The symbol *name* refers the variable referenced later in your program. Symbolic names specified in `.globl` directives are defined in the current module. The loader resolves external references to global symbolic names during the linking process. You can define 20 symbolic names per statement using a comma-separated list—more than that returns the error

```
Wrong  number of operands.
```

The next example illustrates the use of the `.globl` directive.

```
    .globl abc      ;declare the sybolic name "abc"
                    ;as globally visible
    .text
abc:
    sub.w #4,sp     ;routine entry point is global
```

In the preceding example, another assembly-language source file could refer to the symbolic name `abc` without any special declaration. The assembler treats any symbolic names in a source file that are referenced, but not defined, as unresolved external symbolic names.

### .comm directive

The `.comm` directive is usually used for data shared between program units. Use the `.comm` directive to define external symbolic names of storage areas. This directive reserves storage in the `.bss` program section, for example, for FORTRAN `COMMON` areas. The loader resolves the common storage directives by allocating storage in the `.bss` section. The size of the allocated area is the maximum size specified in any module that contains a definition for the given symbolic name. If more than one `.comm` directive appears in a file, the largest size defined is the size for that common block.

The `.comm` directives can appear anywhere in your source code. You must, however, precede a `.cdata` directive with a `.comm` directive. The `.comm` directive allocates the space filled by the `.cdata` segment. A `.comm`/`.cdata` pair can appear in one file, and a `.comm` directive for the same common block can appear in a separate file (that is, the common block is declared and initialized in one FORTRAN file and referenced in another). When this occurs, the loader determines that the common block belongs in the `.data` section and is really an initialized common block.

The format of the `.comm` directive is as follows:

```
.comm name, expression
```

where *expression* defines the size of the common area in bytes and must be an absolute constant. *name* refers to the symbolic name defined as the name of the common storage block. Symbolic names used in `.comm` directives can be redefined in other `.comm` directives. The size of the generated common block is the maximum size declared in any module.

## Thread-private directives

There are two thread-private pseudo directives: `.tcdata` and `.tcbss`. These directives affect the allocation of unshared thread data areas. Memory allocated by the `.tcdata` directive is put in the `.tdata` program segment, while memory allocated by the `.tcbss` directive is put in the `.tbss` program segment. In

general, the `.tcdata` directive is a thread-private version of the `.comm` directive; it allocates a common data segment for a thread. The `.tcbss` directive is a thread-private version of the `.bss` directive.

The `.tcdata` directive is usually used for data shared among program units executed by different threads. Use the `.tcdata` directive to define external symbolic names of thread storage areas. This directive reserves storage in the `.tdata` program section, for example, for FORTRAN COMMON areas. The loader resolves the common storage directives by allocating storage in the `.tdata` segment. The size of the allocated area is the maximum size specified in any module that contains a definition for the given symbolic name. If more than one `.tcdata` directive appears in a file, the largest size defined is the size for that common block.

The format of the `.tcdata` directive is as follows:

`.tcdata` *name, expression*

where *expression* defines the size of the common area in bytes and must be an absolute constant. *name* refers to the symbolic name defined as the name of the common storage block. Symbolic names used in `.tcdata` directives can be redefined in other `.tcdata` directives. The size of the generated common block is the maximum size declared in any module.

## Symbol-table directives

The symbol table contains information about all symbolic names in a program. The assembler maintains the symbol table and writes it to the generated object file. The loader then uses the table to resolve external references and to add symbolic name relocation to instructions and data that refer to relocatable symbolic names.

Three symbol-table directives transfer source language information from compilers to the source-code debugger, `csd`. These directives are as follows:

`.stabs` *string_constant, abs_expr, abs_expr, abs_expr, reloc_expr*
`.stabn` *abs_expr, abs_expr, abs_expr, reloc_expr*
`.stabd` *abs_expr, abs_expr, abs_expr*

In each of the 3 directives, the first absolute expression (*abs_expr*) specifies the type of symbol-table entry to be made. Use the absolute expressions, *abs_expr*, to fill in specific descriptive fields within the symbol-table entry. Only the `.stabs` directive enters a string (string constant) into the output object file. The `.stabs` and `.stabn` directives enter a relocatable value (*reloc_expr*) into the symbol-table entry. The `.stabd` directive implicitly enters the relocatable value into the symbolic name equal to the current location counter of the `.stabd` directive.

## Procedure summary directives

Procedure summary (psum) directives are used to annotate assembly-language code that is analyzed by the CONVEX Application Compiler. Psum directives occur only in psum files.

The format of the `.psum` directive is:

`.psum` *directive*`,` `[`*arg*`] [,...]`

The values of *directive* include:

`arguments,` *arg1* [,...]
: This directive defines the arguments of the procedure.

`asgs,` *symbol* [,...]
: This directive tells the Application Compiler that the procedure assigns a value to *symbol* (either a global or an argument), depending on what happens at runtime.

`entry_name,` *symbol*
: This directive states that the name of the procedure being annotated is *symbol*. It is required. All other psum directives are associated with the most recent `entry_name` directive.

`kills,` *symbol* [,...]
: This directive makes a stronger statement than the `asgs` directive. The `kills` directive tells the Application Compiler that the procedure *always* assigns a value to *symbol*, a global variable or argument, *regardless* of what happens at runtime.

`no_arg_mods`
: This directive tells the Application Compiler that the procedure where it appears does not assign a value to any of its arguments.

`no_global_mods`

This directive tells the Application Compiler that the procedure does not assign a value to any global variable.

`range_flags`, *procedure*, `heap`

This directive is used to track memory allocation routines. The symbol *procedure* refers to a procedure that returns a pointer to memory that is allocated on the heap.

`range_names`, *symbol* [,...]

Use this directive on a procedure that returns a pointer. The argument *symbol* is a variable or list of variable names that tell the Application Compiler what data objects the return pointer can point to.

`reentrant`

This directive tells the Application Compiler that the procedure is reentrant.

`uses`, *symbol* [,...]

This directive is similar to the `asgs` directive in that it states what *may* happen at runtime. In this case, the directive tells the Application Compiler that the procedure may reference the value of *symbol*, a global variable or argument.

`varargs`, $*num*

Use the `varargs` directive with a procedure that has a variable number of arguments or parameters. The symbol *num* tells the Application Compiler how many arguments or parameters to consider significant for error checking. If the Application Compiler finds that a call does not pass enough arguments or parameters to a procedure, it issues a message telling you that it has found a fatal error. Unless you tell it otherwise, the Application Compiler assumes that all arguments or parameters listed in a procedure declaration are required. Note the use of the dollar sign ($); this is required.

For more information on psum directives, refer to the *CONVEX Application Compiler User's Guide*, Chapter 5, "Creating libraries and object files."

# Operands

# 11

This chapter describes the component parts of an assembly-language operand. Specifically, this chapter discusses the following topics:

- Assembler character set
- Operators
- Terms, including
  - Constants, including numeric constants, character constants, and symbolic names
  - Expressions, including type propagation in expressions
  - Symbolic-name assignment statements
- Location counter
- Addressing modes

## Assembler character set

The assembler uses a subset of the ASCII character set to represent named entities, numbers, and the operations performed on these language components. The assembler uses the characters shown in the following list:

- The letters A through Z and a through z
- The numerals 0 through 9
- Left and right parentheses ( )
- The operators + - * / & and |
- The operator @ to denote operand indirection
- Commas to separate operands
- Semicolons to identify comment fields
- Colons in label definitions

- The characters . _ and $ in names
- The tilde character (~) denotes the unary complement operator.
- The character = in direct assignment statements
- Space and tab characters are treated as white space.
- The line feed <lf>, form feed <ff>, and ! characters are statement terminators.
- The backslash character \ for escape sequences
- The character # for immediate operands

## Operators

Operators are characters that join constants and symbolic names to create expressions. Table 22 lists binary and unary operators.

**Table 22**
Binary and unary operators

| Binary | Unary |
|---|---|
| + Addition | - Unary minus |
| - Subtraction | - Logical negation |
| * Multiplication | |
| / Division | |
| & Logical AND | |
| \| Logical OR | |

The assembler evaluates expressions from left to right with no operator precedence. Thus, the expression `1+2*3` evaluates to 9, not 7. Unary operators have precedence over binary operators because they are considered part of a term; both terms of a binary operator must be resolved before you can apply the binary operator.

## Terms

Terms are constants and symbolic names that make up an expression. Following are the 4 types of terms:

- Numeric constants
- Character constants
- Symbolic names
- Numeric constants, character constants, and symbolic names preceded by a unary operator. For example, both `sum` and `~sum` are terms. Multiple unary operators are also allowed; for example, `--A` has the same value as `A`.

### Numeric constants

The assembler supports decimal, octal, and hexadecimal integer constants, and floating-point constants in C and FORTRAN notation. The default radix for integers is decimal. A decimal number is a sequence of digits that begins with a digit other than zero. A string of digits that begins with a zero is interpreted as an octal number. A hexadecimal constant begins with the sequence "0x" or "0X", and is followed by a sequence of decimal digits or the uppercase or lowercase letters A through F.

Floating-point constants can be represented either as a sequence of decimal digits followed by a period (followed by additional decimal digits) or in the usual scientific notation. For example:

```
1.25e2 = 125.0 = 0.125E3 = 125. = 125E0
```

A floating-point constant represents either a single-precision (32-bit) or a double-precision (64-bit) value, depending on the context in which it appears.

Numeric constants follow the same rules as those in the C compiler. An integer constant can have a trailing "LL" (`long long`) to indicate a 64-bit constant. A decimal integer constant that is larger than the largest signed 32-bit integer is treated as a 64-bit constant. An octal or hexadecimal constant that is larger than the largest unsigned 32-bit integer is treated as a 64-bit constant. Note that 32-bit integer constants can have an optional trailing "L" (for `long`) for compatibility with constants accepted by the C compiler. Specifying all 64 bits of `long long` constants eliminates sign-extension problems when converting a 32-bit constant to a 64-bit constant.

## Character constants

In addition to numeric constants, the assembler also supports character constants. A character constant is denoted by enclosing the desired character within apostrophes ('). For example:

'a' represents the character constant for lowercase a and has a hexadecimal value of 0x61.

'A' represents the character constant for uppercase A and has a hexadecimal value of 0x41.

The assembler allows you to specify nonprinting characters using escape sequences in character constants. Escape sequences are described in Table 23.

Table 23
Assembler escape sequences

| Description | Assembler notation |
|---|---|
| Backslash | `\\` |
| Backspace | `\b` |
| Bit pattern | `\ddd` |
| Carriage return | `\r` |
| Double quote | `\"` |
| Form feed | `\f` |
| Horizontal tab | `\t` |
| Newline | `\n` |
| Single quote | `\'` |

The escape sequence `\ddd` consists of the backslash and 1 to 3 octal digits specifying the value of the desired character. For example, `\0` indicates the character NUL. When the character following a backslash is not one of those specified in Table 23, the backslash is ignored.

## Symbolic names

You can use decimal digits, letters, and the special characters dollar sign ($), dot (.), and underscore (_) in symbolic names. However, a digit cannot be used as the first character in the name. All characters in the name are significant. Uppercase and

lowercase letters are distinct; for example, the assembler considers the symbolic names "ONE" and "one" to be separate symbolic names.

A symbolic name is declared when the assembler first recognizes it as a symbolic name in the program. A symbolic name is defined when a value is associated with it. A symbolic name used as a label can not be redefined; all other symbolic names can receive a new value.

A symbolic name can be declared:

- As the label of a statement
- In a symbolic-name assignment statement
- As an external symbolic name using the `.globl` directive
- As a common symbolic name using the `.comm` directive
- As a local symbolic name

Instruction mnemonics, assembler directive names, and register names are reserved symbolic names. You cannot redefine a reserved symbolic name.

## Expressions

Expressions are a combination of constants and symbolic names joined by operators. The assembler resolves expressions to a signed 4-bit value, then converts the result of the expression to a size appropriate for the destination. An expression can be relocatable, absolute, or external.

### Relocatable expressions

Relocatable expressions reference addresses that cannot be resolved at assembly time. Relocatable expressions contain two parts: a constant value and an offset determined by the loader. Relocatable expressions are necessary because the loader determines the addresses of certain symbolic names after assembly is complete. If the assembler encounters one of these symbolic names, the expression is relocatable, and the symbolic name value is measured by the origin of the program segment in which it is found.

Symbolic names that cannot be resolved at assembly time include text, data, tdata, bss, and tbss symbolic names. Text, data, tdata, bss, and tbss symbolic names are located in the `.text`, `.data`, `.tdata`, `.bss` and `.tbss` segments of the program, respectively.

Because most of the symbols encountered in a typical module are relocatable, the majority of expressions found in a given module are relocatable. Valid relocatable expressions include the following:

- Expressions that consist of a relocatable term
- Expressions that consist of a relocatable term plus or minus a constant
- Expressions that consist of a relocatable term plus or minus an absolute term

The following examples illustrate both valid and invalid uses of relocatable expressions:

```
sum         ; relocatable
sum+5       ; relocatable
sum*2       ; not relocatable (error)
2-sum       ; not relocatable (error)
```

## Absolute expressions

Absolute expressions contain only terms that do not need to be relocated and can be resolved at assembly time. Because addresses are not the principal terms of these expressions, subsequent manipulation of these terms by the loader is counterproductive. An absolute symbolic name is one whose value is an absolute expression. To prevent loader manipulation, define values for absolute symbolic names using assignment statements.

## External expressions

External expressions contain at least 1 external symbolic name. An external symbolic name is visible from other modules, and its value is accessible to the loader. External symbolic names are not always defined in another module, but are visible globally and can be used to resolve external references in other modules.

There are two types of external symbolic names. The first type is called a *defined external,* and is declared externally using the `.globl` assembler directive. These names are also defined in the current assembly. The loader has access to the value and type of these symbolic names in order to resolve references from other object modules.

The second type of external symbolic name is called an *undefined external*. These names are not defined in the current assembly. Any symbolic name used in the current source file, but not defined, is assumed to be defined in another module. Similarly, symbolic names specified in a `.globl` directive, but not declared, are assumed to be defined in another module. If you use such a symbolic name, the loader resolves the undefined external references to definitions in other modules.

You define a symbolic name when you give it a value either explicitly (using an assignment statement) or implicitly (using the symbolic name as a statement label).

## Type propagation in expressions

When you combine operands by expression operators, the resulting type depends on the types of the operands and the operator. The combination rules include the following:

- If both operands are absolute, the result is absolute.
- **Addition operations**—If one operand is either relocatable or an undefined external, the other operand must be absolute, and the result is relocatable.
- **Subtraction operations**—If the first operand is relocatable, the second operand can be absolute or relocatable. If the second operand is absolute, the result is relocatable. If the second operand is relocatable within the same segment as the first, the result is absolute. If the first operand is external undefined, the second must be absolute. All other combinations are invalid.

## Symbolic name assignment statements

A symbolic name assignment statement assigns the value of an expression to a symbolic name. The format of a symbolic name assignment statement is as follows:

*symbolic_name* = *expression*

Following are examples of valid assignments:

```
count = 0x100
init = 0
```

Each statement assigns one value to one symbolic name. You can redefine symbolic names defined by assignment to assume the value of the most recent assignment. You can not redefine labels or register symbolic names.

Absolute, relocatable, and external symbolic names adhere to the following rules:

- If the defining expression is absolute, the symbolic name is also absolute. Absolute symbolic names can be treated as constants in subsequent expressions.
- If the expression is relocatable, the symbolic name is also relocatable. Relocatable symbolic names are declared in the same program section as the relocatable symbol in the relocatable expression.
- If the expression contains an external symbolic name, the symbolic name defined by the assignment statement is also considered external. You can define external symbolic names by direct assignment.

### Location counter

The " . " character (called `dot`) is a symbolic name that contains the value of the current assembler location counter. The value of the location counter during assembly is the number of bytes generated in the current subsegment (0, 1, 2 or 3) of the current program segment (text, data, tdata, bss, or tbss). The value of the location counter is always relocatable.

The dot symbolic name follows the standard rules of use or assignment. Use reassignment to change the offset of the next instruction or datum. For example

```
xyz = .
```

assigns the symbolic name xyz the current value of the assembler location counter, whereas

```
. =. + 10
```

adds 10 to the assembler location counter.

This facility can be used to deposit instructions or data at fixed offsets within a routine, such as in array initialization, where the addresses and values of the initialization data may appear in random order (as for DATA statements in FORTRAN). For example, the following two sequences of directives in Table 24 produce the same effect; each creates a table holding the values 1, 2, and 3.

**Table 24**
Location-counter-directive sequences

| Sequence 1 | Sequence 2 |
|---|---|
| .data | .data |
| table: | table: |
| ds.w 1 | .=.+8 |
| | ds.w 3 |
| ds.w 2 | .=.-8 |
| | ds.w 2 |
| ds.2 3 | .=.-8 |
| | ds.w 1 |

In Table 24, the current value of dot (.) is the byte offset address where the next byte generated is deposited. This value changes as soon as any bytes are generated. Thus, under Sequence 2, the lines `. = . +8` and `. = . -8` do not refer to the same offset within the table.

Using this mechanism, there is no way to specify an absolute address in memory when the program executes. You can use this mechanism in conjunction with the `.align` directive to force alignment of data or instructions. Any value assigned to dot (.) must not contain any relocatable symbolic name references. Any holes created in the text or data segment by skipping locations are set to zero if no other values are deposited at the locations.

## Addressing modes

The CONVEX assembly-language instruction set supports 8 operand-addressing modes: register mode, immediate mode, and 6 modes for specifying operands in memory. Because the CONVEX architecture is based on 3 sets of high-speed registers, most of the instructions use the register mode, which is used for register-to-register operations.

### Register mode

The majority of instructions in the instruction set require one or more register-mode operands that specify which register is used as the operand. The actual machine instruction generated by the assembler depends on both the register set used and the register within that register set.

Table 25 lists the register names for the CONVEX supercomputer register set.

**Table 25**
Register names

| Registers | Function |
|---|---|
| **Address registers** | |
| a0, A0, sp, SP | Stack pointer |
| a1, A1 | |
| a2, A2 | |
| a3, A3 | |
| a4, A4 | |
| a5, A5 | |
| a6, A6, ap, AP | Argument pointer |
| a7, A7, fp, FP | Frame pointer |
| **Scalar registers** | |
| s0, S0 | Scalar registers |
| s1, S1 | |
| s2, S2 | |
| s3, S3 | |
| s4, S4 | |
| s5, S5 | |
| s6, S6 | |
| s7, S7 | |
| **Vector registers** | |
| v0, V0 | Vector registers |
| v1, V1 | |
| v2, V2 | |
| v3, V3 | |
| v4, V4 | |
| v5, V5 | |
| v6, V6 | |
| v7, V7 | |
| vl, VL | Vector length register |
| vls, VLS | Vector length and stride combination |

**Table 25 (continued)**
Register names

| Registers | Function |
|---|---|
| `vm, VM` | Vector merge register |
| `vmu, VMU` | Vector merge register, upper longword |
| `vml, VML` | Vector merge register, lower longword |
| `vs, VS` | Vector stride register |
| `vv, VV` | Vector valid register |
| **Other registers** | |
| `cir, CIR` | Communication index register |
| `cpuid, CPUID` | CPU identification |
| `icr, ICR` | Interrupt control register |
| `psw, PSW` | Processor status word |
| `pc, PC` | Program counter |
| `tcpu, TCPU` | Target CPU register |
| `tid, TID` | Thread identification |
| `toc, TOC` | Time-of-century clock |
| `ttr, TTR` | Thread timer |

General-register operands are specified by a letter and a number. The letter denotes the register set, whereas the number specifies the register number in the set. The general-register sets for CONVEX supercomputers are the address registers (A), the scalar registers (S), and the vector registers (V). You can specify the register-set letter in either uppercase or lowercase. Each set contains eight registers denoted by the numbers 0 through 7. The following examples illustrate the correct use of register-mode operands.

```
a2   ; Address register 2
A2   ; Address register 2
S4   ; Scalar register 4
V0   ; Vector register 0
v2   ; Vector register 2
```

To improve program readability, you can reference the following 3 address registers with special functions by special names.

```
SP or sp          ; Stack pointer (A0)
AP or ap          ; Argument pointer (A6)
FP or fp          ; Frame pointer (A7)
```

The assembler uses reserved words to denote the machine registers. The assembler references each register by the reserved words only; hence, you may not use other symbolic names to denote machine registers.

Special-purpose registers in the machine are used for machine control. Use special reserved words to specify these registers. Table 26 lists these special registers. Consult the *CONVEX C Series Architecture Reference Manual* for more information on the function of these registers.

Table 26
Machine-control registers

| Special registers | Descriptions |
|---|---|
| `CIR` or `cir` | Communication index register |
| `CPUID` or `cpuid` | CPU identification |
| `ICR` or `icr` | Interrupt control register |
| `PSW` or `psw` | Processor status word |
| `PC` or `pc` | Program counter |
| `TCPU` or `tcpu` | Target CPU register |
| `TID` or `tid` | Thread identification |
| `TOC` or `toc` | Time of century |
| `TTR` or `ttr` | Thread timer |
| `VL` or `vl` | Vector length register |
| `VLS` or `vls` | Vector length and stride combination |
| `VM` or `vm` | Vector merge register |
| `VMU` or `vmu` | Vector merge register, upper longword |
| `VML` or `vml` | Vector merge register, lower longword |
| `VS` or `vs` | Vector stride register |
| `VV` or `vv` | Vector valid register |

## Immediate addressing mode

Immediate operands provide a method for referencing data from the program's instruction stream. The assembler syntax used to specify an immediate operand is as follows:

#*expression*

where *expression* defines the value of the immediate operand in question. The value of the expression can be either relocatable or absolute. The loader resolves valid relocatable expressions by adding a relocation offset to the value of the absolute portion of the expression. Table 27 shows both valid and invalid immediate operands.

**Table 27**
Immediate operands

| Valid immediate operands | |
|---|---|
| `#3` | The absolute constant 3. |
| `#label-5` | The relocatable expression with the value of the label plus the absolute constant -5. |
| `#lab1-lab2` | The absolute value that is the difference between the relocatable symbolic names. This value does not result in a relocatable expression because the relocation constants cancel each other in the subtraction. |
| **Invalid immediate operands** | |
| `#lab1+lab2` | Sum of two relocatable expressions. |
| `#lab1+2` | This operand is not a relocatable expression because the relocation offset cannot be added to an absolute quantity. |
| `#5-label` | This operand is not a relocatable expression because the relocation offset cannot be added to an absolute quantity. |

Floating-point values can appear as immediate operands in `.s` and `.d` versions of the instructions that support immediate operands. For example, following is a legal instruction that loads the value 3.14159 into register `s0`:

```
ld.s  #3.14159,s0
```

The type of datum placed in the immediate field should agree with the suffix of the instruction operator. For example,

```
ld.w #3.5,s0
```

generates an error message because `ld.w` is an integer operation.

The proper encoding of this instruction is

```
ld.s #3.5,s0
```

Values of 32 bits or less are assembled directly into the immediate field of the instruction. For values greater than 32 bits, the assembler encodes either the upper or lower 32 bits into the immediate field of the instruction, depending on the width specifier of the op code.

The pseudo-instructions `ld.du`, `ld.dl`, `ld.lu`, and `ld.ll` simplify the loading of 64-bit immediate operands. For example, the code sequence

```
ld.lu   #0x123456789abcdef, s0
ld.ll   #0x123456789abcdef, s0
```

loads the upper and lower halves of the constant into register `s0`. The `ld.lu` and `ld.ll` instructions discard the lower and upper halves, respectively, of their immediate operand. Similarly, the instructions

```
ld.du   #1.1,s1
ld.dl   #1.1,s1
```

load the double-precision floating-point value 1.1 into the `s1` register. These pseudo-instructions generate the CONVEX op codes (`ld.d`, `ld.l`, and `ld.w`) needed to load the immediate value. The assembler does not check to ensure that these instructions occur in pairs. The assembler converts floating-point constants into either 32- or 64-bit representations, depending on the context.

## Memory-addressing modes

This section describes the addressing modes used to specify operands in memory. These modes are typically used to load registers from memory or to transfer the contents of registers to memory addresses.

### Absolute-addressing mode

An absolute address in the virtual address space is referenced when a program specifies its location. You can specify this location with an expression that evaluates to the desired address. To achieve this evaluation, include an absolute number and a relocatable portion in the expression. The loader resolves the relocatable portion by adding a relocation constant to the relocatable portion, and then adding in the absolute number.

Following are examples of instructions that use absolute addressing modes:

```
tas    1000         ;Test and Set the byte
                    ;at location 1000 in the
                    ;current segment.
ld.w   data+6,a4    ;a4 <= c(data+6)
                    ;load the value located
                    ;at the sixth through ninth byte
                    ;after the relocatable symbolic
                    ;name "data" into register a4.
```

### Register-deferred mode

When you use register-deferred mode, the address registers can contain the address of an operand to be used by the instruction. When you use this mode, the assembler expects the specified register to contain a pointer to the operand rather than the operand itself. To specify this addressing mode, enclose in parentheses the address register that contains the pointer. The following examples illustrate the use of this mode.

```
ld.w (a3),a4        ;a4<=c(a3)
                    ;load the contents of the 4 bytes
                    ;pointed to by address register
                    ;a3 into address register a4.

ld.b (a3),s4        ;s4<=s4c(a3)
                    ;load the byte pointed to by
                    ;address register a3 into
                    ;scalar register s4.
```

### Indexed mode

The indexed mode adds an offset, or base, to the contents of the specified address register, and the result is used as a pointer to the operand. To form this addressing mode, precede a deferred-register specifier with an expression, as in the following examples:

```
ld.w    100(a3),a4      ;a4<=c(a3+100)
                        ;The contents of address
                        ;register a3 and the value
                        ;100 are added to form a
                        ;pointer to the 4-byte source
                        ;operand.

st.w    s5,blue(a4)     ;c(blue+a4)<=s5
                        ;The contents of scalar
                        ;register s5 are stored in
                        ;the memory location that is
                        ;pointed to by the sum of
                        ;the value of absolute
                        ;symbolic name "blue" and
                        ;the contents of address
                        ;register a4.
```

### Indirect-absolute mode

This mode enables you to use a single level of indirection when specifying an absolute address. An expression specifies the absolute address that contains the operand pointer. To specify this mode, include the @ character before an expression. The following examples illustrate the use of this addressing mode.

```
ld.w   @1000,a4        ;a4<=c(c(1000))
                       ;The 4-byte memory location
                       ;1000 contains the address of
                       ;a 4-byte operand that is to
                       ;be loaded into address
                       ;register a4.
st.b   s4,@dpointer    ;c(c(dpointer))<=s4
                       ;Store the byte contained in
                       ;scalar register s4 into the
                       ;memory location pointed to
                       ;by the 4 bytes at memory
                       ;location specified by
                       ;"dpointer".
```

### Indirect-deferred mode

This mode adds another level of indirection to the register-deferred mode. To specify this mode, prefix the @ character to a deferred-register operand enclosed within parentheses. The deferred-register operand contains the address of a pointer to the desired operand. The following examples illustrate the use of this addressing mode.

```
ld.w   @(a4),a5     ;a5<=c(c(a4))
                    ;Address register a4 contains the
                    ;address of 4 bytes that specify
                    ;the address of the desired word
                    ;operand.

st.b   s5,@(a5)     ;c(c(a5))<=s5
                    ;Register a5 contains the
                    ;address of 4 bytes that specify
                    ;the address of the location
                    ;where the byte contained in s5
                    ;is stored.
```

### Indirect-indexed mode

To denote indirect-indexed mode, precede an indexed operand with the @ character. When you specify this mode, the assembler adds the value of the register and the value of the absolute index expression to form the address of a pointer to the desired operand. Use of this mode is illustrated by the following examples.

```
ld.w  @sum(a3),a4   ;a4=c(c(sum+a3))
                    ;The value of the absolute
                    ;symbolic name "sum" and the
                    ;contents of address register
                    ;a3 are added together to form
                    ;the address of a pointer to
                    ;the 4-byte operand to be
                    ;loaded into address
                    ;register a4.

st.b  s5,@8(a4)     ;c(c(8+a4))=s5
                    ;The byte contained in register
                    ;s5 is stored in the memory
                    ;location pointed to by the
                    ;pointer that is stored at the
                    ;address that the sum of
                    ;register a4 and 8.
```

# Calling conventions 12

This chapter describes conventions you should be familiar with when using the assembler. Specifically, this chapter discusses the following topics:

- Special address registers, including the stack pointer, argument pointer, and frame pointer
- Linkages
- General calling conventions
- Runtime-stack layout
- C calling conventions
- FORTRAN calling conventions

## Special address registers

Some address registers are used for special functions by the hardware and the compilers to implement subprogram calls. The address registers A0, A6, and A7 are used as the stack-pointer (SP), argument-pointer (AP), and frame-pointer (FP) registers, respectively.

### Stack pointer

The `psh` and `pop` machine instructions manipulate data on the top of a runtime stack by incrementing and decrementing the SP. Although the other address registers can be used as index registers for load and store instructions, the SP cannot. The hardware uses a 0 in the index register field of a load or store instruction to indicate that no index register is to be used. This encoding prevents the use of the stack pointer (A0) as an index register. Compilers generate automatic (local) storage on top of the runtime stack by directly adjusting the SP.

## Argument pointer

By convention, the code generated by CONVEX compilers uses address register A6 as an argument-pointer (AP) register. Before calling a subprogram (subroutine or function), the generated code loads into the AP the base address of the argument list to be passed to the called routine. This argument list can be located either on the runtime stack or anywhere in the program address space. When possible, compilers build packets of subprogram arguments at compile time to increase runtime efficiency. This called routine references the arguments passed to it by using the AP as an index register.

## Frame pointer

The call instructions (`call` and `calls`) and the return instructions (`rtn` and `rtnc`) use address register A7 as a frame pointer in order to maintain linkage information for subprogram calls. Each `call` or `calls` instruction causes the contents of several registers to be saved on the top of the runtime stack (determined by the contents of SP), and the value of FP is updated to point to the saved context. The compiler-generated code uses the FP as an index register to refer to automatic storage. The program can find the previous routine context blocks by retrieving subsequent frame pointers out of the saved context block pointed to by the current frame pointer.

A function uses scalar register S0 to return its result value to the calling routine. The return (`rtn`) instruction does not alter the value of S0, so the return value is available to the calling routine.

## Compiler-generated code

Other than the SP, AP, and FP registers, the compiler-generated code makes no other assumptions about registers being preserved across procedure calls. The CONVEX C compiler (`cc`) ignores register variables. The CONVEX FORTRAN compiler (`fc`) explicitly saves whatever intermediate information exists in registers before calling a subprogram.

## Linkages

The interprocedural call instructions `call`, `calls`, and `callq` store information on and retrieve information from the runtime stack. Each of these 3 instructions push current state information on the runtime stack and branch to a destination address contained within the instruction. For the return, the instructions keep track of the called routine's virtual address.

### callq

The `callq` instruction, or fast call, pushes the current value of the program-counter register on the runtime stack. The value in the stack-pointer register is decreased by 4 to account for the pushed value. None of the other registers is saved by the `callq` instruction.

To return, the routine executes a `rtnq` instruction, which removes the program-counter (PC) value (the top word of the runtime stack), and places the return address into the PC where execution continues after the subroutine call. The CONVEX C and FORTRAN compilers never use the `callq` instruction for calling user-written functions; `callq` is used only for calling short library routines that perform housekeeping functions.

### calls

The `calls` instruction is the most frequently used instruction for interprocedural calling. Like the `callq` instruction, the `calls` instruction pushes the value of the return address onto the runtime stack. It also pushes the value of the processor status word, the value of the frame-pointer register (A7), and the value of the argument-pointer register (A6).

After these 4 words have been pushed on the stack, the frame-pointer register (A7) is updated to contain the current value of the stack pointer. The called routine can then access the last stack frame on the runtime stack using the frame-pointer register. To return control to its caller, the called routine executes a `rtn` instruction.

### `call`

Like the `calls` instruction, the `call` instruction saves the current values of the program counter, processor status word, frame pointer, and argument pointer on the runtime stack. The `call` instruction also saves the current values of the other address and scalar registers, except A0 and S0.

The CONVEX C and FORTRAN compilers currently do not generate any `call` instructions, but use the faster `calls` instruction instead.

## General calling conventions

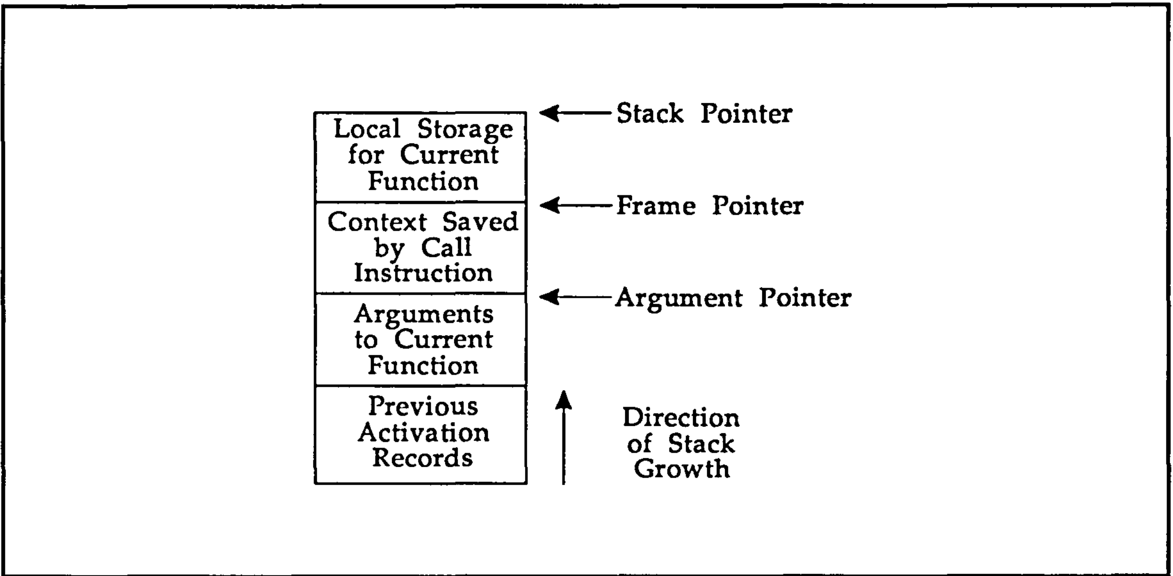
This section describes the general layout of the activation records used with CONVEX C functions and FORTRAN functions and subroutines. When function or subroutine calls are executed, the current state of several hardware registers used by the calling routine must be preserved. The contents of these registers are pushed onto the runtime stack as a part of an activation record. The called function may then alter the machine registers as it runs. The hardware, as part of the return to the original calling routine, restores the old values of the saved registers.

CONVEX compilers do not preserve the value of every register across a function call. Only those registers required to maintain the state of the runtime stack are preserved. Called routines that can restore the frame-pointer register to its original state are allowed to modify any register passed to them.

### Function or subroutine stack layout

Figure 25 illustrates the top of the runtime stack. Although the stack grows downward in the address space, this diagram shows the stack as growing upward on the page. The stack-pointer register contains the address of the topmost location on the runtime stack. The frame-pointer register contains the address of the last frame pushed on the runtime stack by a `call` or `calls` instruction. The argument-pointer register contains the address of the arguments passed to the current routine.

Figure 25
Top of the runtime stack




## Function or subroutine calling sequences

When a function is called, the compiler-generated code follows one of the following two sequences:

### Sequence 1

1. Push the values of the arguments to the function onto the runtime stack in reverse order.

2. Update the argument-pointer register. The updated register should point to the first argument in the argument list. (The first argument in the list is the last one pushed.)

3. Push an additional word. This word should contain the number of arguments passed.

4. Call the routine with a `calls` instruction.

Executing the `calls` instruction places a stack frame on the runtime stack. The stack frame contains the current value of the program counter (return address), the current value of the processor status word, the old value of the frame pointer, and the current value of the argument pointer.

## Sequence 2

For FORTRAN, the arguments are precompiled when possible, so the calling sequence is reduced to the following:

1. Load the address of the argument packet into the argument pointer.

2. Call the subroutine (using the `calls` instruction) Conventions that apply to function calls are listed below:

- The called routine can allocate storage for local variables on top of the runtime stack. No stack references in CONVEX C code are made relative to the top of the runtime stack. Storage allocated on the stack by called routines is automatically deallocated when the function returns.

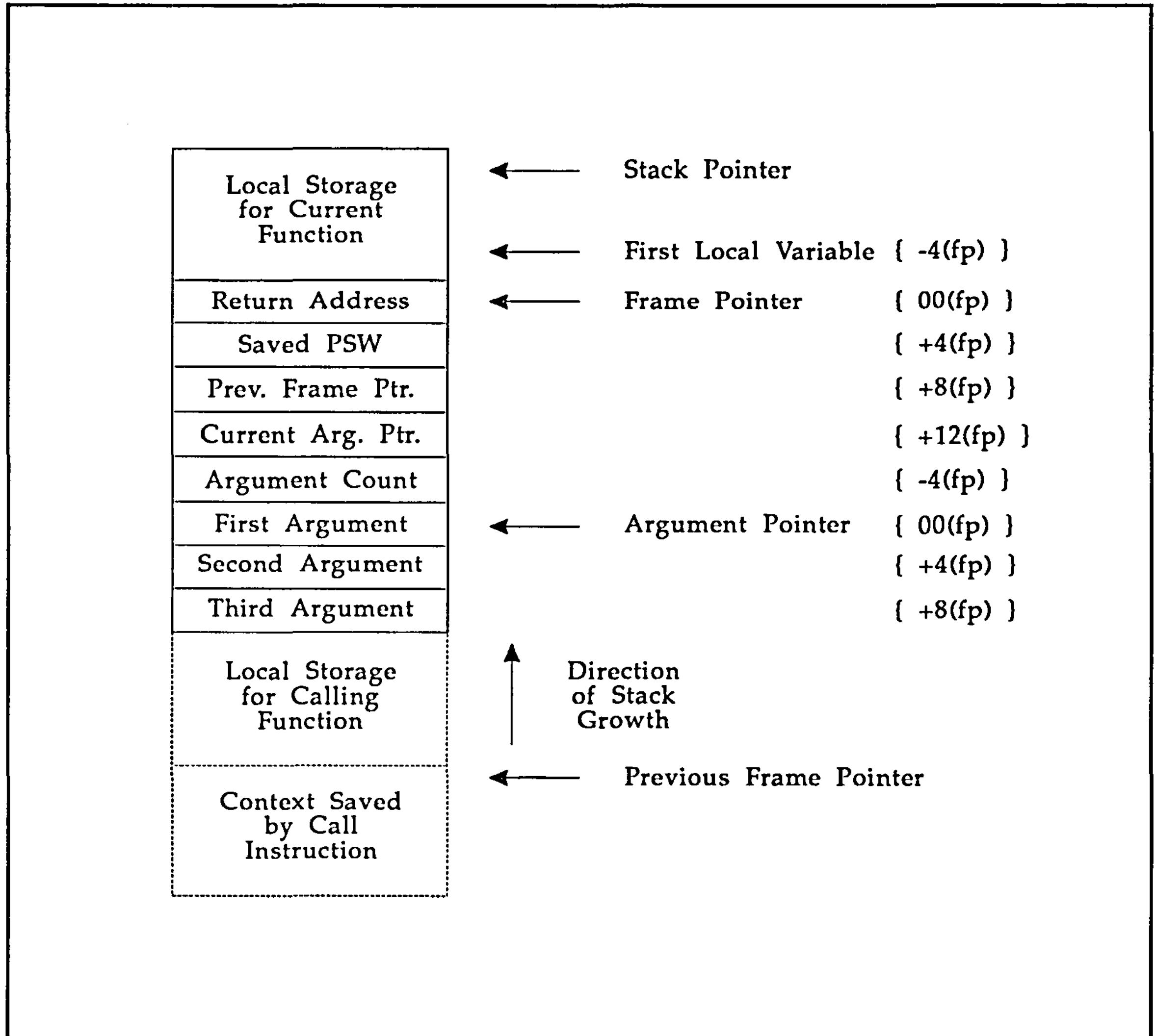
- The called routine need not preserve the contents of any register except the frame pointer. The called routine uses the current value of the argument-pointer register to access the arguments passed to the routine by its parent.

- The frame-pointer register points to the context block pushed by the caller when calling the child routine. The called child routine references the local storage it has allocated on the runtime stack by referencing negative offsets from the frame pointer.

- The called routine references arguments passed to it by its parent by referencing positive offsets from the argument-pointer register: 0(ap) corresponds to the first argument, 4(ap) corresponds to the second argument, and so on. The word with an address of -4 relative to the argument pointer contains a count of the number of arguments passed to the routine.

Figure 26 illustrates the layout of the stack as seen by a routine after it has been called, and after it has allocated some storage for local variables on the top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 26
Stack layout




Called routines return to their parents by:

1. Placing a return value in register S0.

2. Executing the `rtn` instruction.

When the computer executes the `rtn` instruction, automatic storage allocated by the called routine is automatically deallocated. This instruction also restores the PSW register and the frame pointer to their previous states, and then returns control to the location immediately following the `calls`

instruction that called the routine. After control returns to the parent, the stack-pointer register points to the location that contains the pushed argument count. The parent routine adds a fixed offset to the value in the stack pointer to remove the argument count and any pushed arguments. The value added is the total number of bytes pushed prior to the call. Finally, before the parent can access any of its own arguments, it must reload its own argument-pointer register from the current frame on the stack. This value is offset +12 bytes from the recently restored frame pointer.

## C calling conventions

This section includes basic information on C calling conventions. For more information, refer to the *CONVEX C Guide*.

### Code generated for function calls

The following is a section of sample CONVEX assembly-language code for a function call. All C functions are called in this manner. Only a few code-support routines are called using other mechanisms, which the compiler generates on a case-by-case basis. Figure 27 illustrates the call to a C function, `child (a, b, c)`.

Figure 27
Calling a C function

```
_parent:

         psh.w c3              ; value of rightmost argument
         psh.w b2              ; value of second argument
         psh.w a1              ; value of first argument
         mov sp,ap             ; arg pointer points at first arg
         pshea 3               ; push count of 3 of args passed
         calls _child          ; use "calls" to call function
         add.w 16,sp           ; remove bytes pushed for args
         ld.w 12(fp),ap        ; reload argument pointer

                               ; the code shown below is used in
                               ; the called child function:

         .globl _child         ; called function
_child:
         sub.w 20,sp           ; allocate bytes for local variables
         ld.w 0(ap),s0         ; load the value of the first arg
                               ; assuming 32-bit size
         ld.w -4(ap),s1        ; load count of the args passed
         st.w s0,-4(fp)        ; first local int is at -4(fp)
         sub.w s0,s0           ; function returns 0 in s0
         rtn                   ; return to caller
```

## Function names

Function names and global variables produced by the compiler in object code are unrestricted in length. An underscore character is prefixed to each global variable.

## Function arguments and return values

C arguments are passed using the "call-by-value" method, in which the values of arguments, rather than their addresses, are passed. C programs can pass addresses as arguments to functions by using pointer variables. Structures in C are also passed by value. All of the elements of the structure are pushed onto the runtime stack prior to the call. The calling process accelerates when structure pointers, rather than the structures themselves, are passed. Results returned from functions are returned in scalar register S0. Functions that return structures as results place the function result in a static area in the data segment and return a pointer to that area in S0.

## FORTRAN calling conventions

This section includes some basic information on FORTRAN calling conventions. For more detailed information, refer to the *CONVEX FORTRAN User's Guide.*

### Code generated for function calls

Figure 28 is an example of FORTRAN, assembler, and C code to call a FORTRAN subroutine.

**Figure 28**
Calling a FORTRAN subroutine

```
call sub (a,b,c)          ;FORTRAN call with three real arguments

                                  ; equivalent assembler calling sequence
 pshea c                          ; push the third argument's address
 pshea b                          ; push the second argument's address
 pshea a                          ; push the first argument's address
 mov sp,ap                        ; set up the ap
 pshea 3                          ; push number of words in argument list
 calls sub                        ; call the subroutine
 ld.w 12(fp)ap                    ; restore ap
 add.w #16,sp                     ; restore sp

;   sub (a,b,c);                /* equivalent C call */
```

## Note

**The arguments are pushed in reverse because the stack grows toward low memory addresses. Thus, the last argument pushed ends up at the top of the list. This way, trailing arguments can be omitted while function arguments are still uniformly referenced.**

### Subprogram names

On ConvexOS systems, the name of a common block or a FORTRAN subprogram has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. FORTRAN library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

## Function arguments and return values

A function of type integer, logical, real, or double precision returns the corresponding type in scalar register S0. A complex or double-complex function is mapped to a FORTRAN subroutine with an additional initial argument pointing to where the return value is to be stored. Table 28 illustrates a complex function and its FORTRAN subroutine equivalent.

**Table 28**
Complex function and FORTRAN subroutine

| Complex FORTRAN function | Subroutine equivalent |
|---|---|
| `complex function f( ... )` | `subroutine f(temp,...)` |

A character-valued function is equivalent to a FORTRAN subroutine with two extra initial arguments: a data address and a length. Table 29 illustrates a character-valued FORTRAN function, its C equivalent, and how to invoke its C equivalent.

**Table 29**
Character-valued function equivalent

| Character-valued FORTRAN function | Subroutine equivalent |
|---|---|
| `character*15 function g( ... )` | `subroutine g(temp, ...)` |

## Subroutine arguments and return values

Subroutines are invoked as if they are integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3), nret( )
```

All FORTRAN arguments are passed by address. Also, for every argument that is of type character, an argument giving the length of the value is passed. (The string lengths are `word` quantities passed by value.) The order of arguments is as follows:

1. Extra arguments for complex and character functions.
2. Address for each datum or function.
3. A `word` for each character or procedure argument.

# Using the assembler

# 13

This chapter describes general procedures for using the assembler. Specifically, this chapter discusses the following topics:

- Invoking the assembler
- Command-line options, including options to:
  - Select floating point format
  - Redirect assembler object output
  - Suppress warning messages
  - Generate a source listing
- Error messages

## Invoking the assembler

To invoke the assembler directly from the shell, use the following command sequence:

```
as [options] filename
```

where *filename* is the name of a previously created assembly-language program that ends in `.s`. You do not have to specify the `.s` extension in the argument because the assembler assumes that the file name ends in a lowercase `.s` (see `abc` in the example later in this section). Specify only one file name argument per `as` command. The next section, "Command line options," describes the values of *options*.

The next example illustrates the use of the as command to produce object files from 3 assembly-language programs named abc.s, abd.s, and abl.asm.s, respectively. User input is in bold type, and the ls command lists the object files created by the assembler and the original source files.

```
%as abc
%as abd.s
%as abl.asm.s

%ls
abc.o   abd.o   abl.asm.o
abc.s   abd.s   abl.asm.s
```

The as command in this example calls the assembler to produce object files from the source files abc.s, abd.s, and abl.asm.s. The assembler writes the object file produced as output to a file with the same name as the source file, but with an extension of .o instead of .s. In this example, the assembler produces object files named abc.o, abd.o, and abl.asm.o.

## Command line options

The command line options that are available with the as command are as follows:

-ada

This option permits special characters to be used in Ada symbols. It also recognizes assembler directives that are used only in Ada.

-cvtdc3

This option is required for assembling source files that will be executed on a CONVEX C3800. This option converts the instruction cvtd.w v0,v0 to the two instructions cvtd.l v0,v0 and cvtl.w v0,v0.

**Caution**

**You must have IEEE hardware for the assembler to create IEEE object files. If you run the assembler and specify IEEE format when your site does not have the appropriate hardware to support it, you receive a message stating that IEEE hardware is unavailable, and the assembler exits.**

`-fi`
`-fn`

The `-fi` option specifies that constants are translated to the IEEE floating-point format and that the mode of the object file is IEEE. Only machines with IEEE hardare support can assemble files with the `-fi` option. The `-fn` option specifies that constants are translated to the CONVEX native floating-point format and the floating-point mode of the object file is native. If neither the `-fi` nor `-fn` option is included on the command line, the site default is determined with the `getsysinfo` system call. You can use the `getsysinfo -v` command to see what the default is on your system.

**-l**

This option generates a source listing on the standard output stream. The listing contains the following fields:

| | |
|---|---|
| First field | The source line number within the input file. |
| Second field | The value of the location counter (".") within the current program segment. |
| Third field | The binary representation of the data or instruction. |
| Fourth field | The current input source statement. |

**-n**

This option instructs the assembler not to generate product code ID information at the start of the text section. This information is generated by default.

`-o` *outfile*

This option specifies that *outfile* is the file name to contain the generated object module. The default object file name is the source file name with an extension of `.o` .

`-tm` *target*

This option specifies the target instruction set for the assembler. *target* can be one of the following:

| Target | Computer |
| --- | --- |
| `c1,C1` | CONVEX C1 Series |
| `c2,C2` | CONVEX C2 Series |
| `c32,C32` | CONVEX C3200 Series |
| `c34,C34` | CONVEX C3400 Series |
| `c38,C38` | CONVEX C3800 Series |

**-w**

This option suppresses warning messages issued by the compiler.

## Error messages

When the assembler detects errors, it sends error messages to the standard error stream stderr. This error stream is displayed on your terminal unless you redirect it. Each error message includes the file name, line number, and a description of the error. A typical error message is

```
"file.s", line 13: Invalid op-code
```

When the assembler detects an error, it ignores the remainder of the statement and begins processing on the next line.

# Sample programs

# 14

This chapter illustrates and explains sample assembly-language code. Specifically, this chapter describes examples of code to do the following:

- Copy a block of storage from one location in memory to another
- Execute a vector routine
- Execute a parallel program

## Code to copy block of memory

Figure 29 contains a sample CONVEX assembly-language program that copies a block of memory from one location to another.

**Figure 29**
Example of bcopy

```
1       ;
2       ; _Bcopy -- block memory copy
3       ;
4       ; This routine copies a block of storage from one location in
5       ; memory to another.
6       ;
7       ; The calling sequence is:
8       ;
9       ;           psh.w byte-count-value
10      ;           psh.w destination-address
11      ;           psh.w source-address
12      ;           mov sp,ap
13      ;           pshea #3
14      ;           calls _Bcopy
15      ;           add.w #16,sp
16      ;
17      ; No registers are preserved
18
19                  .text
20                  .globl _Bcopy
21
22       _Bcopy:
```

Characters following a semicolon on a line are comments and are ignored by the assembler.

**Lines 1-17**

Comments that describe the function of the routine.

**Line 19**

The `.text` directive specifies that any instructions or data that follow are to be treated as program instructions. The assembler places program instructions in write-protected memory when the program is executed.

**Line 20**

The `.globl` directive specifies that the symbolic name `_Bcopy` is visible to other routines in the program. Use `.globl` to create symbolic names that are visible to all the modules in a program. By convention, the C compiler creates global symbolic names with a leading underscore character.

**Line 22**

The `_Bcopy:` label supplies the location used as the relocatable value for the global symbolic name specified in the `.globl` directive.

**Figure 29 (continued)**
Example of bcopy

```
23          ld.w 4(ap),a1          ; a1 <== dest
24          ld.w 0(ap),a2          ; a2 <== source
25          ld.w 8(ap),a3          ; a3 <== count
26
27   lcopy: leu.w #8,a3            ; copy any longwords?
28          jbra.f wtest           ; no--try words
29          ld.l (a2),s0           ; yes--load a longword
30          st.l s0,(a1)           ; store a longword
31          add.w #8,a1            ; bump destination pointer
32          add.w #8,a2            ; bump source pointer
33          sub.w #8,a3            ; decrement count
34          jbr lcopy              ; try again
35
36   wtest:
37          leu.w #4,a3            ; can copy one word?
38          jbra.f htest           ; no--try halfword
39          ld.w (a2),s0           ; yes--load a word
40          st.w s0,(a1)           ; store a word
41          add.w #4,a1            ; bump destination pointer
42          add.w #4,a2            ; bump source pointer
43          sub.w #4,a3            ; decrement count
44
```

**Lines 23–25**
Load the values of the 3 arguments into address (A) registers.

**Lines 27–35**
Contain a loop of 8-byte load instructions and 8-byte store instructions that move longwords from the source address to the destination address. The counter in register `a3` is decremented by 8 during each pass through the loop (see line 33), and the pointers are incremented by 8 (see lines 31 and 32). The `leu` instruction in line 27 is an unsigned, less-than-or-equal-to comparison. The `jbra.f` instruction in line 28 is a conditional branch instruction that exits the loop.

**Lines 36–43**
Along with lines 45 through 52 (on the next page), and 54 through 58 (on the next page), copy a 4-byte, 2-byte, and 1-byte parcel, respectively, to decrement the byte count to zero.

**Figure 29 (continued)**
Example of bcopy

```
45      htest:
46              leu.w #2,a3         ; can copy one halfword?
47              jbra.f btest        ; no--try bytes
48              ld.h (a2),s0        ; yes--load a halfword
49              st.h s0,(a1)        ; store a halfword
50              add.w #2,a1         ; bump destination pointer
51              add.w #2,a2         ; bump source pointer
52              sub.w #2,a3         ; decrement count
53
54      btest:
55                leu.w #1,a3       ; one last byte ?
56                jbra.f done       ; no--finish up
57                ld.b (a2),s0      ; yes--load a byte
58                st.b s0,(a1)      ; store a byte
59
60  done:
61                rtn               ; return
62
63                .comm src, 24     ; allocate 24-byte space called "src"
64                .cdata src        ; switch sections to "src"
65                ds.w 1            ; declare a word - assign the value 1
66                ds.w 2            ; assign the value 2 to another word
67                ds.w 3            ; assign a word the value 3
```

**Lines 45–52**
Copy a 2-byte parcel to decrement the byte count to zero.

**Lines 54–58**
Copy a 1-byte parcel to decrement the byte count to zero.

**Lines 60–61**
The `rtn` instruction returns program control to the caller of `_Bcopy`.

**Lines 63–90**
Illustrate how `_Bcopy` is called and how `.comm` and `.cdata` directives are used.

**Lines 63–67**
Create and initialize a block of memory named `src` that is 24 bytes long. The initialized values are words containing the integers 1 through 3.

**Figure 29 (continued)**
Example of bcopy

```
68
69              .comm dest, 24    ; allocate 24 bytes to dest
70              .cdata dest       ; switch sections to "dest"
71              ds.w 6(0)         ; assign 6 words the value zero
72
73              .text             ; switch to the text section
74              .globl main       ; make main a global symbol
75
76        main: pshea #24         ; push the constant 24 - arg 3
77              pshea dest        ; push the address of dest - arg 2
78              pshea src         ; push the address of src - arg 1
79              mov sp, ap        ; set the argument pointer
80              pshea #3          ; push the count of arguments
81              calls _Bcopy      ; copy 24 bytes from src to dest
82              add.w #16, sp     ; remove the arguments from the stack
83              ld.w 12(fp), ap   ; reset the argument pointer
```

**Lines 69–71**
Create another block of memory, named `dest`, that is also 24 bytes long and initialized to zero.

**Line 73**
Switches back to the `.text` segment. The code that follows this line is placed after the `rtn` instruction on line 61.

**Line 74**
Makes `_main` a global symbol corresponding to a C program named `main`.

**Lines 76–81**
Push a set of arguments onto the stack. They are pushed onto the stack in reverse order; the C call is `_Bcopy (src, dest, 24)`. Line 79 sets the argument pointer to point to the argument packet. Line 80 pushes the argument count onto the stack for use by routines with variable-length argument lists. The argument count is unused by `_Bcopy`, but could be found at location `-4 (ap)` if needed. Line 81 calls `_Bcopy`, pushing a short call frame onto the stack.

**Line 82**
Removes the arguments and argument count from the stack.

**Line 83**
Restores the argument pointer, `ap`, from the call frame created when `_main` was called. This is unused here, but could be used in a longer procedure.

**Figure 29 (continued)**
Example of bcopy

```
84          rtn             ; return
85
86           .cdata src     ; switch to the section src
87           .=.+12         ; move over the initialized space
88          ds.w 4          ; the fourth word of src
89          ds.w 5          ; initialize the fifth word to 5
90          ds.w 6          ; initialize the sixth word to 6
```

**Line 84**

Returns program control to the caller of `_main`.

**Lines 86–90**

Create additional memory in the block of memory named `src`. The initialized values in these lines are words containing the integers 4 through 6. In line 87, the `.=.+12` increments the location counter past the initialized storage space in the first `.cdata src` directive on line 64. Without moving the PC in this way, the data would be reinitialized.

## Vector routine

Figure 30 contains a sample assembly-language program that performs an elementary vector operation. This code is essentially the SAXPY subprogram in the VECLIB library.

The FORTRAN call statement for this subprogram is as follows:

```
CALL SAXPY (n, a, X, incx, y, incy)
```

Refer to the *CONVEX VECLIB User's Guide* for further details on this function.

Figure 30
Vector routine

```
1       ;--- subroutine saxpy (n, a, x,incx, y,incy)
2       ;
3       ;    saxpy computes a real*4 linked triad.
4       ;
5       ;--- FORTRAN call.
6       ;
7       ;    This is an optimized CONVEX Assembly-Language
8       ;    implementation of the following FORTRAN subprogram:
9       ;
10      ;            subroutine saxpy (n, a, x,incx, y,incy)
11      ;            real*4 (*),y(*)
12      ;            lf ( n .le. 0 ) return
13      ;            lf ( a .eq. 0.0 ) return
14      ;            ix = 1
15      ;            iy = 1
16      ;            if ( incx .lt. 0 ) ix = 1 - (n-1) * incx
17      ;            if ( incy .lt. 0 ) iy = 1 - (n-1) * incy
18      ;            do 10 i = 1, n
19      ;            y(iy) = a * x(ix) + y(iy)
20      ;            ix= ix + incx
21      ;            iy= iy + incy
22      ;       10  continue
23      ;           return
24      ;           end
25
26                .globl _saxpy_
27
28      _saxpy_:                    ; entry point
29
```

**Lines 1-25**
Comments that describe the function of the routine. Characters following a semicolon on a line are comments and are ignored by the assembler.

**Line 26**
Is a `.globl` assembler directive specifying that the symbolic name `_saxpy_` is visible to other routines in the program. By convention, the FORTRAN compiler appends underscore characters to both ends of subprogram names.

**Line 28**
Defines the global symbolic name as the value of the current location counter. Because no program-section directive occurs before this line, the assembler begins using the `.text 0` section.

**Figure 30 (continued)**
Vector routine

```
30      ld.w @ 0(ap),s0              ; fetch n
31      ld.s @ 4(ap),s1              ; fetch a
32      ld.w @12(ap),a2              ; fetch incx
33      ld.w @20(ap),a4              ; fetch incy
34      ld.w 8(ap),a1                ; fetch )x
35      ld.w 16(ap),a3               ; fetch )y
36
37      lt.w #0,s0                   ; check n
38      xor s3,s3
39      brs.f exit                   ; exit immediately if n <= 0
40
41      eq.s s3,s1                   ; a : 0.0
42      mov a2,s2
43      mov a4,s4
44      shf #2,a2                    ; byte stride for x
45      shf #2,a4                    ; byte stride for y
46      brs.t exit                   ; exit immediately if a = 0.0
```

**Lines 30–33**

Load the values of the non-array arguments into scalar and address registers. These quantities are maintained in the registers throughout the routine for efficiency. The `@` character specifies indirect addressing for FORTRAN calls by address.

**Lines 34–35**

Load the addresses of the 2 array arguments into address registers. The unbalanced closing parentheses in the comment fields of these statements means "the address of."

**Lines 37, 39,and 98**

Perform the `if` test in the comment at line 12.

**Lines 38, 41, 46, and 98**

Perform the `if` test in the comment at line 13. The two instructions on lines 38 and 41 are used instead of the single instruction `eq.s #0.0,.s1,` so as to treat negative zero in IEEE floating-point mode as zero.

**Lines 42–45**

Perform indexing initialization for the strip-mining loop. These instructions are placed between the floating-point comparison instruction on line 41 and the conditional branch instruction on line 46 so their execution can occur in parallel with the comparison.

**Figure 30 (continued)**
Vector routine

```
47
48          le.w #0,s2             ; 0 : incx
49          mov.w s0,a5            ; n
50          shf #9,s2              ; segment stride for x
51          brs.t saxl             ; if incx >= 0
52          mul.w a2,a5            ; move to other end of x vector
53          add.w a2,al
54          sub.w a5,al
55
56    saxl: mov.w s0,VL            ; load first segment of x(i)
57           mov.w a2,VS
58           ld.s 0(al),v0
59           add.w s2,al           ; next )x
60           mul.s v0,sl,vl        ; a * x(i)
61
```

**Lines 48–54**

Perform the `if` test and dependent statement in the comment at line 17. This code, however, does not determine the value of variable `ix`. Instead, it determines the address of array element `x(ix)`.

**Lines 56–60**

Begin the vectorized processing of the loop in the comment lines 18 through 22. Lines 56 through 58 load the first strip of the `x` vector from memory. Line 60 performs the scalar-times-vector multiplication for the strip. The vector load will chain into the vector multiply because the result register of the load is an operand register for the multiply, and because there are no functional unit or register reservations.

If the value of argument `n` does not exceed 128, the entire `x` vector is loaded and multiplied; otherwise, the processing is restricted to the first 128 elements of `x`. Line 59 adds 128×`incx` to the register containing the address of the `x` vector; the resulting register value is the address of `x(ix)` when loop index `i` = 129. If `n` exceeds 128, this value is used in the loop in lines 82 through 96; otherwise, it is not used.

**Figure 30 (continued)**
Vector routine

```
62            le.w #0,s4             ; 0 : incy
63            mov.w s0,a5            ; n
64            shf #9,s4              ; segment stride for y
65            brs.t sax2             ; if incy = 0
66            mul.w a4,a5            ; move to other end of y vector
67            add.w a4,a3
68            sub.w a5,a3
69
70      sax2: mov.w a4,VS            ; load next segment of y(i)
71            ld.s 0(a3),v2
72            add.s v1,v2,v3         ; new y(i)
73
74            sub.w #128,s0          ; advance length
75            st.s v3,0(a3)          ; store new y(i)
76            lt.w #0,s0             ; check length
77            add.w s4,a3            ; next )y
78            brs.f exit             ; exit if done
79
```

**Lines 62–68**

Perform the `if` test and dependent statement in the comment at line 18. This is similar to the processing of lines 48 through 54. Putting this code here instead of before line 56 allows its execution to occur in parallel with lines 56 through 60. Furthermore, because lines 56 through 60 do not depend on lines 62 through 68, arranging the code as shown allows the vector operations to begin sooner. Once vector operations begin, the remainder of the execution time is dominated by the vector processing, and almost all scalar processing is "free."

**Lines 70–78**

Complete the vectorized processing for the first strip, begun in lines 56 through 60. Lines 70 through 71 load the first strip of the `y` vector. Line 71 chains into the vector add in line 72, and fills the first VL elements of vector register V3 with `a×x(ix)` + `y(iy)`. Line 75 stores V3 back into the `y` vector in memory. Line 77 adds 128×incy to the register containing the address of the `y` vector. Lines 74, 76, and 78 check to see if `n` exceeds 128. If not, line 78 branches to the return instruction in line 98.

**Figure 30 (continued)**
Vector routine

```
80      ;--- linked triad loop.
81
82      sax3: mov.w s0,VL              ; load next segment of x(i)
83            mov.w a2,VS
84            ld.s 0(a1),v0
85            add.w s2,a1              ; next )x
86            mul.s v0,s1,v1           ; a * x(i)
87
88            mov.w a4,VS              ; load next segment of y(i)
89            ld.s 0(a3),v2
90            add.s v1,v2,v3           ; new y(i)
91
92            sub.w #128,s0            ; advance loop
93            st.s v3,0(a3)            ; store new y(i)
94            lt.w #0,s0               ; check loop
95            add.w s4,a3              ; next )y
96            brs.t sax3               ; loop if not done
97
98      exit: rtn                      ; return to caller
99
100     ;--- end of _saxpy_
```

**Lines 82–96**
Form the strip-mine loop. Each iteration of this loop, except the last one, corresponds to 128 iterations of the DO-loop in lines 18 through 22. The last iteration processes the remaining 1 to 128 DO-loop iterations. The processing is identical to that in lines 56 through 60 and 70 through 78, except that the branch condition in line 96 is inverted from that in line 78 to repeat the strip-mine loop rather than skip it.

**Line 98**
Corresponds to the return statement in comment line 23.

## Sample parallel program

Figure 31 contains a sample C program that points to assembly-language code to execute row computations in parallel. Lines 1 through 90 make up the C program. The assembly-language code begins on line 93.

**Figure 31**
Parallel program

```
1       #include <sys/time.h>
2       #include <sys/resource.h>
3       struct rusage rusage;
4       struct tlmeval tml,tm2,tm3,tm4;
5       long long t1,t2,t3,t4;
6       int n, pcnt, d1,d2;
7       double f1,f2,z1,z2;
9       #define ARRSIZE 512
10      double a[ARRSIZE][ARRSIZE],b[ARRSIZE][ARRSIZE],c[ARRSIZE][ARRSIZE];
11
12      long long gettoc();
13      int multrow();
14
15      main()
16      { int i, j;
17
18      /* initialize a, b */
19      for (i = 0; i < ARRSIZE; i++)
20              for (j = 0; j < ARRSIZE; j++)
21              { a[i][j] = i+j;
22              b[i][j] = i*j;
23              }
24
25      /* Print headers */
26      printf( "n                          wall clock    cpu time");
27      printf( "mflops seq/par ratio \n");
28      for (n = 16; n <= ARRSIZE; n *= 2) /* let n = 16, 32, ... ARRSIZE */
29      {
30      /* execute row computations in parallel */
31         gettime(&t1,&tml);
32         pcnt = exfn(multrow,n);
33         gettime(&t2,&tm2);
34
35      /* execute row computations sequentially */
36         gettime(&t3,&tm3);
37         for(i = 1; i <= n; i++) multrow(i,n);
38         gettime(&t4,&tm4);
39
40      /* compute wall clock and cpu delta times */
41      /* compute mflop rates, print results for current n */
42         dl = deltatime(&tml,&tm2); d2 = deltatime(&tm3,&tm4);
43         z1 = (double)(t2-tl); z2 = (double)(t4-t3);
44         fl = n*n*n*2/z1; f2 = n*n*n*2/z2;
45      printf("%4d parallel (%d) %8d usecs  %8d usecs      %5.1f\n",
46              n,pcnt,(int)(t2-t1),d1,f1);
47         printf("%4d sequential %8d usecs X8d usecs      %5.1f    %4.2f\n",
```

**Figure 31 (continued)**
Parallel program

```
48                n,(int)(t4-t3),d2,f2, z2/z1);
49      }
50 }
51
52
53
54      gettime(t,tm)
55      /* set t = current time of century clock */
56      /* set tm = current exact user cpu usage */
57      struct timeval *tm;
58      long long *t;
59      {
60          getrusage(RUSAGE_SELF,&rusage);
61          *tm = rusage.ru_exutime;
62          *t = gettoc();
63      }
64
65      deltatime(t1,t2)
66 /* compute difference between two time values, return single int     */
67 /* overflows when diff is greater than 2**31 usec (about 2000 secs) */
68      struct timeval *t1,*t2;
69 { return ((t2->tv_usec - t1->tv_usec)+(t2-tv_sec - t1->tv_sec)*1000000);
70 }
71
72
73
74
75
76 multrow(i,n)
77 /* compute single row of matrix multiply */
78 /* use static matrices a,b,c */
79      int i,n;
80 {
81          int j,k;
82          i--;          /* adust i to be 0 - (n-1) */
83
84      for (j = 0; j < n; j++) c[i] [j]  = 0;
85
86      for (k = 0; k < n; k++)
87        for (j = 0; j < n; j++)
88                   c [i] [j ] += a[i] [k] *b [k] [j];
89
90      }
91
92
```

**Figure 31 (continued)**
Parallel program

```
93      ;       exfn(func,n) --
94      ;       Spawns as many parallel threads as possible. Each has
95      ;       a separate stack of exfnssize bytes. Each thread obtains
96      ;       the next available index and executes (*func)(index,n). When
97      ;       no more indexes are available, each thread executes a join
98      ;       instruction. The join terminates all but the last thread
99      ;       so that exfn returns single threaded. The number of times the
100     ;       stack is incremented determines the number of active
101     ;       threads during the computation. This value is the return
102     ;       value of exfn.
103     ; Registers x8000 - x801f are reserved by compiler and runtime
104     ; systems.
105     STKOFF = 0x8020
106     INDEX  = 0x8021
107             .data
108             .globl _exfnssize       ; exfnssize is made global so that it
109                                     ; may be changed by the caller if a
110                                     ; longer stack is required.
111             .align 4                ; align word on word boundary
112 _exfnssize: ds.w 0x10000
113             .text
114             .globl _exfn
115
```

Characters following a semicolon on a line are comments ignored by the assembler.

**Lines 1–90**
Make up a C program that computes comparative execution speeds of a row computation. Line 32 points to the assembly-language code to execute the computation in parallel. Lines 76 through 90 compute a single row per invocation.

**Lines 93–103**
Begin the assembly-language code and include comments that describe the function of the routine.

**Lines 104-114**
Define the global variables and segments for the program.

**Figure 31 (continued)**
Parallel program

```
116     exfn:
117             ulk STKOFF                ; initialize STKOFF as stack offset
118             ulk INDEX                 ; initialize INDEX as index count
119             sub.w a1,a1
120             snd.w a1,STKOFF           ; place a zero in STKOFF, set lock bit
121             snd.w a1,INDEX            ; place a zero in INDEX, set lock bit
122             spawn L1,fp               ; spawn other threads
123
124     L1:
125             ld.w _exfnssize,a1      ; increment sp by stack size for thread
126             inc.w STKOFF,a1
127             jbra.f L1
128             sub.w a1,sp               ; set individual stack for each thread
```

**Lines 116–121**

Initialize two communication registers for use in synchronizing access to common data between the threads. The `ulk` instruction clears the hardware lock bit for a particular communication register so that the `snd` operation will not block. The `snd` instruction places an initial value in the communication register and locks the register. This locking is necessary so that the `inc` instruction used in lines 126 and 135 will not block.

**Line 122**

Starts as many threads as there are processors available. All have identical FP, AP, and SP values.

**Lines 124–128**

Increment the `STKOFF` communication register by `exfnssize` (stack size) so that each thread will have a different SP. The `inc` instruction automatically synchronizes access to `STKOFF`. Note that `jbra.f L1` includes an extra instruction. The purpose of this apparent inefficiency is to prevent the automatic "potential deadlock detection" hardware trap from coming into play. In this code, `STKOFF` is known to have the lock bit clear only during another `inc.w` instruction; the `inc.w` instruction will succeed within a very small number of tries.

**Figure 31 (continued)**
Parallel program

```
129 L2:
130         ld.w 4(ap),a2           ; a2 = n
131         ld.w 0(ap),a3           ; a3 = function address
132
133 L3:
134         ld.w #1,a1              ; compute next counter, place in a1
135         inc.w INDEX,a1
136         bra.f L3
137         le.w a1,a2
138         jbra.f L4               ; If counter = n, then (branch to exit)
139
140         psh.w a2                ; push n
141         psh.w a1                ; push counter
142         mov sp,ap               ; set argument pointer
143         pshea #2                ; set argument count
144         calls (a3)              ; call func(a,b,c,index,n)
145         add.w #12,sp            ; pop arguments from stack
146         ld.w 12(fp),ap          ; restore ap to original value
147         br L2
```

**Line 129**
: Is the start of the main loop for `counter=1,n`. `INDEX` is stored in a communication register, and each access to it is synchronized by means of `inc.w` instructions. Thus, each thread will pick up the appropriate next value on each pass through the loop. When all values have been obtained, each thread executes a `join` instruction. The `join` instruction on line 149 causes all threads except the last one to terminate. The last thread uses the `STKOFF` value to determine the number of active threads and returns that value in `s0`.

**Lines 129–132**
: Fetch arguments from the shared portion of the stack.

**Lines 133–138**
: Keep the current `INDEX` in a communication register, allowing automatic synchronization with the `inc.w` instruction. As in Line 127, an extra instruction is included when the `inc.w` fails because of the other threads accessing the same communication register at the same time. This extra instruction prevents deadlock detection traps from invoking the kernel to reschedule the thread.

**Lines 140–147**
: Set up the function call. After the call, `ap` is restored, as registers might have been modified by the called subroutine.

**Figure 31 (continued)**
Parallel program

```
148 L4:
149         join                    ; now do joins until single threaded again
150
151         get.l STKOFF,s0         ; return thread count as curiosity
152         ld.w _exfnssize,s1
153         div.w s1,s0
154         rtn                     ; restores correct sp for calling routine
155
156
157
158
159  ;         gettoc - return the time of century register
160  ;         without overhead of syscall
161  ;         toc register is only available on C2 and C3 processors
162          .globl _gettoc
163 _gettoc:
164          mov toc,s0
165          rtn
166
```

**Line 149**

Is the point a thread reaches only when `counter > n`, so that all calls to `func` have been started. Each thread will terminate with the `join`, except the last one. The `join` instruction also prevents any more threads from being spawned. The last thread computes the thread count based on the value of the `STKOFF` communication register.

**Lines 150-154**

Compute and return the number of threads used.

**Lines 159-165**

This function returns the value contained in the time of century register.

# Advanced topics 15

This chapter describes general information and procedures for tailoring your assembly-language code. Specifically, this chapter discusses the following topics:

- Coding techniques
- Using macros and the preprocessor
- Optimizing performance
- Using `adb` and `csd` to debug programs
- Writing parallel programs

## Coding techniques

Knowing when to write code in assembly language is an important consideration in programming. The profilers available in the CONVEX Consultant debugging software (an optional product) are `prof` and `gprof`. These profiling tools can help identify CPU-intensive routines. To profile a program, specify the `-p` or `-pg` compiler option. After the program completes, run `prof` or `gprof`.

Generally, code your original algorithm in a high-level language and get your program to run completely (if slowly) before attempting to rewrite the code in assembly language. It is only worthwhile to forego the ease of maintenance and coding provided by a high-level language if the code is very slow. Assembly code is machine-dependent and not portable; it can, however, run faster.

## High-level language processors

Using a high-level language processor, such as C or FORTRAN, is a time-saving way to generate assembly-language code. To use `cc` or `fc` to generate assembly-language code, specify the `-S` option. The `-S` option causes assembly-language code to be written to a file with the same name as the compiler source file but with an extension of `.s`.

To optimize the code, specify one of the `-O` (optimization) options. The `-O` options tell the compiler to call the optimizer.

To further improve the code, you can manually tune the `.s` file, which contains working code that can be hand optimized instead of generated from scratch.

## Coding hints

Often, no matter how much improvement is made in the code for an algorithm, the program still runs too slowly. When this happens, search for a better algorithm rather than attempt to enhance an existing one. The book, *The Elements of Programming Style*, by Kernighan and Plauger, has several hints on how to restructure programs for greater speed.

When writing assembly-language code, follow the normal methods of subroutine linkage and register use. Do not hesitate, though, to use the registers cleverly (for example, use registers instead of local variables) and reuse them as necessary in code that does not require a variable to be available throughout the extent of the code (for example, a loop variable can often be reused).

To embed assembly-language statements in C code instead of writing an entire routine in assembly language, use the `asm` directive. Following is an example that uses the `asm` statement:

```
asm ("dsi");          /* disable interrupts */
for (i = 0; i < 10; i++) {
   if (interr[i]) {
      blast[i] = ON;
   }
}
asm ("eni");          /* enable interrupts */
```

## Using macros and the preprocessor

To make your assembly-language code more readable or to tailor your code to a particular application, use macros and the preprocessor.

The `m4` macro processor provides a collection of built-in macros and allows you to define new macros. To define a new macro, enter

```
define (macro_name, macro_text)
```

where *macro_name* is an alphanumeric string that begins with a numeral (underscore, _, counts as a numeral), and *macro_text* is any text that contains balanced parentheses. Subsequent occurrences of *macro_name* are replaced with *macro_text*.

The assembler does not provide a macro capability, but the C preprocessor does. To use the standard C preprocessor for macros and defined constants, enter

```
/lib/cpp -pcc file.s > temp.s
```

where *file* is the name of your assembly-language file and *temp* is the name of your output file. Then, invoke the assembler.

## Optimizing performance

Understanding and using the basic concurrency of CONVEX supercomputers, as well as its pipelining, chaining, overlapping, and parallelizing abilities, can greatly increase the performance of the assembler.

During processing, the CPU passes instructions between multiple asynchronous processing units. These units are interconnected through high-speed, 64-bit buses and can operate concurrently. Pipelining occurs when the CPU breaks an instruction into separate parts so that the parts are processed simultaneously by different processing units.

The processing units also have caches that store information and can be used to speed up processing. Once an instruction has retrieved an address from main memory, the address resides in a cache within a processing unit. By ordering assembly-language instructions so that the assembler can retrieve addresses from the cache instead of having to go to main memory, you can cut down on processing time.

## Optimizing scalar code

When optimizing scalar code, you can use caches more effectively if you order the instructions so that all load instructions are performed before any operations, and all operations are performed before any store instructions.

## Optimizing vector code

When optimizing vector code, remember that there are 3 fully pipelined functional units that operate concurrently. The first unit performs add, subtract, and logical operations. The second performs load and store functions. The third performs multiply and divide operations. Vector edit functions operate in the second unit in the CONVEX C1 Series architecture and in the third unit on the CONVEX C2 Series architecture. These units can operate concurrently on 3 streams of independent data or can be chained together to operate on 1 data stream.

To optimize the code, then, rearrange instructions so that intrinsics that use the same functional unit are not sequential. For example, if you need to do both multiply and divide intrinsics, do not put the instructions in sequence. Split the multiply and divide functions among other functions, such as add, load, or store.

To further speed up the assembler, do not put 3 references to a specific vector register bank in 2 sequential instructions. For example, the instructions

```
add.d V1,V1,V1
mul.d V1,V3,V4
```

are processed sequentially instead of simultaneously because there are 3 references to `V1`. If, instead, the 2 instructions are

```
add.d V0,V1,V2
mul.d V2,V3,V4
```

the instructions are processed simultaneously. This process is called vector chaining. The functional unit that performs vector operations and scalar floating-point operations is separate from the functional unit that performs operations with the address of registers. This distinction permits overlapping execution of the instructions. For example, scalar loads and vector compares are executed concurrently.

If you are trying to optimize code that has been called by the FORTRAN compiler, be careful not to modify argument packets. Some of the argument packets can be precompiled. Because you cannot tell whether an argument packet has been precompiled, follow the general rule of not modifying any argument packets.

## Debugging with adb, csd, and CXdb

The adb and csd debuggers operate on CONVEX supercomputers. The csd debugger is not helpful, however, when debugging assembly-language code. In contrast, the CXdb debugger can debug assembly-language code, as well as mixed-language code.

The adb debugger is an object-code debugger that requires no special support from the loader or compilers. To use adb, enter

```
adb [-w] [ object_file [core_file]]
```

where *object_file* is an executable ConvexOS file and *core_file* is a core-image file produced by the operating system when a job terminates abnormally. The -w option creates both *object_file* and *core_file* (if necessary) and opens them for reading and writing so that adb can modify files.

For more information on csd, refer to *CONVEX Consultant User's Guide* (DSW-025). For more information on CXdb, refer to *CONVEX CXdb Concepts Guide* (DSW-471).

## Parallel programming in CONVEX assembly language

This section addresses commonly used parallel constructs and discusses managing the parallel execution of the threads of a process. Before continuing, you should be familiar with communication registers, synchronization instructions, and CPU control instructions (refer to the sections "Communications registers," on page 69, "Synchronization instructions," on page 93, and "CPU control instructions," on page 95, respectively, to review these topics). Refer to the section "Sample parallel program," on page 159, for an example of parallel code. Specifically, this section describes:

- Processes and threads
- Instructions to begin and end thread execution
- Communication between threads
- Thread memory management

## Processes and threads

The ConvexOS operating system recognizes two types of data flow: processes and threads. A process, in the traditional sense of the word, is the execution of a program and associated data. To parallelize at the process level requires that you use system calls such as `fork` and `exec`. A thread is a single stream of execution within a process. A single process can create many threads, each sharing a common memory and kernel environment. To parallelize at the thread level requires that you use one of two programming schemes. One scheme uses `pfork`, `wfork`, and `cfork` instructions. The other scheme uses `spawn` and `join` instructions. (For further information on these instructions, refer to the section, "CPU control instructions," on page 95.)

These programming schemes represent different ways of synchronizing resources and allocating available CPU time. The following sections specifically address how to use these instructions to begin and end thread execution.

## Creating threads

The `pfork` (post a fork) and `spawn` instructions divide a process into threads for execution. The `pfork` instruction requests that one additional thread start executing. It is a dynamic, on-call request for an additional thread. The `spawn` instruction requests that as many threads start executing as there are processors available. It is a static, one-time request.

### Processor scheduling

The `pfork` and `spawn` instructions take advantage of the CONVEX architecture's automatic self-allocating processors (ASAP). ASAP works as follows: if a processor is idle when it encounters a `pfork` or `spawn` request, a new thread is created and starts executing. If a processor is occupied, then the request for an additional thread remains pending, meaning that it remains posted until a processor becomes available or the request is cleared. If a thread request is already pending when either a `pfork` or `spawn` is executed, then the operation fails, and the address-carry bit (C) in the PSW is reset to zero.

## Note

**Do not mix `spawn` and `pfork` instructions without proper synchronization.**

### Special registers

When a thread is started, the contents of its registers are undefined except for the argument pointer (AP), stack pointer (SP), frame pointer (FP), and program counter (PC). A thread assumes the following values of these registers:

- AP (usually A6) is the same as the thread that initiates the `pfork` or `spawn` instruction.
- FP (usually A7) is the same as the thread that initiates the `pfork` or `spawn` instruction.
- SP is initially the value of FP.
- PC is the instruction address; `pfork` and `spawn` instructions include fields for an address and an address (A) register, as illustrated in the following example:

```
spawn L1, fp  ; fp = A6
```

## Ending threads

Two instructions return a process to a single thread: `wfork` (wait for a fork) and `join`. The `wfork` instruction causes the thread to terminate, and waits for a pending thread. It does not, however, clear pending thread requests. The `wfork` instruction works as follows: if the thread belongs to a process that has a request for a new thread posted, then the new thread is created. If the thread belongs to a process that does not have a request for a new thread, but does have active threads on other processors, then another process is scheduled for the current processor.

**Caution**

**If the current thread is the only active thread in the process and no other request for a new thread is pending, then the kernel receives a deadlock-detection interrupt, and the process is terminated.**

Deadlock occurs when the threads of a process are waiting for a resource that is unavailable. The last thread of a process should never encounter a `wfork`, because `wfork`, by definition, looks for pending thread requests. Be sure that at least one thread will remain active before specifying a `wfork` instruction.

Use the `cfork` (clear a fork) instruction with `wfork` to clear any pending thread requests generated by a `pfork` or `spawn` instruction.

The `join` instruction works as follows: each thread executes a `join` after it has been processed. The `join` instruction clears any pending thread requests for the current process. If more than one thread is active when a `join` is executed, then the CPU terminates the current thread. Terminating the thread relinquishes and deallocates the CPU. If the current thread is the last active thread of the current process, then the process continues with the next instruction. After all threads have executed a `join`, the process will only have one thread running.

## Communicating between threads

A group of instructions included in the CONVEX C2 and C3 Series instruction sets work specifically to synchronize activities between threads of a process and the related communication registers (refer to the section, "Synchronization instructions," on page 93).

Each communication register has a lock bit that can be used to synchronize access to the register. The `lck` instruction sets the lock bit. If the lock bit is clear, then C is set. If the lock bit is already set, then C is cleared to indicate failure. The `ulk` instruction clears the lock bit, and C is set to the previous value of the lock bit.

Generally speaking, you should follow a synchronization instruction with a `jbra.f` or `jbrs.f` instruction to branch back and repeat the synchronization instruction until it succeeds. These instructions use the address-carry bit (C) or scalar-carry bit (SC) to indicate success or failure.

## Thread memory management

A thread's memory map is by default the same as that of the initiating thread, so all threads of a process share the same memory. A process or thread, however, may have either *shared* or *unshared* memory.

### Shared memory

Shared memory means that more than one thread may use the same logical address to read from or write to the same physical location in memory. Thread data segments are shared by default.

### Unshared memory

Unshared memory means that each thread wants to use the same logical address to access different physical locations in memory. Two registers on each CPU assist in managing unshared thread memory: the communication index register (CIR) and the thread identifier (TID).

- CIR—Serves as the primary point of reference between the CPU, a ConvexOS process, and the communication registers. The C200 Series hardware supports 8 sets of 128 communication registers. Each CPU is linked to a set of communication registers by means of the CIR, which is assigned an appropriate value by the operating system. When a process divides into threads for execution, the CIR tracks which subset of communication registers is being used by the CPU.
- TID—Subdivides a process into individual pages of physical thread memory. TID is usually used with unshared memory because it makes each thread unique in the translation from logical to physical memory.

### Thread data segments and thread stacks

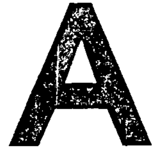
Thread data segments are shared by default. To create data segments with unshared memory, use the assembler directives `.data` and `.tbss`. These directives work like `.data` and `.bss`, except that address space is *private*. Thread private means that each thread gets a private copy of variables and data declared in `.data` and `.tbss`.

Threads with unshared memory require separate stack spaces. The behavior of programs that do not have separate stack spaces for different threads is difficult to debug. To create individual thread stacks, use the `.tbss` directive to allocate data space and then assign a thread SP to that space.

## Further reference

For more information on multithreaded parallel programs, refer to the *CONVEX C Series Architecture Reference Manual*. For information on debugging multithreaded parallel programs, refer to *CONVEX adb (Assembly-Language Debugger) User's Guide.*, *CONVEX Consultant User's Guide*, and *CONVEX CXdb Concepts*.

# Loader error messages



The following error messages are produced by the loader. This appendix lists and explains only a few of the many warnings and errors that the loader utility can produce. The messages listed in this appendix are the most ambiguous ones.

---

```
%s: vaddr (%#llx) and d_off (%#llx) disagree
```

The loader prints this message when your program has tried to lay out the virtual addresses of the executable image.

---

```
__.SYMDEF entry has disappeared from '%s'
Unable to read __.SYMDEF archive entry in '%s'
```

These messages indicate that the library is malformed (`__.SYMDEF` concerns libraries on which the `ranlib` utility has executed). Contact your system manager to see if executing `ranlib` on the library again fixes it.

```
Unknown FP mode in o_flags (%llx)
ICB '%s' initializers overlap: %s(%#llx,%#x) and
%s(%#llx,%#x)
ICB symbol '%s' has disappeared
ICB symbol not N_SCNHDR type (%s %d)
```

where:

FP
: Floating point

ICB
: Initialized common block

The first ICB message above indicates that two separate files attempted to initialize the same elements of a common block. Refer to the section, "Handling initialized common blocks," on page 47, and the section, "Initialization data format (ICBs)," on page 48, for more information.

```
unable to seek to ICB TOC's in '%s'
unable to write ICB TOC count in '%s'
unable to write ICB TOC entry in '%s'
```

where:

TOC
: Table of contents, which has to do with the way the loader represents common blocks.

If these error messages appear, it could indicate a problem with file permissions.

```
bad obj_type() return value (%d)
bad filetype() return value (%d)
make_symbol() called with local symbol (%s)
make_symbol() called with STAB symbol (%s)
Invalid symbol type (%s %d) in reloc_one_symbol()
```

These messages indicate internal loader errors because they mention internal routines. Use the `contact` utility to report these problems to the CONVEX Technical Assistance Center. Include a copy of your program that caused the error, if possible.

---

```
Symbol entry for common block '%s' is gone
```

This message indicates another internal loader error.

---

```
external symbol '%s' not found for symvec
```

symvec is a data structure internal to the loader. Use the `contact` utility to report this problem to the CONVEX Technical Assistance Center. Please include files you were trying to link.

# Glossary

This glossary contains definitions of terms that aply to the use of the assembler and the loader utilities described in this document.

## A

### absolute expressions

Expressions that contain only terms that do not need to be relocated and can be resolved at assembly time.

### argument pointer (AP)

Pointer to the location on the runtime stack of the first argument passed from the calling routine to the current routine.

### assembler directive

A command to the assembler that either alters its operation or causes it to reserve storage for data values.

## C

### critical region

A region of code that modifies data or computer resources shared by more than one process.

## D

### dot

The . character, which serves as a symbolic name for the value of the current assembler location counter.

## E

### expressions

Combination of constants and symbolic names joined by operators.

### extended op codes

Op codes that have halfword prefixes modifying the behavior of the instruction.

### external expressions

Expressions that contain at least one external symbolic name.

## F

### frame pointer (FP)

Pointer to the last frame pushed on the runtime stack by a subprogram call.

## I

### instruction mnemonic

A symbolic name for a machine instruction.

### instruction typing

Term referring to the fact that each CONVEX assembly-language instruction is associated with a machine type in the CONVEX family of supercomputers.

## M

### multiprocessing

The creation and scheduling of processes on any subset of CPUs in a system configuration.

## N

### name label

A user-defined symbolic name that allows instructions to refer to a specific address within the program.

## O

### operations under mask

Term referring to the ability to include or exclude different vector elements in the instruction depending on whether the vector-merge bit for an element is set to 1 or 0.

## P

### prefix op code

Halfword value that precedes a standard op code and modifies its behavior.

### process

A collection of one or more instruction streams executing within a single logical address space.

### processor status word (PSW)

A 32-bit register that contains coded bits indicating the current state of the processor.

### program counter (PC)

A 32-bit register that contains the address of the next instruction to be executed.

### program segment

A block of code identified by a program-segment directive.

### PSW

*See* processor status word.

## R

### relocatable expressions

Expressions that reference addresses that cannot be resolved at assembly time.

## S

### stack pointer (SP)

Pointer to the top of the runtime stack, above which the next activation record is allocated.

T

### terms

Constants and symbolic names that make up an expression.

### thread

Any single instruction stream executing within a process.

### thread `.bss`

Uninitialized data that is unshared among all threads comprising a process, but is unique to that one thread.

### thread data

Initialized data that is unshared among all threads comprising a process, but is unique to that one thread.

# Index

## A


a.out file 3, 15
absolute expressions 118
  defined 181
  in expressions 118
absolute-addressing mode 128
adb
  assembly-language debugger 65, 171
adb option
  -w 171
add, vector 69
address carry bit 174
address registers 68, 129
address resolution 24, 25, 41
  between modules 63
addressing modes 122
  immediate 126
  memory 128, 129, 130, 131
  register mode 122
.align
  directive 104
  example 104
  using with .pad 105
alignment
  code and data 104
  storage s_align 38
alignment directives 104
AP
  *See* argument pointer
ar command format 53
  key characters 54
ar program 3, 51, 53, 59, 63
architecture overview 67
argument packets 134
argument pointer (AP) 68, 133, 134
  defined 181
arithmetic instructions, table 85
ASAP scheduling 172
asm directive in C code 168
assembler
  invoking 145
  optimum use of 66
  use of ASCII character set 113
assembler directive 76
assembler directives 97
  defined 181
assembler escape sequences, table 103
assembler object output, redirecting 147
assembler utility 3
assembly-language debugger 65
assignment, symbolic name 120
assistance, calling TAC xix


## B


backslash escape sequence
  use in strings 103
backspace escape sequence
  use in strings 103
_Bcopy example 149
benefits of loaders 1
bibliography xviii
binary and unary operators, table 114
bit pattern escape sequence
  use in strings 103
bits
  address carry 174
  E bit 80
  prefix code 80
  scalar carry 174
  vector merge 182
block-storage directive 101
bs directive 101
  example 101
.bss
  effect of .comm directive 108
  segment 98
bss segment 22


## C


C compiler 168
  interface example 63
C language
  calling conventions 140
  function arguments 141
  function call code generation 140
  function name generation 141
  function return values 141
C library 51
C math library 51
C preprocessor 169
C1 register sets 68
C2 register sets 68
C3 register sets 68
calculating effective address 82
call instruction 134, 136
calling a FORTRAN subroutine
  figure 142
calling conventions
  C language 140
  FORTRAN language 142
  general 136
  linkages 135

calling sequence
  function or subroutine 137
callq instruction 135
calls instruction 134, 135
carriage return escape sequence
  use in strings 103
cautions
  IEEE hardware required for IEEE object files 147
  invoke the loader with a compiler 9
  pointer fields in SOFF format are long long 29
  threads and deadlock-detection 173
  *See also* notes
cc command 168
.cdata
  following .comm directive 108
  multiple directives 99
  segment 98, 99
cfork 96
  instruction 172
changing execution modes 12
character constants in terms 116
character data 67
character set, assembler 113
chmod 13
CIR
  communication index register 69, 175
coding hints 168
coding techniques 167
.comm
  directive 107, 108
  effect on .bss segment 108
  effect on .tbss segment 108
  format 108
  preceding .cdata directive 108
  use with .cdata 99
command format
  examples of ld 10
command line options available 12
commands
  ar 3
  cc 168
  fc 168
  gprof 167
  ld 10
  m4 169
  nm 58
  prof 167
  ranlib 3
comment field 77
common blocks 46
  initialized 36, 47
common-block symbol 46
communicating between threads 174
communication register 68, 69, 70
  index register (CIR) 69, 175
  lock bit 70, 174
compiler options 168
compiler-generated code 134
compilers 66
  C 168
  FORTRAN 168
constants
  character 116
  numeric 115
  symbolic 116
Consultant software package 167
conventions
  C calling 140
  notational xvii
CONVEX architecture overview 67
copy block of memory
  example 149
  figure 149
counter, location 74, 75, 76
CPU control instructions 95
creating thread stacks 175
creating threads 172
critical region
  defined 181
csd debugger 171


# D


-D option 12, 15
data
  segment 98
data relocation commands 23
data segment 22
  threads 175
data-conversion instructions 88
data-type specifiers, table 87
deadlock, defined 173
debugger, assembly-language 65
debugging
  adb 171
  csd 171
define-storage directive 101
defining macros 169
demand-paged mode 12
diagram of loader functions 2
directives
  .align 104
  alignment 104
  asm in C code 168
  block-storage 101
  bs 101
  bs example 101
  .bss 98
  .cdata 98, 99
  .comm 99, 107, 108
  creating unshared thread memory 175
  .data 98, 175
  define-storage 101
  dividing into subsections 100
  ds 101
  effect of .comm on .bss segment 108
  effect of .comm on .tbss segment 108

directives (continued)
effect of .tcdata on .tdata segment 109
external symbolic-name 107
floating-point 105
.fpmode 105
.globl 107
.pad 105
preceding .cdata with .comm 108
program-segment 98
.stabd 109
.stabn 109
.stabs 109
storage 101
symbol-table 109
.tbss 98, 175
.tdata 98
.text 98
using .align with .pad 105
documentation
ordering xix
dot (.) character 120
defined 181
double quote escape sequence
use in strings 103
ds directive 101
example 102
format 101
use with .bss segment 102
use with .tbss segment 102
use with .text segment 102
ds directive example 102

# E

E bit 80
-E demand 12
-E nonswap 12
-E option 12, 13
-E prepage 12
effective address calculation 82
ending threads 173
entry address 23
error messages
loader 177, 179
error messages, format 148
error utility 66
escape sequence
backslash 103
backspace 103
bit pattern 103
carriage return 103
double quote 103
form feed 103
horizontal tab 103
newline 103
single quote 103
escape sequences in strings 103
examples
.align directive 104
_Bcopy 149
bs directive 101
copy a block of memory 149
ds directive 102
ds directive repeat factor 102
.fpmode directive 105
.globl directive 107
name labels 74
.pad directive 105
parallel program 159
temporary labels 75
vector routine 154
exec system call 172
expressions 117
absolute expressions 118
defined 182
external expressions 118
relocatable expressions 117
extended op code 80
defined 182
operations under mask, format 80
external expressions 118
defined 182
in expressions 118
external references
resolution of 5
external symbolic names
defined 118
undefined 119
external symbolic-names
directives 107

# F

-F option 13, 15
fc command 168
files
library 2
object 2
flags
n_type 43
flags word 4, 80
floating-point constants 115
floating-point directives 105
floating-point format
-fi 13
-fn 13
floating-point modes
IEEE 13
native 13
floating-point numbers
as operands 102
fork system call 172
form feed escape sequence
use in strings 103

format
.comm directive 108
.globl directive 107
.tcdata directive 109
bs directive 101
ds directive 101
FORTRAN compiler 168
FORTRAN language
calling conventions 142
function arguments 143
function call code generation 142
function return values 143
subprogram names 142
subroutine arguments 144
subroutine return values 144
FORTRAN libraries 52
I/O 52
libD77.a 52
libF77.a 52
libI77.a 52
libU77.a 52
libV77.a 52
*See also* libraries
FP
*See* frame pointer
.fpmode
directive 105
example 105
use with -fi option 106
use with -fn option 106
frame pointer (FP) 68, 133, 134
defined 182
function arguments
C 141
FORTRAN 143
function call code generation
FORTRAN 142
function call code generation, C 140
function calling sequence 137
function names
C 141
function return values
C 141
FORTRAN 143
functions of the loader 1, 4
further reference xviii

## G

general register operands 124
.globl 107
directive 107
example 107
format 107
gprof command 167

## H

-H option 13
hardware lock bit 70
header format 21
description of 22
header format (SOFF)
description of 29
description of fields 30
headers
(SOFF) segment 35
high-level langauges 168
hints, coding 168
horizontal tab escape sequence
use in strings 103

## I

IEEE mode 13
IEEE translation 106
immediate addressing mode 126
immediate operands 126
indexed mode 129
indirect loader access, example of 63
indirect-absolute mode 130
indirect-deferred mode 130
indirect-indexed mode 131
initialized common blocks 36, 47
initialized storage allocation 101
input to the loader 2
instruction format 73, 75, 77
comment field 77
figure 73
label field 74
label field, name labels 74
label field, temporary labels 75
memory reference 83
operand field 76
operation field 76
operation field, assembler directives 76
operation field, instruction mnemonics 76
terminator field 77
instruction mnemonic 76
defined 182
instruction set
general 80
instruction typing 4, 80
defined 182
instructions
call 134
calls 134
cfork 96, 172
jbra.f 174
jbrs.f 174
join 96, 172, 173, 174
lck 93, 174
pfork 96, 172, 173
pop 133
psh 133

instructions (continued)
- rcv 94
- rtn 134
- rtnc 134
- rtnq 135
- snd 94
- spawn 96, 172, 173
- types of, arithmetic 85
- types of, CPU control 95
- types of, data-conversion 88
- types of, logical 86
- types of, machine-control 92
- types of, memory reference 81
- types of, program-control 90
- types of, span-independent program-control 91
- types of, synchronization 93
- types of, vector-reduction 89
- ulk 93, 174
- wfork 96, 172, 173

internal references
- diagram 5

invoking the assembler 145


## J


jbra.f instruction 174
jbrs.f instruction 174
join instruction 96, 172, 173, 174


## L


-L option 13
label field 74
- name labels 74
- temporary labels 75

layout of relocation information (SOFF) 41
layout of symbol table entries 2
lck instruction 93, 174
ld command format 10
- examples of 10

length, vector 69
libD77.a, FORTRAN library 52
libF77.a, FORTRAN library 52
libI77.a, FORTRAN library 52
libraries
- C 51
- FORTRAN 52
- I/O 52
- libF77.a 52
- libI77.a 52
- libU77.a 52
- libV77.a 52
- math 51
- object 51
- system 51

library
- file generation, diagram 3
- files 2
- searches 5

libU77.a, FORTRAN library 52
libV77.a, FORTRAN library 52
linkages 135
- call instruction 136
- callq instruction 135
- calls instruction 135

load map 14
load, vector 69
loader 66
- benefits of 1
- function of 1
- functions 4
- input 2
- output 3

loader error messages 177, 179
loader functions
- diagram 2
- iterative loading 6
- library searches 5
- object-file linkage 4
- resolving external references 5

loader use, examples of 63
location counter 75, 76, 120
- dot (.) character 120

location-counter-directive sequences, table 121
lock bit, communication register 174
logical instructions 86
- table 86

longword, memory structure 82


## M


-M option 14
m4 macro processor 169
machine instructions, format 80
machine op codes 81
machine-control instructions 92
macro processor, m4 169
macros and the preprocessor 169
magic number 22
make utility 65
manual coding 167
mask, operations under 89
memory longword structure, figure 82
memory management
- thread 174

memory, copy block of 149
memory-addressing modes 128
memory-reference instruction
- figure of formats 83

memory-reference instructions 81
- format 83
- table 81

mnemonics, instruction 76

modes
  absolute-addressing 128
  addressing 122
  demand-paged 12
  immediate addressing 126
  indexed 129
  indirect-absolute 130
  indirect-deferred 130
  indirect-indexed 131
  memory-addressing 128
  nonswapped 12
  prepaged 12
  register 122
  register-deferred 129
multiple scan of libraries 14
multiply-defined symbols 11
multiprocessing
  defined 182
multiprocessor management hardware 67

## N

n_type flags 43, 45
name label
  defined 182
name labels example
  figure 74
native mode 13
newline escape sequence
  use in strings 103
-NL option 14
nm command 58
nonswapped mode 12
notational conventions xvii
notes
  default floating-point mode 13
  difference between file header and optional header 32
  do not mix spawn and pfork without synchronization 172
  file header stores SOFF information only 32
  order of function arguments on stack 142
  reserved communication registers 70
  sod cannot examine stripped files 60
  specify only one executable file as input to loader 17
  using branch back instructions with rcv 94
  using branch back instructions with snd 94
  *See also* cautions
numeric constants in terms 115

## O

-o option 15
object file 2
  format 21
  header format 21
object file (continued)
  linkage 4
  linkage, internal references 4
object libraries 51
object modules 3
object output
  redirecting assembler 147
op codes
  extended 80
  machine 81
operand field 76
operands
  floating-point numbers 105
  general-register 124
  immediate 126
operation field 76
  assembler directives 76
  instruction mnemonics 76
operations under mask 89
  .f suffix 89
  .t suffix 89
  defined 182
operations, pseudo 97
operators 114
optimizing performance 169
  scalar code 170
  vector code 170
optimum use of the assembler 66
optional header (SOFF)
  description of 32
  description of fields 33
  layout of 32
options
  -fi 106
  -fi and .fpmode directive 106
  -fn 106
  -fn and .fpmode directive 106
  -l 147
  -o 147
  -p 167
  -pg 167
  -s 12
  -w 148, 171
  -x 12
  compiler, -O 168
  compiler, -S 168
  compiler, generate assembly code 168
  compiler, optimize code 168
  -D 12, 15
  directing object output 147
  -E 12, 13
  -F 13, 15
  floating-point format 106
  generate source listing 147
  -H 13
  -L 13
  -l 13
  -M 14
  -NL 14
  -o 15
options (continued)
  order of 11

-p 15
profiling 167
-r 4, 12, 15
redirecting object output 147
-s 15
specifying 11
suppressing warnings 148
-T 15
-u 16
-v 16
-w 16
-X 16
-y 16
ordering documentation xix
organization of this book xvi
output of the loader 3


# P


-p option 15
packets
runtime 134
.pad
directive 105
example 105
using with .align 105
parallel processing, general 67
parallel programming in assembly-language 171
performance, optimizing 169
pfork 96
instruction 172
pfork instruction 172, 173
pop instruction 133
prefix code bit 80
prefix op code 80
defined 183
prepaged mode 12
preprocessor and macros 169
process
defined 183
processes and threads 172
processing
parallel 67
vector 157
processor scheduling 172
processor status word (PSW)
defined 70, 183
prof command 167
profiling 167
program counter
.cdata segment 99
program counter (PC)
defined 183
program segment
defined 183

program segment directives
.bss 98
.cdata 98
.data 98
.tbss 98
.tdata 98
.text 98
program-control instructions 90
program-segment directives 98
programs
ar 51, 53, 59
ranlib 51, 59
sod 51, 59
pseudo-operations 97
psh instruction 133
PSW
*See* processor status word


# R


-r option 4, 12, 15
ranlib program 3, 13, 51, 59, 63
rcs utility 66
rcv 94
redirecting assembler object output 147
register mode 122
register sets 68
register-deferred mode 129
registers
address 68, 129
argument pointer 134
communication 69, 70
compiler-generated code 134
frame pointer 134
S0 134
scalar 68, 69, 124
scalar register 134
special 173
special address 133
special purpose 125
stack pointer 133
supporting vector processing 67
vector 68, 69, 124
vector add 69
vector length 69
vector load 69
vector store 69
vector stride 69
relocatable expressions 117
defined 183
relocatable symbol
defined 46
relocation entries 24
data 24
description of fields 24
text 24
relocation entry (SOFF) 41
description of 41
description of fields 42, 43
relocation information (SOFF) 41
reprocessing executable files 16

resolution of symbol references 11, 16
resolution of symbolic addresses 24, 41
rtn instruction 134
rtnc instruction 134
rtnq instruction 135

## S

-s option 15
S0
  scalar register 134
s_align 38
s_vaddr 38
sample program
  copy block of memory 149
  parallel program 159
  vector routine 154
scalar carry bit 172, 174
scalar code
  optimizing 170
scalar register
  S0 134
scalar registers 68, 69, 124
scheduling
  processor 172
segment
  .bss 98
  .cdata 98, 99
  .data 98
  .tbss 98
  .tdata 98
  .text 98
segment data (SOFF) 40
segment headers (SOFF)
  description of 35
  description of fields 38
  layout of 37
semaphore bit 70
shared memory
  threads 174
single quote escape sequence
  use in strings 103
snd 94
sod program 51, 59
source listing
  -l option 147
SP
  *See* stack pointer
span-independent program-control instructions 91
spawn 96
  instruction 172
spawn instruction 172, 173
special address registers 133
special purpose registers 125
special registers 173
specification
  data-type 87
specifying subsections
  .bss directive 100
  .cdata directive 100
  .data directive 100
  .tbss directive 100
  .tdata directive 100
  .text directive 100
.stabd, directive 109
.stabn, directive 109
.stabs, directive 109
stack layout
  figure 138
stack pointer (SP) 68, 133
  defined 183
stack spaces 98
stacks, threads 175
standard op code 80
storage alignment
  s_align 38
storage allocation
  initialized 101
  uninitialized 101
storage directives 101
store, vector 69
stride, vector 69
string table (SOFF)
  description of 49
string table, description of 28
strings
  directive argument 103
  examples 103
  termination by null byte 104
  valid escape sequences 103
strip mining 156
subprogram names
  C 141
  FORTRAN 142
subroutine arguments
  C 141
  FORTRAN 144
subroutine calling sequence 137
subroutine return values
  C 141
  FORTRAN 144
supplemental reading xviii
symbol
  common-block 46
  relocatable 46
symbol references
  resolution of 16
symbol table 5, 15, 16, 25, 43
  description of 25
  description of entries 25
  description of fields 26
  size of 23

symbol table (SOFF)
description of 43
description of fields 45
layout of 43
n_type flags 45
n_value field 46
symbol table entries
layout of 2
use of 25
symbol-table directives 109
symbolic name assignment statements 120
symbolic names in terms 116
synchronization instructions 93
system calls
exec 172
fork 172
system libraries 51
C library 51
C math library 51
*See also* libraries

## T

t key 57
-T option 15
table of contents key 57
TAC
technical assistance center xix
.tbss
effect of .comm directive 108
segment 98
.tcdata
effect on .tbss segment 109
effect on .tdata segment 109
format 109
.tdata
effect of .tcdata directive 109
segment 98
technical assistance center (TAC) xix
temporary labels example
figure 75
terminating threads 173
terminator field 77
terms 115
character constants 116
defined 184
numeric constants 115
symbolic names 116
.text
segment 98
text relocation 24
commands 23
text segment
definition of 23
size of 22
thread .bss, defined 184
thread data, defined 184
thread memory map 174
thread private 175
thread, defined 184
threads
communicating between 174
creating 172
creating unshared memory 175
data segments 175
ending 173
memory management 174
memory map 174
separate stack spaces 175
shared memory 174
stack spaces 98
stacks 175
terminating 173
unshared memory 98, 175
threads and processes 172
TID
thread identifier register 98, 175
top of the runtime stack, figure 136
trace command 15
type propagation in expressions 119
typing
instruction 80
typing, instruction 4

## U

-u option 16
ulk instruction 93, 174
uninitialized storage allocation 101
unshared memory, threads 175
user library creation, example of 63
utilities 65
error 66
make 65
rcs 66

## V

-v option 16
vector
add 69
length 69
load 69
merge 69
store 69
stride 69
vector code, optimizing 170
vector length (VL) register 69
vector merge (VM) register 69, 89
vector merge bit 69, 89
vector processing 158
vector reduction instructions 89
vector registers 68, 69, 124
vector stride (VS) register 69
VM register 89

## W

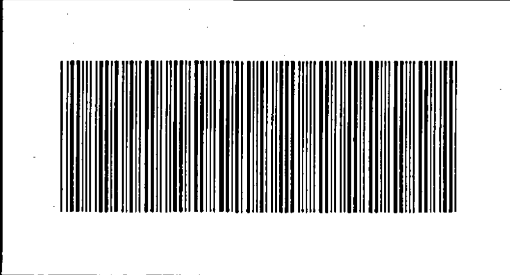
-w option 16
wfork
  instruction 172
wfork instruction 96, 173

## X

-X option 16

## Y

-y option 16



Order Number
DSW-096

Document Number
720-005130-000